Chapter 1 Python Primer

Contents

1.1 Python Overview

Building data structures and algorithms requires that we communicate detailed instructions to a computer. An excellent way to perform such communications is using a high-level computer language, such as Python. The Python programming language was originally developed by Guido van Rossum in the early 1990s, and has since become a prominently used language in industry and education. The second major version of the language, Python 2, was released in 2000, and the third major version, Python 3, released in 2008. We note that there are significant incompatibilities between Python 2 and Python 3. *This book is based on Python 3 (more specifically, Python 3.1 or later).* The latest version of the language is freely available at `www.python.org`, along with documentation and tutorials.

In this chapter, we provide an overview of the Python programming language, and we continue this discussion in the next chapter, focusing on object-oriented principles. We assume that readers of this book have prior programming experience, although not necessarily using Python. This book does not provide a complete description of the Python language (there are numerous language references for that purpose), but it does introduce all aspects of the language that are used in code fragments later in this book.

1.1.1 The Python Interpreter

Python is formally an ***interpreted*** language. Commands are executed through a piece of software known as the ***Python interpreter***. The interpreter receives a command, evaluates that command, and reports the result of the command. While the interpreter can be used interactively (especially when debugging), a programmer typically defines a series of commands in advance and saves those commands in a plain text file known as ***source code*** or a ***script***. For Python, source code is conventionally stored in a file named with the `.py` suffix (e.g., `demo.py`).

On most operating systems, the Python interpreter can be started by typing `python` from the command line. By default, the interpreter starts in interactive mode with a clean workspace. Commands from a predefined script saved in a file (e.g., `demo.py`) are executed by invoking the interpreter with the filename as an argument (e.g., `python demo.py`), or using an additional `-i` flag in order to execute a script and then enter interactive mode (e.g., `python -i demo.py`).

Many ***integrated development environments*** (IDEs) provide richer software development platforms for Python, including one named IDLE that is included with the standard Python distribution. IDLE provides an embedded text-editor with support for displaying and editing Python code, and a basic debugger, allowing step-by-step execution of a program while examining key variable values.

1.1.2 Preview of a Python Program

As a simple introduction, Code Fragment 1.1 presents a Python program that computes the grade-point average (GPA) for a student based on letter grades that are entered by a user. Many of the techniques demonstrated in this example will be discussed in the remainder of this chapter. At this point, we draw attention to a few high-level issues, for readers who are new to Python as a programming language.

Python's syntax relies heavily on the use of whitespace. Individual statements are typically concluded with a newline character, although a command can extend to another line, either with a concluding backslash character (\), or if an opening delimiter has not yet been closed, such as the { character in defining value_map.

Whitespace is also key in delimiting the bodies of control structures in Python. Specifically, a block of code is indented to designate it as the body of a control structure, and nested control structures use increasing amounts of indentation. In Code Fragment 1.1, the body of the **while** loop consists of the subsequent 8 lines, including a nested conditional structure.

Comments are annotations provided for human readers, yet ignored by the Python interpreter. The primary syntax for comments in Python is based on use of the # character, which designates the remainder of the line as a comment.

```
print('Welcome to the GPA calculator.')
print('Please enter all your letter grades, one per line.')
print('Enter a blank line to designate the end.')
# map from letter grade to point value
points = {'A+':4.0, 'A':4.0, 'A-':3.67, 'B+':3.33, 'B':3.0, 'B-':2.67,
          'C+':2.33, 'C':2.0, 'C':1.67, 'D+':1.33, 'D':1.0, 'F':0.0}
num_courses = 0
total_points = 0
done = False
while not done:
  grade = input( )                          # read line from user
  if grade == '':                           # empty line was entered
    done = True
  elif grade not in points:                 # unrecognized grade entered
    print("Unknown grade '{0}' being ignored".format(grade))
  else:
    num_courses += 1
    total_points += points[grade]
if num_courses > 0:                         # avoid division by zero
  print('Your GPA is {0:.3}'.format(total_points / num_courses))
```

Code Fragment 1.1: A Python program that computes a grade-point average (GPA).

1.2 Objects in Python

Python is an object-oriented language and ***classes*** form the basis for all data types. In this section, we describe key aspects of Python's object model, and we introduce Python's built-in classes, such as the **int** class for integers, the **float** class for floating-point values, and the **str** class for character strings. A more thorough presentation of object-orientation is the focus of Chapter 2.

1.2.1 Identifiers, Objects, and the Assignment Statement

The most important of all Python commands is an ***assignment statement***, such as

```
temperature = 98.6
```

This command establishes temperature as an ***identifier*** (also known as a ***name***), and then associates it with the ***object*** expressed on the right-hand side of the equal sign, in this case a floating-point object with value 98.6. We portray the outcome of this assignment in Figure 1.1.

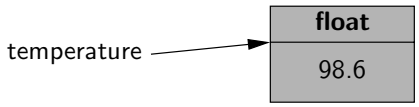


Figure 1.1: The identifier temperature references an instance of the float class having value 98.6.

Identifiers

Identifiers in Python are ***case-sensitive***, so temperature and Temperature are distinct names. Identifiers can be composed of almost any combination of letters, numerals, and underscore characters (or more general Unicode characters). The primary restrictions are that an identifier cannot begin with a numeral (thus 9lives is an illegal name), and that there are 33 specially reserved words that cannot be used as identifiers, as shown in Table 1.1.

Reserved Words								
False	as	continue	else	from	in	not	return	yield
None	assert	def	except	global	is	or	try	
True	break	del	finally	if	lambda	pass	while	
and	class	elif	for	import	nonlocal	raise	with	

Table 1.1: A listing of the reserved words in Python. These names cannot be used as identifiers.

For readers familiar with other programming languages, the semantics of a Python identifier is most similar to a reference variable in Java or a pointer variable in C++. Each identifier is implicitly associated with the ***memory address*** of the object to which it refers. A Python identifier may be assigned to a special object named None, serving a similar purpose to a null reference in Java or C++.

Unlike Java and C++, Python is a ***dynamically typed*** language, as there is no advance declaration associating an identifier with a particular data type. An identifier can be associated with any type of object, and it can later be reassigned to another object of the same (or different) type. Although an identifier has no declared type, the object to which it refers has a definite type. In our first example, the characters 98.6 are recognized as a floating-point literal, and thus the identifier temperature is associated with an instance of the float class having that value.

A programmer can establish an ***alias*** by assigning a second identifier to an existing object. Continuing with our earlier example, Figure 1.2 portrays the result of a subsequent assignment, original = temperature.

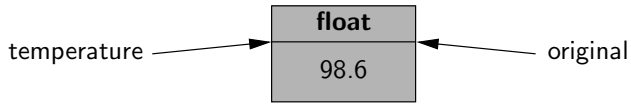


Figure 1.2: Identifiers temperature and original are aliases for the same object.

Once an alias has been established, either name can be used to access the underlying object. If that object supports behaviors that affect its state, changes enacted through one alias will be apparent when using the other alias (because they refer to the same object). However, if one of the *names* is reassigned to a new value using a subsequent assignment statement, that does not affect the aliased object, rather it breaks the alias. Continuing with our concrete example, we consider the command:

```
temperature = temperature + 5.0
```

The execution of this command begins with the evaluation of the expression on the right-hand side of the = operator. That expression, temperature + 5.0, is evaluated based on the *existing* binding of the name temperature, and so the result has value 103.6, that is, 98.6 + 5.0. That result is stored as a new floating-point instance, and only then is the name on the left-hand side of the assignment statement, temperature, (re)assigned to the result. The subsequent configuration is diagrammed in Figure 1.3. Of particular note, this last command had no effect on the value of the existing float instance that identifier original continues to reference.

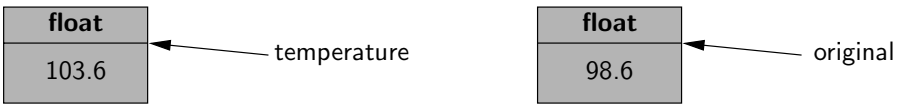


Figure 1.3: The temperature identifier has been assigned to a new value, while original continues to refer to the previously existing value.

1.2.2 Creating and Using Objects

Instantiation

The process of creating a new instance of a class is known as ***instantiation***. In general, the syntax for instantiating an object is to invoke the ***constructor*** of a class. For example, if there were a class named Widget, we could create an instance of that class using a syntax such as `w = Widget()`, assuming that the constructor does not require any parameters. If the constructor does require parameters, we might use a syntax such as `Widget(a, b, c)` to construct a new instance.

Many of Python's built-in classes (discussed in Section 1.2.3) support what is known as a ***literal*** form for designating new instances. For example, the command `temperature = 98.6` results in the creation of a new instance of the **float** class; the term `98.6` in that expression is a literal form. We discuss further cases of Python literals in the coming section.

From a programmer's perspective, yet another way to indirectly create a new instance of a class is to call a function that creates and returns such an instance. For example, Python has a built-in function named `sorted` (see Section 1.5.2) that takes a sequence of comparable elements as a parameter and returns a new instance of the **list** class containing those elements in sorted order.

Calling Methods

Python supports traditional functions (see Section 1.5) that are invoked with a syntax such as `sorted(data)`, in which case `data` is a parameter sent to the function. Python's classes may also define one or more ***methods*** (also known as ***member functions***), which are invoked on a specific instance of a class using the dot (".") operator. For example, Python's list class has a method named `sort` that can be invoked with a syntax such as `data.sort()`. This particular method rearranges the contents of the list so that they are sorted.

The expression to the left of the dot identifies the object upon which the method is invoked. Often, this will be an identifier (e.g., `data`), but we can use the dot operator to invoke a method upon the immediate result of some other operation. For example, if `response` identifies a string instance (we will discuss strings later in this section), the syntax `response.lower().startswith('y')` first evaluates the method call, `response.lower()`, which itself returns a new string instance, and then the `startswith('y')` method is called on that intermediate string.

When using a method of a class, it is important to understand its behavior. Some methods return information about the state of an object, but do not change that state. These are known as ***accessors***. Other methods, such as the `sort` method of the list class, do change the state of an object. These methods are known as ***mutators*** or ***update methods***.

1.2.3 Python's Built-In Classes

Table 1.2 provides a summary of commonly used, built-in classes in Python; we take particular note of which classes are mutable and which are immutable. A class is ***immutable*** if each object of that class has a fixed value upon instantiation that cannot subsequently be changed. For example, the float class is immutable. Once an instance has been created, its value cannot be changed (although an identifier referencing that object can be reassigned to a different value).

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Table 1.2: Commonly used built-in classes for Python

In this section, we provide an introduction to these classes, discussing their purpose and presenting several means for creating instances of the classes. Literal forms (such as 98.6) exist for most of the built-in classes, and all of the classes support a traditional constructor form that creates instances that are based upon one or more existing values. Operators supported by these classes are described in Section 1.3. More detailed information about these classes can be found in later chapters as follows: lists and tuples (Chapter 5); strings (Chapters 5 and 13, and Appendix A); sets and dictionaries (Chapter 10).

The bool Class

The **bool** class is used to manipulate logical (Boolean) values, and the only two instances of that class are expressed as the literals True and False. The default constructor, bool( ), returns False, but there is no reason to use that syntax rather than the more direct literal form. Python allows the creation of a Boolean value from a nonboolean type using the syntax bool(foo) for value foo. The interpretation depends upon the type of the parameter. Numbers evaluate to False if zero, and True if nonzero. Sequences and other container types, such as strings and lists, evaluate to False if empty and True if nonempty. An important application of this interpretation is the use of a nonboolean value as a condition in a control structure.

The int Class

The **int** and **float** classes are the primary numeric types in Python. The **int** class is designed to represent integer values with arbitrary magnitude. Unlike Java and C++, which support different integral types with different precisions (e.g., int, short, long), Python automatically chooses the internal representation for an integer based upon the magnitude of its value. Typical literals for integers include 0, 137, and −23. In some contexts, it is convenient to express an integral value using binary, octal, or hexadecimal. That can be done by using a prefix of the number 0 and then a character to describe the base. Example of such literals are respectively 0b1011, 0o52, and 0x7f.

The integer constructor, int( ), returns value 0 by default. But this constructor can be used to construct an integer value based upon an existing value of another type. For example, if f represents a floating-point value, the syntax int(f) produces the *truncated* value of f. For example, both int(3.14) and int(3.99) produce the value 3, while int(−3.9) produces the value −3. The constructor can also be used to parse a string that is presumed to represent an integral value (such as one entered by a user). If s represents a string, then int(s) produces the integral value that string represents. For example, the expression int('137') produces the integer value 137. If an invalid string is given as a parameter, as in int('hello'), a ValueError is raised (see Section 1.7 for discussion of Python's exceptions). By default, the string must use base 10. If conversion from a different base is desired, that base can be indicated as a second, optional, parameter. For example, the expression int('7f', 16) evaluates to the integer 127.

The float Class

The **float** class is the sole floating-point type in Python, using a fixed-precision representation. Its precision is more akin to a double in Java or C++, rather than those languages' float type. We have already discussed a typical literal form, 98.6. We note that the floating-point equivalent of an integral number can be expressed directly as 2.0. Technically, the trailing zero is optional, so some programmers might use the expression 2. to designate this floating-point literal. One other form of literal for floating-point values uses scientific notation. For example, the literal 6.022e23 represents the mathematical value 6.022×10^{23}.

The constructor form of float( ) returns 0.0. When given a parameter, the constructor attempts to return the equivalent floating-point value. For example, the call float(2) returns the floating-point value 2.0. If the parameter to the constructor is a string, as with float('3.14'), it attempts to parse that string as a floating-point value, raising a ValueError as an exception.

Sequence Types: The list, tuple, and str Classes

The **list**, **tuple**, and **str** classes are ***sequence*** types in Python, representing a collection of values in which the order is significant. The list class is the most general, representing a sequence of arbitrary objects (akin to an "array" in other languages). The tuple class is an ***immutable*** version of the list class, benefiting from a streamlined internal representation. The str class is specially designed for representing an immutable sequence of text characters. We note that Python does not have a separate class for characters; they are just strings with length one.

The list Class

A **list** instance stores a sequence of objects. A list is a ***referential*** structure, as it technically stores a sequence of ***references*** to its elements (see Figure 1.4). Elements of a list may be arbitrary objects (including the None object). Lists are ***array-based*** sequences and are ***zero-indexed***, thus a list of length n has elements indexed from 0 to $n-1$ inclusive. Lists are perhaps the most used container type in Python and they will be extremely central to our study of data structures and algorithms. They have many valuable behaviors, including the ability to dynamically expand and contract their capacities as needed. In this chapter, we will discuss only the most basic properties of lists. We revisit the inner working of all of Python's sequence types as the focus of Chapter 5.

Python uses the characters [ ] as delimiters for a list literal, with [ ] itself being an empty list. As another example, `['red', 'green', 'blue']` is a list containing three string instances. The contents of a list literal need not be expressed as literals; if identifiers a and b have been established, then syntax [a, b] is legitimate.

The list( ) constructor produces an empty list by default. However, the constructor will accept any parameter that is of an ***iterable*** type. We will discuss iteration further in Section 1.8, but examples of iterable types include all of the standard container types (e.g., strings, list, tuples, sets, dictionaries). For example, the syntax list(`'hello'`) produces a list of individual characters, `['h', 'e', 'l', 'l', 'o']`. Because an existing list is itself iterable, the syntax backup = list(data) can be used to construct a new list instance referencing the same contents as the original.

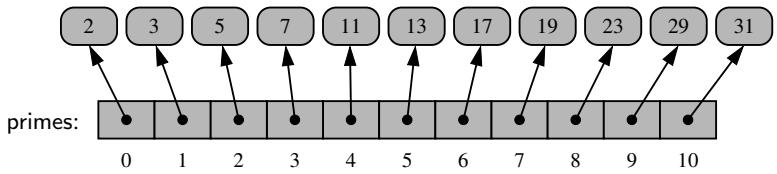


Figure 1.4: Python's internal representation of a list of integers, instantiated as prime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]. The implicit indices of the elements are shown below each entry.

The tuple Class

The **tuple** class provides an immutable version of a sequence, and therefore its instances have an internal representation that may be more streamlined than that of a list. While Python uses the [ ] characters to delimit a list, parentheses delimit a tuple, with ( ) being an empty tuple. There is one important subtlety. To express a tuple of length one as a literal, a comma must be placed after the element, but within the parentheses. For example, (17,) is a one-element tuple. The reason for this requirement is that, without the trailing comma, the expression (17) is viewed as a simple parenthesized numeric expression.

The str Class

Python's **str** class is specifically designed to efficiently represent an immutable sequence of characters, based upon the Unicode international character set. Strings have a more compact internal representation than the referential lists and tuples, as portrayed in Figure 1.5.

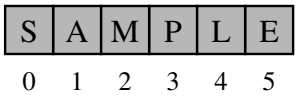


Figure 1.5: A Python string, which is an indexed sequence of characters.

String literals can be enclosed in single quotes, as in `'hello'`, or double quotes, as in `"hello"`. This choice is convenient, especially when using another of the quotation characters as an actual character in the sequence, as in `"Don't worry"`. Alternatively, the quote delimiter can be designated using a backslash as a so-called ***escape character***, as in `'Don\'t worry'`. Because the backslash has this purpose, the backslash must itself be escaped to occur as a natural character of the string literal, as in `'C:\\Python\\'`, for a string that would be displayed as `C:\Python\`. Other commonly escaped characters are `\n` for newline and `\t` for tab. Unicode characters can be included, such as `'20\u20AC'` for the string 20€.

Python also supports using the delimiter `'''` or `"""` to begin and end a string literal. The advantage of such triple-quoted strings is that newline characters can be embedded naturally (rather than escaped as `\n`). This can greatly improve the readability of long, multiline strings in source code. For example, at the beginning of Code Fragment 1.1, rather than use separate print statements for each line of introductory output, we can use a single print statement, as follows:

```
print("""Welcome to the GPA calculator.
Please enter all your letter grades, one per line.
Enter a blank line to designate the end.""")
```

The set and frozenset Classes

Python's **set** class represents the mathematical notion of a set, namely a collection of elements, without duplicates, and without an inherent order to those elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a ***hash table*** (which will be the primary topic of Chapter 10). However, there are two important restrictions due to the algorithmic underpinnings. The first is that the set does not maintain the elements in any particular order. The second is that only instances of ***immutable*** types can be added to a Python set. Therefore, objects such as integers, floating-point numbers, and character strings are eligible to be elements of a set. It is possible to maintain a set of tuples, but not a set of lists or a set of sets, as lists and sets are mutable. The **frozenset** class is an immutable form of the set type, so it is legal to have a set of frozensets.

Python uses curly braces { and } as delimiters for a set, for example, as {17} or {'red', 'green', 'blue'}. The exception to this rule is that { } does not represent an empty set; for historical reasons, it represents an empty dictionary (see next paragraph). Instead, the constructor syntax set( ) produces an empty set. If an iterable parameter is sent to the constructor, then the set of distinct elements is produced. For example, set('hello') produces {'h', 'e', 'l', 'o'}.

The dict Class

Python's **dict** class represents a ***dictionary***, or ***mapping***, from a set of distinct ***keys*** to associated ***values***. For example, a dictionary might map from unique student ID numbers, to larger student records (such as the student's name, address, and course grades). Python implements a dict using an almost identical approach to that of a set, but with storage of the associated values.

A dictionary literal also uses curly braces, and because dictionaries were introduced in Python prior to sets, the literal form { } produces an empty dictionary. A nonempty dictionary is expressed using a comma-separated series of key:value pairs. For example, the dictionary {'ga' : 'Irish', 'de' : 'German'} maps 'ga' to 'Irish' and 'de' to 'German'.

The constructor for the dict class accepts an existing mapping as a parameter, in which case it creates a new dictionary with identical associations as the existing one. Alternatively, the constructor accepts a sequence of key-value pairs as a parameter, as in dict(pairs) with pairs = [('ga', 'Irish'), ('de', 'German')].

1.3 Expressions, Operators, and Precedence

In the previous section, we demonstrated how names can be used to identify existing objects, and how literals and constructors can be used to create instances of built-in classes. Existing values can be combined into larger syntactic ***expressions*** using a variety of special symbols and keywords known as ***operators***. The semantics of an operator depends upon the type of its operands. For example, when a and b are numbers, the syntax a + b indicates addition, while if a and b are strings, the operator indicates concatenation. In this section, we describe Python's operators in various contexts of the built-in types.

We continue, in Section 1.3.1, by discussing ***compound expressions***, such as a + b * c, which rely on the evaluation of two or more operations. The order in which the operations of a compound expression are evaluated can affect the overall value of the expression. For this reason, Python defines a specific order of precedence for evaluating operators, and it allows a programmer to override this order by using explicit parentheses to group subexpressions.

Logical Operators

Python supports the following keyword operators for Boolean values:

not	unary negation
and	conditional and
or	conditional or

The **and** and **or** operators ***short-circuit***, in that they do not evaluate the second operand if the result can be determined based on the value of the first operand. This feature is useful when constructing Boolean expressions in which we first test that a certain condition holds (such as a reference not being **None**), and then test a condition that could have otherwise generated an error condition had the prior test not succeeded.

Equality Operators

Python supports the following operators to test two notions of equality:

is	same identity
is not	different identity
==	equivalent
!=	not equivalent

The expression a **is** b evaluates to **True**, precisely when identifiers a and b are aliases for the *same* object. The expression a == b tests a more general notion of equivalence. If identifiers a and b refer to the same object, then a == b should also evaluate to **True**. Yet a == b also evaluates to **True** when the identifiers refer to

different objects that happen to have values that are deemed equivalent. The precise notion of equivalence depends on the data type. For example, two strings are considered equivalent if they match character for character. Two sets are equivalent if they have the same contents, irrespective of order. In most programming situations, the equivalence tests == and != are the appropriate operators; use of **is** and **is not** should be reserved for situations in which it is necessary to detect true aliasing.

Comparison Operators

Data types may define a natural order via the following operators:

$<$	less than
$<=$	less than or equal to
$>$	greater than
$>=$	greater than or equal to

These operators have expected behavior for numeric types, and are defined lexicographically, and case-sensitively, for strings. An exception is raised if operands have incomparable types, as with `5 < 'hello'`.

Arithmetic Operators

Python supports the following arithmetic operators:

$+$	addition
$-$	subtraction
$*$	multiplication
$/$	true division
$//$	integer division
%	the modulo operator

The use of addition, subtraction, and multiplication is straightforward, noting that if both operands have type int, then the result is an int as well; if one or both operands have type float, the result will be a float.

Python takes more care in its treatment of division. We first consider the case in which both operands have type int, for example, the quantity 27 divided by 4. In mathematical notation, $27 \div 4 = 6\frac{3}{4} = 6.75$. In Python, the / operator designates ***true division***, returning the floating-point result of the computation. Thus, `27 / 4` results in the float value 6.75. Python supports the pair of operators // and % to perform the integral calculations, with expression `27 // 4` evaluating to int value 6 (the mathematical *floor* of the quotient), and expression `27 % 4` evaluating to int value 3, the remainder of the integer division. We note that languages such as C, C++, and Java do not support the // operator; instead, the / operator returns the truncated quotient when both operands have integral type, and the result of true division when at least one operand has a floating-point type.

Python carefully extends the semantics of // and % to cases where one or both operands are negative. For the sake of notation, let us assume that variables n and m represent respectively the ***dividend*** and ***divisor*** of a quotient $\frac{n}{m}$, and that q = n // m and r = n % m. Python guarantees that q * m + r will equal n. We already saw an example of this identity with positive operands, as $6*4+3=27$. When the divisor m is positive, Python further guarantees that $0 \leq r < m$. As a consequence, we find that -27 // 4 evaluates to -7 and -27 % 4 evaluates to 1, as $(-7)*4+1=-27$. When the divisor is negative, Python guarantees that $m < r \leq 0$. As an example, 27 // -4 is -7 and 27 % -4 is -1, satisfying the identity $27=(-7)*(-4)+(-1)$.

The conventions for the // and % operators are even extended to floating-point operands, with the expression q = n // m being the integral floor of the quotient, and r = n % m being the "remainder" to ensure that q * m + r equals n. For example, 8.2 // 3.14 evaluates to 2.0 and 8.2 % 3.14 evaluates to 1.92, as $2.0*3.14+1.92=8.2$.

Bitwise Operators

Python provides the following bitwise operators for integers:

~	bitwise complement (prefix unary operator)
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
<<	shift bits left, filling in with zeros
>>	shift bits right, filling in with sign bit

Sequence Operators

Each of Python's built-in sequence types (**str**, **tuple**, and **list**) support the following operator syntaxes:

s[j]	element at index j
s[start:stop]	slice including indices [start,stop)
s[start:stop:step]	slice including indices start, start + step, start + 2*step, ..., up to but not equalling or stop
s + t	concatenation of sequences
k * s	shorthand for s + s + s + ... (k times)
val **in** s	containment check
val **not in** s	non-containment check

Python relies on ***zero-indexing*** of sequences, thus a sequence of length n has elements indexed from 0 to $n-1$ inclusive. Python also supports the use of ***negative indices***, which denote a distance from the end of the sequence; index -1 denotes the last element, index -2 the second to last, and so on. Python uses a ***slicing***

notation to describe subsequences of a sequence. Slices are described as half-open intervals, with a start index that is included, and a stop index that is excluded. For example, the syntax `data[3:8]` denotes a subsequence including the five indices: $3, 4, 5, 6, 7$. An optional "step" value, possibly negative, can be indicated as a third parameter of the slice. If a start index or stop index is omitted in the slicing notation, it is presumed to designate the respective extreme of the original sequence.

Because lists are mutable, the syntax `s[j] = val` can be used to replace an element at a given index. Lists also support a syntax, **del** `s[j]`, that removes the designated element from the list. Slice notation can also be used to replace or delete a sublist.

The notation `val` **in** `s` can be used for any of the sequences to see if there is an element equivalent to `val` in the sequence. For strings, this syntax can be used to check for a single character or for a larger substring, as with `'amp'` **in** `'example'`.

All sequences define comparison operations based on ***lexicographic order***, performing an element by element comparison until the first difference is found. For example, [5, 6, 9] < [5, 7] because of the entries at index 1. Therefore, the following operations are supported by sequence types:

`s == t`	equivalent (element by element)
`s != t`	not equivalent
`s < t`	lexicographically less than
`s <= t`	lexicographically less than or equal to
`s > t`	lexicographically greater than
`s >= t`	lexicographically greater than or equal to

Operators for Sets and Dictionaries

Sets and frozensets support the following operators:

`key` **in** `s`	containment check
`key` **not in** `s`	non-containment check
`s1 == s2`	s1 is equivalent to s2
`s1 != s2`	s1 is not equivalent to s2
`s1 <= s2`	s1 is subset of s2
`s1 < s2`	s1 is proper subset of s2
`s1 >= s2`	s1 is superset of s2
`s1 > s2`	s1 is proper superset of s2
`s1 | s2`	the union of s1 and s2
`s1 & s2`	the intersection of s1 and s2
`s1 - s2`	the set of elements in s1 but not s2
`s1 ^ s2`	the set of elements in precisely one of s1 or s2

Note well that sets do not guarantee a particular order of their elements, so the comparison operators, such as <, are not lexicographic; rather, they are based on the mathematical notion of a subset. As a result, the comparison operators define

a partial order, but not a total order, as disjoint sets are neither "less than," "equal to," or "greater than" each other. Sets also support many fundamental behaviors through named methods (e.g., add, remove); we will explore their functionality more fully in Chapter 10.

Dictionaries, like sets, do not maintain a well-defined order on their elements. Furthermore, the concept of a subset is not typically meaningful for dictionaries, so the dict class does not support operators such as $<$. Dictionaries support the notion of equivalence, with d1 $==$ d2 if the two dictionaries contain the same set of key-value pairs. The most widely used behavior of dictionaries is accessing a value associated with a particular key k with the indexing syntax, d[k]. The supported operators are as follows:

d[key]	value associated with given key
d[key] = value	set (or reset) the value associated with given key
del d[key]	remove key and its associated value from dictionary
key **in** d	containment check
key **not in** d	non-containment check
d1 == d2	d1 is equivalent to d2
d1 != d2	d1 is not equivalent to d2

Dictionaries also support many useful behaviors through named methods, which we explore more fully in Chapter 10.

Extended Assignment Operators

Python supports an extended assignment operator for most binary operators, for example, allowing a syntax such as count += 5. By default, this is a shorthand for the more verbose count = count + 5. For an immutable type, such as a number or a string, one should not presume that this syntax changes the value of the existing object, but instead that it will reassign the identifier to a newly constructed value. (See discussion of Figure 1.3.) However, it is possible for a type to redefine such semantics to mutate the object, as the list class does for the += operator.

```
alpha = [1, 2, 3]
beta = alpha                # an alias for alpha
beta += [4, 5]              # extends the original list with two more elements
beta = beta + [6, 7]        # reassigns beta to a new list [1, 2, 3, 4, 5, 6, 7]
print(alpha)                # will be [1, 2, 3, 4, 5]
```

This example demonstrates the subtle difference between the list semantics for the syntax beta += foo versus beta = beta + foo.

1.3.1 Compound Expressions and Operator Precedence

Programming languages must have clear rules for the order in which compound expressions, such as $5 + 2 * 3$, are evaluated. The formal order of precedence for operators in Python is given in Table 1.3. Operators in a category with higher precedence will be evaluated before those with lower precedence, unless the expression is otherwise parenthesized. Therefore, we see that Python gives precedence to multiplication over addition, and therefore evaluates the expression $5 + 2 * 3$ as $5 + (2 * 3)$, with value 11, but the parenthesized expression $(5 + 2) * 3$ evaluates to value 21. Operators within a category are typically evaluated from left to right, thus $5 - 2 + 3$ has value 6. Exceptions to this rule include that unary operators and exponentiation are evaluated from right to left.

Python allows a ***chained assignment***, such as `x = y = 0`, to assign multiple identifiers to the rightmost value. Python also allows the ***chaining*** of comparison operators. For example, the expression `1 <= x + y <= 10` is evaluated as the compound `(1 <= x + y)` **and** `(x + y <= 10)`, but without computing the intermediate value `x + y` twice.

Operator Precedence		
	Type	**Symbols**
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) expr[...]
3	exponentiation	**
4	unary operators	+expr, −expr, ~expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, −
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	comparisons containment	**is**, **is not**, ==, !=, <, <=, >, >= **in**, **not in**
12	logical-not	**not** expr
13	logical-and	**and**
14	logical-or	**or**
15	conditional	val1 **if** cond **else** val2
16	assignments	=, +=, −=, *=, etc.

Table 1.3: Operator precedence in Python, with categories ordered from highest precedence to lowest precedence. When stated, we use expr to denote a literal, identifier, or result of a previously evaluated expression. All operators without explicit mention of expr are binary operators, with syntax expr1 operator expr2.

1.4 Control Flow

In this section, we review Python's most fundamental control structures: conditional statements and loops. Common to all control structures is the syntax used in Python for defining blocks of code. The colon character is used to delimit the beginning of a block of code that acts as a body for a control structure. If the body can be stated as a single executable statement, it can technically placed on the same line, to the right of the colon. However, a body is more typically typeset as an ***indented block*** starting on the line following the colon. Python relies on the indentation level to designate the extent of that block of code, or any nested blocks of code within. The same principles will be applied when designating the body of a function (see Section 1.5), and the body of a class (see Section 2.3).

1.4.1 Conditionals

Conditional constructs (also known as **if** statements) provide a way to execute a chosen block of code based on the run-time evaluation of one or more Boolean expressions. In Python, the most general form of a conditional is written as follows:

```
if first_condition:
  first_body
elif second_condition:
  second_body
elif third_condition:
  third_body
else:
  fourth_body
```

Each condition is a Boolean expression, and each body contains one or more commands that are to be executed conditionally. If the first condition succeeds, the first body will be executed; no other conditions or bodies are evaluated in that case. If the first condition fails, then the process continues in similar manner with the evaluation of the second condition. The execution of this overall construct will cause precisely one of the bodies to be executed. There may be any number of **elif** clauses (including zero), and the final **else** clause is optional. As described on page 7, nonboolean types may be evaluated as Booleans with intuitive meanings. For example, if response is a string that was entered by a user, and we want to condition a behavior on this being a nonempty string, we may write

```
if response:
```

as a shorthand for the equivalent,

```
if response != '':
```

As a simple example, a robot controller might have the following logic:

```
if door_is_closed:
  open_door( )
advance( )
```

Notice that the final command, advance( ), is not indented and therefore not part of the conditional body. It will be executed unconditionally (although after opening a closed door).

We may nest one control structure within another, relying on indentation to make clear the extent of the various bodies. Revisiting our robot example, here is a more complex control that accounts for unlocking a closed door.

```
if door_is_closed:
  if door_is_locked:
    unlock_door( )
  open_door( )
advance( )
```

The logic expressed by this example can be diagrammed as a traditional ***flowchart***, as portrayed in Figure 1.6.

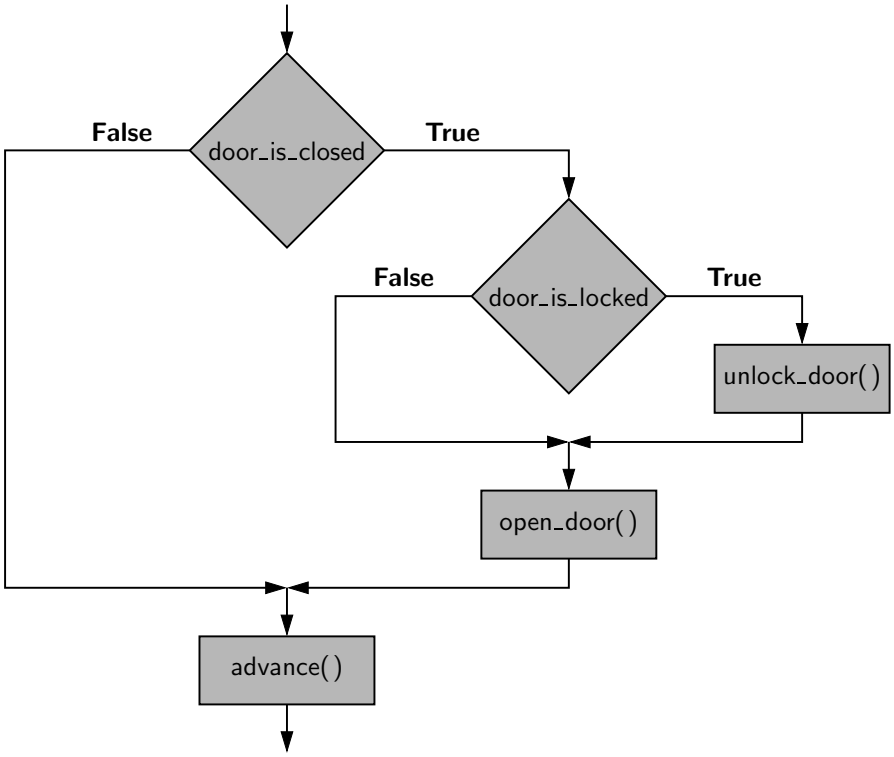


Figure 1.6: A flowchart describing the logic of nested conditional statements.

1.4.2 Loops

Python offers two distinct looping constructs. A **while** loop allows general repetition based upon the repeated testing of a Boolean condition. A **for** loop provides convenient iteration of values from a defined series (such as characters of a string, elements of a list, or numbers within a given range). We discuss both forms in this section.

While Loops

The syntax for a **while** loop in Python is as follows:

```
while condition:
  body
```

As with an **if** statement, *condition* can be an arbitrary Boolean expression, and *body* can be an arbitrary block of code (including nested control structures). The execution of a while loop begins with a test of the Boolean condition. If that condition evaluates to True, the body of the loop is performed. After each execution of the body, the loop condition is retested, and if it evaluates to True, another iteration of the body is performed. When the conditional test evaluates to False (assuming it ever does), the loop is exited and the flow of control continues just beyond the body of the loop.

As an example, here is a loop that advances an index through a sequence of characters until finding an entry with value `'X'` or reaching the end of the sequence.

```
j = 0
while j < len(data) and data[j] != 'X':
  j += 1
```

The len function, which we will introduce in Section 1.5.2, returns the length of a sequence such as a list or string. The correctness of this loop relies on the short-circuiting behavior of the **and** operator, as described on page 12. We intentionally test $j < \text{len}(data)$ to ensure that j is a valid index, prior to accessing element data[j]. Had we written that compound condition with the opposite order, the evaluation of data[j] would eventually raise an IndexError when `'X'` is not found. (See Section 1.7 for discussion of exceptions.)

As written, when this loop terminates, variable j's value will be the index of the leftmost occurrence of `'X'`, if found, or otherwise the length of the sequence (which is recognizable as an invalid index to indicate failure of the search). It is worth noting that this code behaves correctly, even in the special case when the list is empty, as the condition $j < \text{len}(data)$ will initially fail and the body of the loop will never be executed.

For Loops

Python's **for**-loop syntax is a more convenient alternative to a while loop when iterating through a series of elements. The for-loop syntax can be used on any type of ***iterable*** structure, such as a list, tuple str, set, dict, or file (we will discuss iterators more formally in Section 1.8). Its general syntax appears as follows.

```
for element in iterable:
  body                          # body may refer to 'element' as an identifier
```

For readers familiar with Java, the semantics of Python's for loop is similar to the "for each" loop style introduced in Java 1.5.

As an instructive example of such a loop, we consider the task of computing the sum of a list of numbers. (Admittedly, Python has a built-in function, sum, for this purpose.) We perform the calculation with a for loop as follows, assuming that data identifies the list:

```
total = 0
for val in data:
  total += val                  # note use of the loop variable, val
```

The loop body executes once for each element of the data sequence, with the identifier, val, from the for-loop syntax assigned at the beginning of each pass to a respective element. It is worth noting that val is treated as a standard identifier. If the element of the original data happens to be mutable, the val identifier can be used to invoke its methods. But a reassignment of identifier val to a new value has no affect on the original data, nor on the next iteration of the loop.

As a second classic example, we consider the task of finding the maximum value in a list of elements (again, admitting that Python's built-in max function already provides this support). If we can assume that the list, data, has at least one element, we could implement this task as follows:

```
biggest = data[0]               # as we assume nonempty list
for val in data:
  if val > biggest:
    biggest = val
```

Although we could accomplish both of the above tasks with a while loop, the for-loop syntax had an advantage of simplicity, as there is no need to manage an explicit index into the list nor to author a Boolean loop condition. Furthermore, we can use a for loop in cases for which a while loop does not apply, such as when iterating through a collection, such as a set, that does not support any direct form of indexing.

Index-Based For Loops

The simplicity of a standard for loop over the elements of a list is wonderful; however, one limitation of that form is that we do not know where an element resides within the sequence. In some applications, we need knowledge of the index of an element within the sequence. For example, suppose that we want to know *where* the maximum element in a list resides.

Rather than directly looping over the elements of the list in that case, we prefer to loop over all possible indices of the list. For this purpose, Python provides a built-in class named range that generates integer sequences. (We will discuss generators in Section 1.8.) In simplest form, the syntax range(n) generates the series of n values from 0 to $n-1$. Conveniently, these are precisely the series of valid indices into a sequence of length n. Therefore, a standard Python idiom for looping through the series of indices of a data sequence uses a syntax,

```
for j in range(len(data)):
```

In this case, identifier j is not an element of the data—it is an integer. But the expression data[j] can be used to retrieve the respective element. For example, we can find the *index* of the maximum element of a list as follows:

```
big_index = 0
for j in range(len(data)):
  if data[j] > data[big_index]:
    big_index = j
```

Break and Continue Statements

Python supports a **break** statement that immediately terminate a while or for loop when executed within its body. More formally, if applied within nested control structures, it causes the termination of the most immediately enclosing loop. As a typical example, here is code that determines whether a target value occurs in a data set:

```
found = False
for item in data:
  if item == target:
    found = True
    break
```

Python also supports a **continue** statement that causes the current *iteration* of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

We recommend that the break and continue statements be used sparingly. Yet, there are situations in which these commands can be effectively used to avoid introducing overly complex logical conditions.

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.

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'
```

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
```

Default Parameter Values

Python provides means for functions to support more than one possible calling signature. Such a function is said to be ***polymorphic*** (which is Greek for "many forms"). Most notably, functions can declare one or more default values for parameters, thereby allowing the caller to invoke a function with varying numbers of actual parameters. As an artificial example, if a function is declared with signature

```
def foo(a, b=15, c=27):
```

there are three parameters, the last two of which offer default values. A caller is welcome to send three actual parameters, as in foo(4, 12, 8), in which case the default values are not used. If, on the other hand, the caller only sends one parameter, foo(4), the function will execute with parameters values a=4, b=15, c=27. If a caller sends two parameters, they are assumed to be the first two, with the third being the default. Thus, foo(8, 20) executes with a=8, b=20, c=27. However, it is illegal to define a function with a signature such as bar(a, b=15, c) with b having a default value, yet not the subsequent c; if a default parameter value is present for one parameter, it must be present for all further parameters.

As a more motivating example for the use of a default parameter, we revisit the task of computing a student's GPA (see Code Fragment 1.1). Rather than assume direct input and output with the console, we prefer to design a function that computes and returns a GPA. Our original implementation uses a fixed mapping from each letter grade (such as a B−) to a corresponding point value (such as 2.67). While that point system is somewhat common, it may not agree with the system used by all schools. (For example, some may assign an 'A+' grade a value higher than 4.0.) Therefore, we design a compute_gpa function, given in Code Fragment 1.2, which allows the caller to specify a custom mapping from grades to values, while offering the standard point system as a default.

```
def compute_gpa(grades, points={'A+':4.0, 'A':4.0, 'A-':3.67, 'B+':3.33,
                                'B':3.0, 'B-':2.67,'C+':2.33, 'C':2.0,
                                'C':1.67, 'D+':1.33, 'D':1.0, 'F':0.0}):
  num_courses = 0
  total_points = 0
  for g in grades:
    if g in points:                          # a recognizable grade
      num_courses += 1
      total_points += points[g]
  return total_points / num_courses
```

Code Fragment 1.2: A function that computes a student's GPA with a point value system that can be customized as an optional parameter.

As an additional example of an interesting polymorphic function, we consider Python's support for range. (Technically, this is a constructor for the range class, but for the sake of this discussion, we can treat it as a pure function.) Three calling syntaxes are supported. The one-parameter form, range(n), generates a sequence of integers from 0 up to but not including *n*. A two-parameter form, range(start,stop) generates integers from start up to, but not including, stop. A three-parameter form, range(start, stop, step), generates a similar range as range(start, stop), but with increments of size step rather than 1.

This combination of forms seems to violate the rules for default parameters. In particular, when a single parameter is sent, as in range(n), it serves as the stop value (which is the second parameter); the value of start is effectively 0 in that case. However, this effect can be achieved with some sleight of hand, as follows:

```
def range(start, stop=None, step=1):
  if stop is None:
    stop = start
    start = 0
  ...
```

From a technical perspective, when range(n) is invoked, the actual parameter n will be assigned to formal parameter start. Within the body, if only one parameter is received, the start and stop values are reassigned to provide the desired semantics.

Keyword Parameters

The traditional mechanism for matching the actual parameters sent by a caller, to the formal parameters declared by the function signature is based on the concept of ***positional arguments***. For example, with signature foo(a=10, b=20, c=30), parameters sent by the caller are matched, in the given order, to the formal parameters. An invocation of foo(5) indicates that a=5, while b and c are assigned their default values.

Python supports an alternate mechanism for sending a parameter to a function known as a ***keyword argument***. A keyword argument is specified by explicitly assigning an actual parameter to a formal parameter by name. For example, with the above definition of function foo, a call foo(c=5) will invoke the function with parameters a=10, b=20, c=5.

A function's author can require that certain parameters be sent only through the keyword-argument syntax. We never place such a restriction in our own function definitions, but we will see several important uses of keyword-only parameters in Python's standard libraries. As an example, the built-in max function accepts a keyword parameter, coincidentally named key, that can be used to vary the notion of "maximum" that is used.

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.

Common Built-In Functions	
Calling Syntax	**Description**
abs(x)	Return the absolute value of a number.
all(iterable)	Return True if bool(e) is True for each element e.
any(iterable)	Return True if bool(e) is True for at least one element e.
chr(integer)	Return a one-character string with the given Unicode code point.
divmod(x, y)	Return (x // y, x % y) as tuple, if x and y are integers.
hash(obj)	Return an integer hash value for the object (see Chapter 10).
id(obj)	Return the unique integer serving as an "identity" for the object.
input(prompt)	Return a string from standard input; the prompt is optional.
isinstance(obj, cls)	Determine if obj is an instance of the class (or a subclass).
iter(iterable)	Return a new iterator object for the parameter (see Section 1.8).
len(iterable)	Return the number of elements in the given iteration.
map(f, iter1, iter2, ...)	Return an iterator yielding the result of function calls f(e1, e2, ...) for respective elements e1 $\in$ iter1, e2 $\in$ iter2, ...
max(iterable)	Return the largest element of the given iteration.
max(a, b, c, ...)	Return the largest of the arguments.
min(iterable)	Return the smallest element of the given iteration.
min(a, b, c, ...)	Return the smallest of the arguments.
next(iterator)	Return the next element reported by the iterator (see Section 1.8).
open(filename, mode)	Open a file with the given name and access mode.
ord(char)	Return the Unicode code point of the given character.
pow(x, y)	Return the value x^y (as an integer if x and y are integers); equivalent to x ** y.
pow(x, y, z)	Return the value $(x^y \bmod z)$ as an integer.
print(obj1, obj2, ...)	Print the arguments, with separating spaces and trailing newline.
range(stop)	Construct an iteration of values $0, 1, \ldots,$ stop -1.
range(start, stop)	Construct an iteration of values start, start $+1, \ldots,$ stop -1.
range(start, stop, step)	Construct an iteration of values start, start + step, start + 2*step, ...
reversed(sequence)	Return an iteration of the sequence in reverse.
round(x)	Return the nearest int value (a tie is broken toward the even value).
round(x, k)	Return the value rounded to the nearest 10^{-k} (return-type matches x).
sorted(iterable)	Return a list containing elements of the iterable in sorted order.
sum(iterable)	Return the sum of the elements in the iterable (must be numeric).
type(obj)	Return the class to which the instance obj belongs.

Table 1.4: Commonly used built-in function in Python.

1.6 Simple Input and Output

In this section, we address the basics of input and output in Python, describing standard input and output through the user console, and Python's support for reading and writing text files.

1.6.1 Console Input and Output

The print Function

The built-in function, print, is used to generate standard output to the console. In its simplest form, it prints an arbitrary sequence of arguments, separated by spaces, and followed by a trailing newline character. For example, the command print('maroon', 5) outputs the string 'maroon 5\n'. Note that arguments need not be string instances. A nonstring argument x will be displayed as str(x). Without any arguments, the command print( ) outputs a single newline character.

The print function can be customized through the use of the following keyword parameters (see Section 1.5 for a discussion of keyword parameters):

- By default, the print function inserts a separating space into the output between each pair of arguments. The separator can be customized by providing a desired separating string as a keyword parameter, sep. For example, colon-separated output can be produced as print(a, b, c, sep=':'). The separating string need not be a single character; it can be a longer string, and it can be the empty string, sep='', causing successive arguments to be directly concatenated.
- By default, a trailing newline is output after the final argument. An alternative trailing string can be designated using a keyword parameter, end. Designating the empty string end='' suppresses all trailing characters.
- By default, the print function sends its output to the standard console. However, output can be directed to a file by indicating an output file stream (see Section 1.6.2) using file as a keyword parameter.

The input Function

The primary means for acquiring information from the user console is a built-in function named input. This function displays a prompt, if given as an optional parameter, and then waits until the user enters some sequence of characters followed by the return key. The formal return value of the function is the string of characters that were entered strictly before the return key (i.e., no newline character exists in the returned string).

When reading a numeric value from the user, a programmer must use the input function to get the string of characters, and then use the int or float syntax to construct the numeric value that character string represents. That is, if a call to response = input( ) reports that the user entered the characters, '2013', the syntax int(response) could be used to produce the integer value 2013. It is quite common to combine these operations with a syntax such as

```
year = int(input('In what year were you born? '))
```

if we assume that the user will enter an appropriate response. (In Section 1.7 we discuss error handling in such a situation.)

Because input returns a string as its result, use of that function can be combined with the existing functionality of the string class, as described in Appendix A. For example, if the user enters multiple pieces of information on the same line, it is common to call the split method on the result, as in

```
reply = input('Enter x and y, separated by spaces: ')
pieces = reply.split( )    # returns a list of strings, as separated by spaces
x = float(pieces[0])
y = float(pieces[1])
```

A Sample Program

Here is a simple, but complete, program that demonstrates the use of the input and print functions. The tools for formatting the final output is discussed in Appendix A.

```
age = int(input('Enter your age in years: '))
max_heart_rate = 206.9 - (0.67 * age)    # as per Med Sci Sports Exerc.
target = 0.65 * max_heart_rate
print('Your target fat-burning heart rate is', target)
```

1.6.2 Files

Files are typically accessed in Python beginning with a call to a built-in function, named open, that returns a proxy for interactions with the underlying file. For example, the command, fp = open('sample.txt'), attempts to open a file named sample.txt, returning a proxy that allows read-only access to the text file.

The open function accepts an optional second parameter that determines the access mode. The default mode is 'r' for reading. Other common modes are 'w' for writing to the file (causing any existing file with that name to be overwritten), or 'a' for appending to the end of an existing file. Although we focus on use of text files, it is possible to work with binary files, using access modes such as 'rb' or 'wb'.

When processing a file, the proxy maintains a current position within the file as an offset from the beginning, measured in number of bytes. When opening a file with mode `'r'` or `'w'`, the position is initially 0; if opened in append mode, `'a'`, the position is initially at the end of the file. The syntax fp.close( ) closes the file associated with proxy fp, ensuring that any written contents are saved. A summary of methods for reading and writing a file is given in Table 1.5

Calling Syntax	Description
fp.read( )	Return the (remaining) contents of a readable file as a string.
fp.read(k)	Return the next k bytes of a readable file as a string.
fp.readline( )	Return (remainder of) the current line of a readable file as a string.
fp.readlines( )	Return all (remaining) lines of a readable file as a list of strings.
for line **in** fp:	Iterate all (remaining) lines of a readable file.
fp.seek(k)	Change the current position to be at the k^{th} byte of the file.
fp.tell( )	Return the current position, measured as byte-offset from the start.
fp.write(string)	Write given string at current position of the writable file.
fp.writelines(seq)	Write each of the strings of the given sequence at the current position of the writable file. This command does *not* insert any newlines, beyond those that are embedded in the strings.
print(..., file=fp)	Redirect output of print function to the file.

Table 1.5: Behaviors for interacting with a text file via a file proxy (named fp).

Reading from a File

The most basic command for reading via a proxy is the read method. When invoked on file proxy fp, as fp.read(k), the command returns a string representing the next k bytes of the file, starting at the current position. Without a parameter, the syntax fp.read( ) returns the remaining contents of the file in entirety. For convenience, files can be read a line at a time, using the readline method to read one line, or the readlines method to return a list of all remaining lines. Files also support the for-loop syntax, with iteration being line by line (e.g., **for** line **in** fp:).

Writing to a File

When a file proxy is writable, for example, if created with access mode `'w'` or `'a'`, text can be written using methods write or writelines. For example, if we define fp = open(`'results.txt'`, `'w'`), the syntax fp.write(`'Hello World.\n'`) writes a single line to the file with the given string. Note well that write does not explicitly add a trailing newline, so desired newline characters must be embedded directly in the string parameter. Recall that the output of the print method can be redirected to a file using a keyword parameter, as described in Section 1.6.

1.7 Exception Handling

Exceptions are unexpected events that occur during the execution of a program. An exception might result from a logical error or an unanticipated situation. In Python, ***exceptions*** (also known as ***errors***) are objects that are ***raised*** (or ***thrown***) by code that encounters an unexpected circumstance. The Python interpreter can also raise an exception should it encounter an unexpected condition, like running out of memory. A raised error may be ***caught*** by a surrounding context that "handles" the exception in an appropriate fashion. If uncaught, an exception causes the interpreter to stop executing the program and to report an appropriate message to the console. In this section, we examine the most common error types in Python, the mechanism for catching and handling errors that have been raised, and the syntax for raising errors from within user-defined blocks of code.

Common Exception Types

Python includes a rich hierarchy of exception classes that designate various categories of errors; Table 1.6 shows many of those classes. The Exception class serves as a base class for most other error types. An instance of the various subclasses encodes details about a problem that has occurred. Several of these errors may be raised in exceptional cases by behaviors introduced in this chapter. For example, use of an undefined identifier in an expression causes a NameError, and errant use of the dot notation, as in foo.bar( ), will generate an AttributeError if object foo does not support a member named bar.

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Table 1.6: Common exception classes in Python

Sending the wrong number, type, or value of parameters to a function is another common cause for an exception. For example, a call to abs('hello') will raise a TypeError because the parameter is not numeric, and a call to abs(3, 5) will raise a TypeError because one parameter is expected. A ValueError is typically raised when the correct number and type of parameters are sent, but a value is illegitimate for the context of the function. For example, the int constructor accepts a string, as with int('137'), but a ValueError is raised if that string does not represent an integer, as with int('3.14') or int('hello').

Python's sequence types (e.g., list, tuple, and str) raise an IndexError when syntax such as data[k] is used with an integer k that is not a valid index for the given sequence (as described in Section 1.2.3). Sets and dictionaries raise a KeyError when an attempt is made to access a nonexistent element.

1.7.1 Raising an Exception

An exception is thrown by executing the **raise** statement, with an appropriate instance of an exception class as an argument that designates the problem. For example, if a function for computing a square root is sent a negative value as a parameter, it can raise an exception with the command:

```
raise ValueError('x cannot be negative')
```

This syntax raises a newly created instance of the ValueError class, with the error message serving as a parameter to the constructor. If this exception is not caught within the body of the function, the execution of the function immediately ceases and the exception is propagated to the calling context (and possibly beyond).

When checking the validity of parameters sent to a function, it is customary to first verify that a parameter is of an appropriate type, and then to verify that it has an appropriate value. For example, the sqrt function in Python's math library performs error-checking that might be implemented as follows:

```
def sqrt(x):
  if not isinstance(x, (int, float)):
    raise TypeError('x must be numeric')
  elif x < 0:
    raise ValueError('x cannot be negative')
  # do the real work here...
```

Checking the type of an object can be performed at run-time using the built-in function, isinstance. In simplest form, isinstance(obj, cls) returns True if object, obj, is an instance of class, cls, or any subclass of that type. In the above example, a more general form is used with a tuple of allowable types indicated with the second parameter. After confirming that the parameter is numeric, the function enforces an expectation that the number be nonnegative, raising a ValueError otherwise.

How much error-checking to perform within a function is a matter of debate. Checking the type and value of each parameter demands additional execution time and, if taken to an extreme, seems counter to the nature of Python. Consider the built-in sum function, which computes a sum of a collection of numbers. An implementation with rigorous error-checking might be written as follows:

```
def sum(values):
  if not isinstance(values, collections.Iterable):
    raise TypeError('parameter must be an iterable type')
  total = 0
  for v in values:
    if not isinstance(v, (int, float)):
      raise TypeError('elements must be numeric')
    total = total+ v
  return total
```

The abstract base class, collections.Iterable, includes all of Python's iterable containers types that guarantee support for the for-loop syntax (e.g., list, tuple, set); we discuss iterables in Section 1.8, and the use of modules, such as collections, in Section 1.11. Within the body of the for loop, each element is verified as numeric before being added to the total. A far more direct and clear implementation of this function can be written as follows:

```
def sum(values):
  total = 0
  for v in values:
    total = total + v
  return total
```

Interestingly, this simple implementation performs exactly like Python's built-in version of the function. Even without the explicit checks, appropriate exceptions are raised naturally by the code. In particular, if values is not an iterable type, the attempt to use the for-loop syntax raises a TypeError reporting that the object is not iterable. In the case when a user sends an iterable type that includes a nonnumerical element, such as sum([3.14, 'oops']), a TypeError is naturally raised by the evaluation of expression total + v. The error message

```
unsupported operand type(s) for +:  'float' and 'str'
```

should be sufficiently informative to the caller. Perhaps slightly less obvious is the error that results from sum(['alpha', 'beta']). It will technically report a failed attempt to add an int and str, due to the initial evaluation of total + 'alpha', when total has been initialized to 0.

In the remainder of this book, we tend to favor the simpler implementations in the interest of clean presentation, performing minimal error-checking in most situations.

1.7.2 Catching an Exception

There are several philosophies regarding how to cope with possible exceptional cases when writing code. For example, if a division x/y is to be computed, there is clear risk that a ZeroDivisionError will be raised when variable y has value 0. In an ideal situation, the logic of the program may dictate that y has a nonzero value, thereby removing the concern for error. However, for more complex code, or in a case where the value of y depends on some external input to the program, there remains some possibility of an error.

One philosophy for managing exceptional cases is to ***"look before you leap."*** The goal is to entirely avoid the possibility of an exception being raised through the use of a proactive conditional test. Revisiting our division example, we might avoid the offending situation by writing:

```
if y != 0:
  ratio = x / y
else:
  ... do something else ...
```

A second philosophy, often embraced by Python programmers, is that ***"it is easier to ask for forgiveness than it is to get permission."*** This quote is attributed to Grace Hopper, an early pioneer in computer science. The sentiment is that we need not spend extra execution time safeguarding against every possible exceptional case, as long as there is a mechanism for coping with a problem after it arises. In Python, this philosophy is implemented using a ***try-except*** control structure. Revising our first example, the division operation can be guarded as follows:

```
try:
  ratio = x / y
except ZeroDivisionError:
  ... do something else ...
```

In this structure, the "try" block is the primary code to be executed. Although it is a single command in this example, it can more generally be a larger block of indented code. Following the try-block are one or more "except" cases, each with an identified error type and an indented block of code that should be executed if the designated error is raised within the try-block.

The relative advantage of using a try-except structure is that the non-exceptional case runs efficiently, without extraneous checks for the exceptional condition. However, handling the exceptional case requires slightly more time when using a try-except structure than with a standard conditional statement. For this reason, the try-except clause is best used when there is reason to believe that the exceptional case is relatively unlikely, or when it is prohibitively expensive to proactively evaluate a condition to avoid the exception.

Exception handling is particularly useful when working with user input, or when reading from or writing to files, because such interactions are inherently less predictable. In Section 1.6.2, we suggest the syntax, fp = open('sample.txt'), for opening a file with read access. That command may raise an IOError for a variety of reasons, such as a non-existent file, or lack of sufficient privilege for opening a file. It is significantly easier to attempt the command and catch the resulting error than it is to accurately predict whether the command will succeed.

We continue by demonstrating a few other forms of the try-except syntax. Exceptions are objects that can be examined when caught. To do so, an identifier must be established with a syntax as follows:

```
try:
  fp = open('sample.txt')
except IOError as e:
  print('Unable to open the file:', e)
```

In this case, the name, e, denotes the instance of the exception that was thrown, and printing it causes a detailed error message to be displayed (e.g., "file not found").

A try-statement may handle more than one type of exception. For example, consider the following command from Section 1.6.1:

```
age = int(input('Enter your age in years: '))
```

This command could fail for a variety of reasons. The call to input will raise an EOFError if the console input fails. If the call to input completes successfully, the int constructor raises a ValueError if the user has not entered characters representing a valid integer. If we want to handle two or more types of errors in the same way, we can use a single except-statement, as in the following example:

```
age = -1                    # an initially invalid choice
while age <= 0:
  try:
    age = int(input('Enter your age in years: '))
    if age <= 0:
      print('Your age must be positive')
  except (ValueError, EOFError):
    print('Invalid response')
```

We use the tuple, (ValueError, EOFError), to designate the types of errors that we wish to catch with the except-clause. In this implementation, we catch either error, print a response, and continue with another pass of the enclosing while loop. We note that when an error is raised within the try-block, the remainder of that body is immediately skipped. In this example, if the exception arises within the call to input, or the subsequent call to the int constructor, the assignment to age never occurs, nor the message about needing a positive value. Because the value of age

will be unchanged, the while loop will continue. If we preferred to have the while loop continue without printing the 'Invalid response' message, we could have written the exception-clause as

```
except (ValueError, EOFError):
  pass
```

The keyword, **pass**, is a statement that does nothing, yet it can serve syntactically as a body of a control structure. In this way, we quietly catch the exception, thereby allowing the surrounding while loop to continue.

In order to provide different responses to different types of errors, we may use two or more except-clauses as part of a try-structure. In our previous example, an EOFError suggests a more insurmountable error than simply an errant value being entered. In that case, we might wish to provide a more specific error message, or perhaps to allow the exception to interrupt the loop and be propagated to a higher context. We could implement such behavior as follows:

```
age = -1                          # an initially invalid choice
while age <= 0:
  try:
    age = int(input('Enter your age in years: '))
    if age <= 0:
      print('Your age must be positive')
  except ValueError:
    print('That is an invalid age specification')
  except EOFError:
    print('There was an unexpected error reading input.')
    raise                         # let's re-raise this exception
```

In this implementation, we have separate except-clauses for the ValueError and EOFError cases. The body of the clause for handling an EOFError relies on another technique in Python. It uses the raise statement without any subsequent argument, to re-raise the same exception that is currently being handled. This allows us to provide our own response to the exception, and then to interrupt the while loop and propagate the exception upward.

In closing, we note two additional features of try-except structures in Python. It is permissible to have a final except-clause without any identified error types, using syntax **except**:, to catch any other exceptions that occurred. However, this technique should be used sparingly, as it is difficult to suggest how to handle an error of an unknown type. A try-statement can have a **finally** clause, with a body of code that will always be executed in the standard or exceptional cases, even when an uncaught or re-raised exception occurs. That block is typically used for critical cleanup work, such as closing an open file.

1.8 Iterators and Generators

In Section 1.4.2, we introduced the for-loop syntax beginning as:

for *element* **in** *iterable*:

and we noted that there are many types of objects in Python that qualify as being iterable. Basic container types, such as list, tuple, and set, qualify as iterable types. Furthermore, a string can produce an iteration of its characters, a dictionary can produce an iteration of its keys, and a file can produce an iteration of its lines. User-defined types may also support iteration. In Python, the mechanism for iteration is based upon the following conventions:

- An ***iterator*** is an object that manages an iteration through a series of values. If variable, i, identifies an iterator object, then each call to the built-in function, next(i), produces a subsequent element from the underlying series, with a StopIteration exception raised to indicate that there are no further elements.
- An ***iterable*** is an object, obj, that produces an *iterator* via the syntax iter(obj).

By these definitions, an instance of a list is an iterable, but not itself an iterator. With data = [1, 2, 4, 8], it is not legal to call next(data). However, an iterator object can be produced with syntax, i = iter(data), and then each subsequent call to next(i) will return an element of that list. The for-loop syntax in Python simply automates this process, creating an iterator for the give iterable, and then repeatedly calling for the next element until catching the StopIteration exception.

More generally, it is possible to create multiple iterators based upon the same iterable object, with each iterator maintaining its own state of progress. However, iterators typically maintain their state with indirect reference back to the original collection of elements. For example, calling iter(data) on a list instance produces an instance of the list_iterator class. That iterator does not store its own copy of the list of elements. Instead, it maintains a current *index* into the original list, representing the next element to be reported. Therefore, if the contents of the original list are modified after the iterator is constructed, but before the iteration is complete, the iterator will be reporting the *updated* contents of the list.

Python also supports functions and classes that produce an implicit iterable series of values, that is, without constructing a data structure to store all of its values at once. For example, the call range(1000000) does *not* return a list of numbers; it returns a range object that is iterable. This object generates the million values one at a time, and only as needed. Such a ***lazy evaluation*** technique has great advantage. In the case of range, it allows a loop of the form, **for** j **in** range(1000000):, to execute without setting aside memory for storing one million values. Also, if such a loop were to be interrupted in some fashion, no time will have been spent computing unused values of the range.

We see lazy evaluation used in many of Python's libraries. For example, the dictionary class supports methods keys( ), values( ), and items( ), which respectively produce a "view" of all keys, values, or (key,value) pairs within a dictionary. None of these methods produces an explicit list of results. Instead, the views that are produced are iterable objects based upon the actual contents of the dictionary. An explicit list of values from such an iteration can be immediately constructed by calling the list class constructor with the iteration as a parameter. For example, the syntax list(range(1000)) produces a list instance with values from 0 to 999, while the syntax list(d.values( )) produces a list that has elements based upon the current values of dictionary d. We can similarly construct a tuple or set instance based upon a given iterable.

Generators

In Section 2.3.4, we will explain how to define a class whose instances serve as iterators. However, the most convenient technique for creating iterators in Python is through the use of ***generators***. A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a **yield** statement is executed to indicate each element of the series. As an example, consider the goal of determining all factors of a positive integer. For example, the number 100 has factors 1, 2, 4, 5, 10, 20, 25, 50, 100. A traditional function might produce and return a list containing all factors, implemented as:

```
def factors(n):                    # traditional function that computes factors
  results = [ ]                    # store factors in a new list
  for k in range(1,n+1):
    if n % k == 0:                 # divides evenly, thus k is a factor
      results.append(k)            # add k to the list of factors
  return results                   # return the entire list
```

In contrast, an implementation of a *generator* for computing those factors could be implemented as follows:

```
def factors(n):                    # generator that computes factors
  for k in range(1,n+1):
    if n % k == 0:                 # divides evenly, thus k is a factor
      yield k                      # yield this factor as next result
```

Notice use of the keyword **yield** rather than **return** to indicate a result. This indicates to Python that we are defining a generator, rather than a traditional function. It is illegal to combine yield and return statements in the same implementation, other than a zero-argument return statement to cause a generator to end its execution. If a programmer writes a loop such as **for** factor **in** factors(100):, an instance of our generator is created. For each iteration of the loop, Python executes our procedure

until a yield statement indicates the next value. At that point, the procedure is temporarily interrupted, only to be resumed when another value is requested. When the flow of control naturally reaches the end of our procedure (or a zero-argument return statement), a StopIteration exception is automatically raised. Although this particular example uses a single yield statement in the source code, a generator can rely on multiple yield statements in different constructs, with the generated series determined by the natural flow of control. For example, we can greatly improve the efficiency of our generator for computing factors of a number, n, by only testing values up to the square root of that number, while reporting the factor $n//k$ that is associated with each k (unless $n//k$ equals k). We might implement such a generator as follows:

```
def factors(n):                # generator that computes factors
  k = 1
  while k * k < n:             # while k < sqrt(n)
    if n % k == 0:
      yield k
      yield n // k
    k += 1
  if k * k == n:               # special case if n is perfect square
    yield k
```

We should note that this generator differs from our first version in that the factors are not generated in strictly increasing order. For example, factors(100) generates the series $1, 100, 2, 50, 4, 25, 5, 20, 10$.

In closing, we wish to emphasize the benefits of lazy evaluation when using a generator rather than a traditional function. The results are only computed if requested, and the entire series need not reside in memory at one time. In fact, a generator can effectively produce an infinite series of values. As an example, the Fibonacci numbers form a classic mathematical sequence, starting with value 0, then value 1, and then each subsequent value being the sum of the two preceding values. Hence, the Fibonacci series begins as: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$. The following generator produces this infinite series.

```
def fibonacci( ):
  a = 0
  b = 1
  while True:                  # keep going...
    yield a                    # report value, a, during this pass
    future = a + b
    a = b                      # this will be next value reported
    b = future                 # and subsequently this
```

1.9 Additional Python Conveniences

In this section, we introduce several features of Python that are particularly convenient for writing clean, concise code. Each of these syntaxes provide functionality that could otherwise be accomplished using functionality that we have introduced earlier in this chapter. However, at times, the new syntax is a more clear and direct expression of the logic.

1.9.1 Conditional Expressions

Python supports a ***conditional expression*** syntax that can replace a simple control structure. The general syntax is an expression of the form:

expr1 **if** *condition* **else** *expr2*

This compound expression evaluates to *expr1* if the condition is true, and otherwise evaluates to *expr2*. For those familiar with Java or C++, this is equivalent to the syntax, *condition* ? *expr1* : *expr2*, in those languages.

As an example, consider the goal of sending the absolute value of a variable, n, to a function (and without relying on the built-in abs function, for the sake of example). Using a traditional control structure, we might accomplish this as follows:

```
if n >= 0:
    param = n
else:
    param = -n
result = foo(param)                # call the function
```

With the conditional expression syntax, we can directly assign a value to variable, param, as follows:

```
param = n if n >= 0 else -n        # pick the appropriate value
result = foo(param)                # call the function
```

In fact, there is no need to assign the compound expression to a variable. A conditional expression can itself serve as a parameter to the function, written as follows:

```
result = foo(n if n >= 0 else -n)
```

Sometimes, the mere shortening of source code is advantageous because it avoids the distraction of a more cumbersome control structure. However, we recommend that a conditional expression be used only when it improves the readability of the source code, and when the first of the two options is the more "natural" case, given its prominence in the syntax. (We prefer to view the alternative value as more exceptional.)

1.9.2 Comprehension Syntax

A very common programming task is to produce one series of values based upon the processing of another series. Often, this task can be accomplished quite simply in Python using what is known as a ***comprehension syntax***. We begin by demonstrating ***list comprehension***, as this was the first form to be supported by Python. Its general form is as follows:

```
[ expression for value in iterable if condition ]
```

We note that both *expression* and *condition* may depend on *value*, and that the if-clause is optional. The evaluation of the comprehension is logically equivalent to the following traditional control structure for computing a resulting list:

```
result = [ ]
for value in iterable:
  if condition:
    result.append(expression)
```

As a concrete example, a list of the squares of the numbers from 1 to n, that is $[1, 4, 9, 16, 25, \ldots, n^2]$, can be created by traditional means as follows:

```
squares = [ ]
for k in range(1, n+1):
  squares.append(k*k)
```

With list comprehension, this logic is expressed as follows:

```
squares = [k*k for k in range(1, n+1)]
```

As a second example, Section 1.8 introduced the goal of producing a list of factors for an integer n. That task is accomplished with the following list comprehension:

```
factors = [k for k in range(1,n+1) if n % k == 0]
```

Python supports similar comprehension syntaxes that respectively produce a set, generator, or dictionary. We compare those syntaxes using our example for producing the squares of numbers.

```
[ k*k for k in range(1, n+1) ]          list comprehension
{ k*k for k in range(1, n+1) }          set comprehension
( k*k for k in range(1, n+1) )          generator comprehension
{ k : k*k for k in range(1, n+1) }      dictionary comprehension
```

The generator syntax is particularly attractive when results do not need to be stored in memory. For example, to compute the sum of the first n squares, the generator syntax, `total = sum(k*k for k in range(1, n+1))`, is preferred to the use of an explicitly instantiated list comprehension as the parameter.

1.9.3 Packing and Unpacking of Sequences

Python provides two additional conveniences involving the treatment of tuples and other sequence types. The first is rather cosmetic. If a series of comma-separated expressions are given in a larger context, they will be treated as a single tuple, even if no enclosing parentheses are provided. For example, the assignment

```
data = 2, 4, 6, 8
```

results in identifier, data, being assigned to the tuple (2, 4, 6, 8). This behavior is called ***automatic packing*** of a tuple. One common use of packing in Python is when returning multiple values from a function. If the body of a function executes the command,

```
return x, y
```

it will be formally returning a single object that is the tuple (x, y).

As a dual to the packing behavior, Python can automatically ***unpack*** a sequence, allowing one to assign a series of individual identifiers to the elements of sequence. As an example, we can write

```
a, b, c, d = range(7, 11)
```

which has the effect of assigning a=7, b=8, c=9, and d=10, as those are the four values in the sequence returned by the call to range. For this syntax, the right-hand side expression can be any *iterable* type, as long as the number of variables on the left-hand side is the same as the number of elements in the iteration.

This technique can be used to unpack tuples returned by a function. For example, the built-in function, divmod(a, b), returns the pair of values (a // b, a % b) associated with an integer division. Although the caller can consider the return value to be a single tuple, it is possible to write

```
quotient, remainder = divmod(a, b)
```

to separately identify the two entries of the returned tuple. This syntax can also be used in the context of a for loop, when iterating over a sequence of iterables, as in

```
for x, y in [ (7, 2), (5, 8), (6, 4) ]:
```

In this example, there will be three iterations of the loop. During the first pass, x=7 and y=2, and so on. This style of loop is quite commonly used to iterate through key-value pairs that are returned by the items( ) method of the dict class, as in:

```
for k, v in mapping.items( ):
```

Simultaneous Assignments

The combination of automatic packing and unpacking forms a technique known as ***simultaneous assignment***, whereby we explicitly assign a series of values to a series of identifiers, using a syntax:

```
x, y, z = 6, 2, 5
```

In effect, the right-hand side of this assignment is automatically packed into a tuple, and then automatically unpacked with its elements assigned to the three identifiers on the left-hand side.

When using a simultaneous assignment, all of the expressions are evaluated on the right-hand side before any of the assignments are made to the left-hand variables. This is significant, as it provides a convenient means for swapping the values associated with two variables:

```
j, k = k, j
```

With this command, j will be assigned to the *old* value of k, and k will be assigned to the *old* value of j. Without simultaneous assignment, a swap typically requires more delicate use of a temporary variable, such as

```
temp = j
j = k
k = temp
```

With the simultaneous assignment, the unnamed tuple representing the packed values on the right-hand side implicitly serves as the temporary variable when performing such a swap.

The use of simultaneous assignments can greatly simplify the presentation of code. As an example, we reconsider the generator on page 41 that produces the Fibonacci series. The original code requires separate initialization of variables a and b to begin the series. Within each pass of the loop, the goal was to reassign a and b, respectively, to the values of b and a+b. At the time, we accomplished this with brief use of a third variable. With simultaneous assignments, that generator can be implemented more directly as follows:

```
def fibonacci( ):
  a, b = 0, 1
  while True:
    yield a
    a, b = b, a+b
```

1.10 Scopes and Namespaces

When computing a sum with the syntax x + y in Python, the names x and y must have been previously associated with objects that serve as values; a NameError will be raised if no such definitions are found. The process of determining the value associated with an identifier is known as ***name resolution***.

Whenever an identifier is assigned to a value, that definition is made with a specific ***scope***. Top-level assignments are typically made in what is known as ***global*** scope. Assignments made within the body of a function typically have scope that is ***local*** to that function call. Therefore, an assignment, x = 5, within a function has no effect on the identifier, x, in the broader scope.

Each distinct scope in Python is represented using an abstraction known as a ***namespace***. A namespace manages all identifiers that are currently defined in a given scope. Figure 1.8 portrays two namespaces, one being that of a caller to our count function from Section 1.5, and the other being the local namespace during the execution of that function.

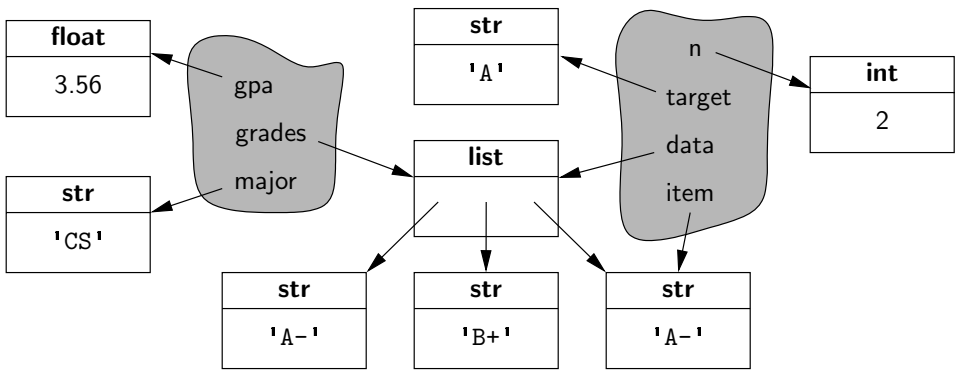


Figure 1.8: A portrayal of the two namespaces associated with a user's call count(grades, 'A'), as defined in Section 1.5. The left namespace is the caller's and the right namespace represents the local scope of the function.

Python implements a namespace with its own dictionary that maps each identifying string (e.g., 'n') to its associated value. Python provides several ways to examine a given namespace. The function, dir, reports the names of the identifiers in a given namespace (i.e., the keys of the dictionary), while the function, vars, returns the full dictionary. By default, calls to dir( ) and vars( ) report on the most locally enclosing namespace in which they are executed.

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.)

1.11 Modules and the Import Statement

We have already introduced many functions (e.g., max) and classes (e.g., list) that are defined within Python's built-in namespace. Depending on the version of Python, there are approximately 130–150 definitions that were deemed significant enough to be included in that built-in namespace.

Beyond the built-in definitions, the standard Python distribution includes perhaps tens of thousands of other values, functions, and classes that are organized in additional libraries, known as ***modules***, that can be ***imported*** from within a program. As an example, we consider the math module. While the built-in namespace includes a few mathematical functions (e.g., abs, min, max, round), many more are relegated to the math module (e.g., sin, cos, sqrt). That module also defines approximate values for the mathematical constants, pi and e.

Python's **import** statement loads definitions from a module into the current namespace. One form of an import statement uses a syntax such as the following:

```
from math import pi, sqrt
```

This command adds both pi and sqrt, as defined in the math module, into the current namespace, allowing direct use of the identifier, pi, or a call of the function, sqrt(2). If there are many definitions from the same module to be imported, an asterisk may be used as a wild card, as in, **from** math **import** $*$, but this form should be used sparingly. The danger is that some of the names defined in the module may conflict with names already in the current namespace (or being imported from another module), and the import causes the new definitions to replace existing ones.

Another approach that can be used to access many definitions from the same module is to import the module itself, using a syntax such as:

```
import math
```

Formally, this adds the identifier, math, to the current namespace, with the module as its value. (Modules are also first-class objects in Python.) Once imported, individual definitions from the module can be accessed using a fully-qualified name, such as math.pi or math.sqrt(2).

Creating a New Module

To create a new module, one simply has to put the relevant definitions in a file named with a `.py` suffix. Those definitions can be imported from any other `.py` file within the same project directory. For example, if we were to put the definition of our count function (see Section 1.5) into a file named `utility.py`, we could import that function using the syntax, **from** utility **import** count.

It is worth noting that top-level commands with the module source code are executed when the module is first imported, almost as if the module were its own script. There is a special construct for embedding commands within the module that will be executed if the module is directly invoked as a script, but not when the module is imported from another script. Such commands should be placed in a body of a conditional statement of the following form,

```
if __name__ == '__main__':
```

Using our hypothetical `utility.py` module as an example, such commands will be executed if the interpreter is started with a command `python utility.py`, but not when the utility module is imported into another context. This approach is often used to embed what are known as ***unit tests*** within the module; we will discuss unit testing further in Section 2.2.4.

1.11.1 Existing Modules

Table 1.7 provides a summary of a few available modules that are relevant to a study of data structures. We have already discussed the math module briefly. In the remainder of this section, we highlight another module that is particularly important for some of the data structures and algorithms that we will study later in this book.

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Table 1.7: Some existing Python modules relevant to data structures and algorithms.

Pseudo-Random Number Generation

Python's random module provides the ability to generate pseudo-random numbers, that is, numbers that are statistically random (but not necessarily truly random). A ***pseudo-random number generator*** uses a deterministic formula to generate the

next number in a sequence based upon one or more past numbers that it has generated. Indeed, a simple yet popular pseudo-random number generator chooses its next number based solely on the most recently chosen number and some additional parameters using the following formula.

```
next = (a*current + b) % n;
```

where a, b, and n are appropriately chosen integers. Python uses a more advanced technique known as a ***Mersenne twister***. It turns out that the sequences generated by these techniques can be proven to be statistically uniform, which is usually good enough for most applications requiring random numbers, such as games. For applications, such as computer security settings, where one needs unpredictable random sequences, this kind of formula should not be used. Instead, one should ideally sample from a source that is actually random, such as radio static coming from outer space.

Since the next number in a pseudo-random generator is determined by the previous number(s), such a generator always needs a place to start, which is called its ***seed***. The sequence of numbers generated for a given seed will always be the same. One common trick to get a different sequence each time a program is run is to use a seed that will be different for each run. For example, we could use some timed input from a user or the current system time in milliseconds.

Python's random module provides support for pseudo-random number generation by defining a Random class; instances of that class serve as generators with independent state. This allows different aspects of a program to rely on their own pseudo-random number generator, so that calls to one generator do not affect the sequence of numbers produced by another. For convenience, all of the methods supported by the Random class are also supported as stand-alone functions of the random module (essentially using a single generator instance for all top-level calls).

Syntax	Description
seed(hashable)	Initializes the pseudo-random number generator based upon the hash value of the parameter
random( )	Returns a pseudo-random floating-point value in the interval $[0.0, 1.0)$.
randint(a,b)	Returns a pseudo-random integer in the closed interval $[a, b]$.
randrange(start, stop, step)	Returns a pseudo-random integer in the standard Python range indicated by the parameters.
choice(seq)	Returns an element of the given sequence chosen pseudo-randomly.
shuffle(seq)	Reorders the elements of the given sequence pseudo-randomly.

Table 1.8: Methods supported by instances of the Random class, and as top-level functions of the random module.

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.

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).

C-1.22 Write a short Python program that takes two arrays a and b of length n storing **int** values, and returns the dot product of a and b. That is, it returns an array c of length n such that $c[i] = a[i] \cdot b[i]$, for $i = 0, \ldots, n-1$.

C-1.23 Give an example of a Python code fragment that attempts to write an element to a list based on an index that may be out of bounds. If that index is out of bounds, the program should catch the exception that results, and print the following error message:
"`Don't try buffer overflow attacks in Python!`"

C-1.24 Write a short Python function that counts the number of vowels in a given character string.

C-1.25 Write a short Python function that takes a string s, representing a sentence, and returns a copy of the string with all punctuation removed. For example, if given the string `"Let's try, Mike."`, this function would return `"Lets try Mike"`.

C-1.26 Write a short program that takes as input three integers, a, b, and c, from the console and determines if they can be used in a correct arithmetic formula (in the given order), like "$a+b=c$," "$a=b-c$," or "$a*b=c$."

C-1.27 In Section 1.8, we provided three different implementations of a generator that computes factors of a given integer. The third of those implementations, from page 41, was the most efficient, but we noted that it did not yield the factors in increasing order. Modify the generator so that it reports factors in increasing order, while maintaining its general performance advantages.

C-1.28 The ***p-norm*** of a vector $v = (v_1, v_2, \ldots, v_n)$ in n-dimensional space is defined as

$$\|v\| = \sqrt[p]{v_1^p + v_2^p + \cdots + v_n^p}.$$

For the special case of $p = 2$, this results in the traditional ***Euclidean norm***, which represents the length of the vector. For example, the Euclidean norm of a two-dimensional vector with coordinates $(4,3)$ has a Euclidean norm of $\sqrt{4^2+3^2} = \sqrt{16+9} = \sqrt{25} = 5$. Give an implementation of a function named `norm` such that `norm(v, p)` returns the p-norm value of v and `norm(v)` returns the Euclidean norm of v. You may assume that v is a list of numbers.

Projects

P-1.29 Write a Python program that outputs all possible strings formed by using the characters `'c'`, `'a'`, `'t'`, `'d'`, `'o'`, and `'g'` exactly once.

P-1.30 Write a Python program that can take a positive integer greater than 2 as input and write out the number of times one must repeatedly divide this number by 2 before getting a value less than 2.

P-1.31 Write a Python program that can "make change." Your program should take two numbers as input, one that is a monetary amount charged and the other that is a monetary amount given. It should then return the number of each kind of bill and coin to give back as change for the difference between the amount given and the amount charged. The values assigned to the bills and coins can be based on the monetary system of any current or former government. Try to design your program so that it returns as few bills and coins as possible.

P-1.32 Write a Python program that can simulate a simple calculator, using the console as the exclusive input and output device. That is, each input to the calculator, be it a number, like `12.34` or `1034`, or an operator, like `+` or `=`, can be done on a separate line. After each such input, you should output to the Python console what would be displayed on your calculator.

P-1.33 Write a Python program that simulates a handheld calculator. Your program should process input from the Python console representing buttons that are "pushed," and then output the contents of the screen after each operation is performed. Minimally, your calculator should be able to process the basic arithmetic operations and a reset/clear operation.

P-1.34 A common punishment for school children is to write out a sentence multiple times. Write a Python stand-alone program that will write out the following sentence one hundred times: "I will never spam my friends again." Your program should number each of the sentences and it should make eight different random-looking typos.

P-1.35 The ***birthday paradox*** says that the probability that two people in a room will have the same birthday is more than half, provided n, the number of people in the room, is more than 23. This property is not really a paradox, but many people find it surprising. Design a Python program that can test this paradox by a series of experiments on randomly generated birthdays, which test this paradox for $n = 5, 10, 15, 20, \ldots, 100$.

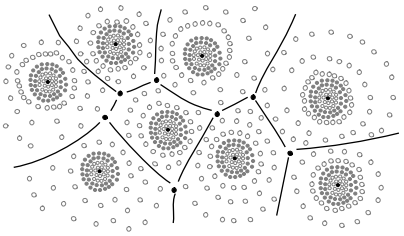
P-1.36 Write a Python program that inputs a list of words, separated by whitespace, and outputs how many times each word appears in the list. You need not worry about efficiency at this point, however, as this topic is something that will be addressed later in this book.

Chapter Notes

The official Python Web site (http://www.python.org) has a wealth of information, including a tutorial and full documentation of the built-in functions, classes, and standard modules. The Python interpreter is itself a useful reference, as the interactive command help(foo) provides documentation for any function, class, or module that foo identifies.

Books providing an introduction to programming in Python include titles authored by Campbell *et al.* [22], Cedar [25], Dawson [32], Goldwasser and Letscher [43], Lutz [72], Perkovic [82], and Zelle [105]. More complete reference books on Python include titles by Beazley [12], and Summerfield [91].

Chapter 2 Object-Oriented Programming



Contents

2.1 Goals, Principles, and Patterns

As the name implies, the main "actors" in the object-oriented paradigm are called ***objects***. Each object is an ***instance*** of a ***class***. Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects. The class definition typically specifies ***instance variables***, also known as ***data members***, that the object contains, as well as the ***methods***, also known as ***member functions***, that the object can execute. This view of computing is intended to fulfill several goals and incorporate several design principles, which we discuss in this chapter.

2.1.1 Object-Oriented Design Goals

Software implementations should achieve ***robustness***, ***adaptability***, and ***reusability***. (See Figure 2.1.)

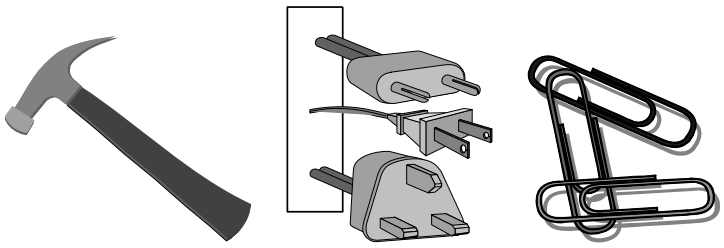


Figure 2.1: Goals of object-oriented design.

Robustness

Every good programmer wants to develop software that is correct, which means that a program produces the right output for all the anticipated inputs in the program's application. In addition, we want software to be ***robust***, that is, capable of handling unexpected inputs that are not explicitly defined for its application. For example, if a program is expecting a positive integer (perhaps representing the price of an item) and instead is given a negative integer, then the program should be able to recover gracefully from this error. More importantly, in ***life-critical applications***, where a software error can lead to injury or loss of life, software that is not robust could be deadly. This point was driven home in the late 1980s in accidents involving Therac-25, a radiation-therapy machine, which severely overdosed six patients between 1985 and 1987, some of whom died from complications resulting from their radiation overdose. All six accidents were traced to software errors.

Adaptability

Modern software applications, such as Web browsers and Internet search engines, typically involve large programs that are used for many years. Software, therefore, needs to be able to evolve over time in response to changing conditions in its environment. Thus, another important goal of quality software is that it achieves ***adaptability*** (also called ***evolvability***). Related to this concept is ***portability***, which is the ability of software to run with minimal change on different hardware and operating system platforms. An advantage of writing software in Python is the portability provided by the language itself.

Reusability

Going hand in hand with adaptability is the desire that software be reusable, that is, the same code should be usable as a component of different systems in various applications. Developing quality software can be an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily reusable in future applications. Such reuse should be done with care, however, for one of the major sources of software errors in the Therac-25 came from inappropriate reuse of Therac-20 software (which was not object-oriented and not designed for the hardware platform used with the Therac-25).

2.1.2 Object-Oriented Design Principles

Chief among the principles of the object-oriented approach, which are intended to facilitate the goals outlined above, are the following (see Figure 2.2):

- Modularity
- Abstraction
- Encapsulation

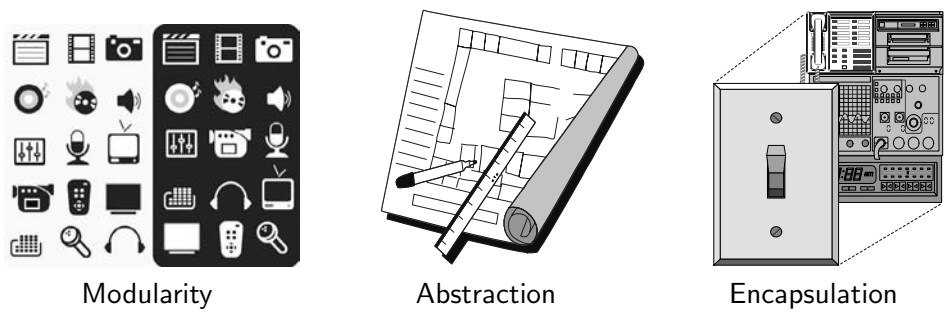


Figure 2.2: Principles of object-oriented design.

Modularity

Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized. Modularity refers to an organizing principle in which different components of a software system are divided into separate functional units.

As a real-world analogy, a house or apartment can be viewed as consisting of several interacting units: electrical, heating and cooling, plumbing, and structural. Rather than viewing these systems as one giant jumble of wires, vents, pipes, and boards, the organized architect designing a house or apartment will view them as separate modules that interact in well-defined ways. In so doing, he or she is using modularity to bring a clarity of thought that provides a natural way of organizing functions into distinct manageable units.

In like manner, using modularity in a software system can also provide a powerful organizing framework that brings clarity to an implementation. In Python, we have already seen that a ***module*** is a collection of closely related functions and classes that are defined together in a single file of source code. Python's standard libraries include, for example, the `math` module, which provides definitions for key mathematical constants and functions, and the `os` module, which provides support for interacting with the operating system.

The use of modularity helps support the goals listed in Section 2.1.1. Robustness is greatly increased because it is easier to test and debug separate components before they are integrated into a larger software system. Furthermore, bugs that persist in a complete system might be traced to a particular component, which can be fixed in relative isolation. The structure imposed by modularity also helps enable software reusability. If software modules are written in a general way, the modules can be reused when related need arises in other contexts. This is particularly relevant in a study of data structures, which can typically be designed with sufficient abstraction and generality to be reused in many applications.

Abstraction

The notion of ***abstraction*** is to distill a complicated system down to its most fundamental parts. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to ***abstract data types*** (ADTs). An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations. An ADT specifies ***what*** each operation does, but not ***how*** it does it. We will typically refer to the collective set of behaviors supported by an ADT as its ***public interface***.

As a programming language, Python provides a great deal of latitude in regard to the specification of an interface. Python has a tradition of treating abstractions implicitly using a mechanism known as ***duck typing***. As an interpreted and dynamically typed language, there is no "compile time" checking of data types in Python, and no formal requirement for declarations of abstract base classes. Instead programmers assume that an object supports a set of known behaviors, with the interpreter raising a run-time error if those assumptions fail. The description of this as "duck typing" comes from an adage attributed to poet James Whitcomb Riley, stating that "when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

More formally, Python supports abstract data types using a mechanism known as an ***abstract base class*** (ABC). An abstract base class cannot be instantiated (i.e., you cannot directly create an instance of that class), but it defines one or more common methods that all implementations of the abstraction must have. An ABC is realized by one or more ***concrete classes*** that inherit from the abstract base class while providing implementations for those method declared by the ABC. Python's `abc` module provides formal support for ABCs, although we omit such declarations for simplicity. We will make use of several existing abstract base classes coming from Python's `collections` module, which includes definitions for several common data structure ADTs, and concrete implementations of some of those abstractions.

Encapsulation

Another important principle of object-oriented design is ***encapsulation***. Different components of a software system should not reveal the internal details of their respective implementations. One of the main advantages of encapsulation is that it gives one programmer freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions. The only constraint on the programmer of a component is to maintain the public interface for the component, as other programmers will be writing code that depends on that interface. Encapsulation yields robustness and adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

Throughout this book, we will adhere to the principle of encapsulation, making clear which aspects of a data structure are assumed to be public and which are assumed to be internal details. With that said, Python provides only loose support for encapsulation. By convention, names of members of a class (both data members and member functions) that start with a single underscore character (e.g., `_secret`) are assumed to be nonpublic and should not be relied upon. Those conventions are reinforced by the intentional omission of those members from automatically generated documentation.

2.1.3 Design Patterns

Object-oriented design facilitates reusable, robust, and adaptable software. Designing good code takes more than simply understanding object-oriented methodologies, however. It requires the effective use of object-oriented design techniques.

Computing researchers and practitioners have developed a variety of organizational concepts and methodologies for designing quality object-oriented software that is concise, correct, and reusable. Of special relevance to this book is the concept of a ***design pattern***, which describes a solution to a "typical" software design problem. A pattern provides a general template for a solution that can be applied in many different situations. It describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand. It consists of a name, which identifies the pattern; a context, which describes the scenarios for which this pattern can be applied; a template, which describes how the pattern is applied; and a result, which describes and analyzes what the pattern produces.

We present several design patterns in this book, and we show how they can be consistently applied to implementations of data structures and algorithms. These design patterns fall into two groups—patterns for solving algorithm design problems and patterns for solving software engineering problems. The algorithm design patterns we discuss include the following:

- Recursion (Chapter 4)
- Amortization (Sections 5.3 and 11.4)
- Divide-and-conquer (Section 12.2.1)
- Prune-and-search, also known as decrease-and-conquer (Section 12.7.1)
- Brute force (Section 13.2.1)
- Dynamic programming (Section 13.3).
- The greedy method (Sections 13.4.2, 14.6.2, and 14.7)

Likewise, the software engineering design patterns we discuss include:

- Iterator (Sections 1.8 and 2.3.4)
- Adapter (Section 6.1.2)
- Position (Sections 7.4 and 8.1.2)
- Composition (Sections 7.6.1, 9.2.1, and 10.1.4)
- Template method (Sections 2.4.3, 8.4.6, 10.1.3, 10.5.2, and 11.2.1)
- Locator (Section 9.5.1)
- Factory method (Section 11.2.1)

Rather than explain each of these concepts here, however, we introduce them throughout the text as noted above. For each pattern, be it for algorithm engineering or software engineering, we explain its general use and we illustrate it with at least one concrete example.

2.2 Software Development

Traditional software development involves several phases. Three major steps are:

1. Design
2. Implementation
3. Testing and Debugging

In this section, we briefly discuss the role of these phases, and we introduce several good practices for programming in Python, including coding style, naming conventions, formal documentation, and unit testing.

2.2.1 Design

For object-oriented programming, the design step is perhaps the most important phase in the process of developing software. For it is in the design step that we decide how to divide the workings of our program into classes, we decide how these classes will interact, what data each will store, and what actions each will perform. Indeed, one of the main challenges that beginning programmers face is deciding what classes to define to do the work of their program. While general prescriptions are hard to come by, there are some rules of thumb that we can apply when determining how to design our classes:

- ***Responsibilities***: Divide the work into different ***actors***, each with a different responsibility. Try to describe responsibilities using action verbs. These actors will form the classes for the program.
- ***Independence***: Define the work for each class to be as independent from other classes as possible. Subdivide responsibilities between classes so that each class has autonomy over some aspect of the program. Give data (as instance variables) to the class that has jurisdiction over the actions that require access to this data.
- ***Behaviors***: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it. These behaviors will define the methods that this class performs, and the set of behaviors for a class are the ***interface*** to the class, as these form the means for other pieces of code to interact with objects from the class.

Defining the classes, together with their instance variables and methods, are key to the design of an object-oriented program. A good programmer will naturally develop greater skill in performing these tasks over time, as experience teaches him or her to notice patterns in the requirements of a program that match patterns that he or she has seen before.

A common tool for developing an initial high-level design for a project is the use of ***CRC cards***. Class-Responsibility-Collaborator (CRC) cards are simple index cards that subdivide the work required of a program. The main idea behind this tool is to have each card represent a component, which will ultimately become a class in the program. We write the name of each component on the top of an index card. On the left-hand side of the card, we begin writing the responsibilities for this component. On the right-hand side, we list the collaborators for this component, that is, the other components that this component will have to interact with to perform its duties.

The design process iterates through an action/actor cycle, where we first identify an action (that is, a responsibility), and we then determine an actor (that is, a component) that is best suited to perform that action. The design is complete when we have assigned all actions to actors. In using index cards for this process (rather than larger pieces of paper), we are relying on the fact that each component should have a small set of responsibilities and collaborators. Enforcing this rule helps keep the individual classes manageable.

As the design takes form, a standard approach to explain and document the design is the use of UML (Unified Modeling Language) diagrams to express the organization of a program. UML diagrams are a standard visual notation to express object-oriented software designs. Several computer-aided tools are available to build UML diagrams. One type of UML figure is known as a ***class diagram***. An example of such a diagram is given in Figure 2.3, for a class that represents a consumer credit card. The diagram has three portions, with the first designating the name of the class, the second designating the recommended instance variables, and the third designating the recommended methods of the class. In Section 2.2.3, we discuss our naming conventions, and in Section 2.3.1, we provide a complete implementation of a Python CreditCard class based on this design.

Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit
Behaviors:	get_customer( ) get_bank( ) get_account( ) make_payment(amount)	get_balance( ) get_limit( ) charge(price)

Figure 2.3: Class diagram for a proposed CreditCard class.

2.2.2 Pseudo-Code

As an intermediate step before the implementation of a design, programmers are often asked to describe algorithms in a way that is intended for human eyes only. Such descriptions are called ***pseudo-code***. Pseudo-code is not a computer program, but is more structured than usual prose. It is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm. Because pseudo-code is designed for a human reader, not a computer, we can communicate high-level ideas, without being burdened with low-level implementation details. At the same time, we should not gloss over important steps. Like many forms of human communication, finding the right balance is an important skill that is refined through practice.

In this book, we rely on a pseudo-code style that we hope will be evident to Python programmers, yet with a mix of mathematical notations and English prose. For example, we might use the phrase "indicate an error" rather than a formal raise statement. Following conventions of Python, we rely on indentation to indicate the extent of control structures and on an indexing notation in which entries of a sequence A with length n are indexed from $A[0]$ to $A[n-1]$. However, we choose to enclose comments within curly braces { like these } in our pseudo-code, rather than using Python's # character.

2.2.3 Coding Style and Documentation

Programs should be made easy to read and understand. Good programmers should therefore be mindful of their coding style, and develop a style that communicates the important aspects of a program's design for both humans and computers. Conventions for coding style tend to vary between different programming communities. The official *Style Guide for Python Code* is available online at

`http://www.python.org/dev/peps/pep-0008/`

The main principles that we adopt are as follows:

- Python code blocks are typically indented by 4 spaces. However, to avoid having our code fragments overrun the book's margins, we use 2 spaces for each level of indentation. It is strongly recommended that tabs be avoided, as tabs are displayed with differing widths across systems, and tabs and spaces are not viewed as identical by the Python interpreter. Many Python-aware editors will automatically replace tabs with an appropriate number of spaces.

- Use meaningful names for identifiers. Try to choose names that can be read aloud, and choose names that reflect the action, responsibility, or data each identifier is naming.

  - Classes (other than Python's built-in classes) should have a name that serves as a singular noun, and should be capitalized (e.g., Date rather than date or Dates). When multiple words are concatenated to form a class name, they should follow the so-called "CamelCase" convention in which the first letter of each word is capitalized (e.g., CreditCard).

  - Functions, including member functions of a class, should be lowercase. If multiple words are combined, they should be separated by underscores (e.g., make_payment). The name of a function should typically be a verb that describes its affect. However, if the only purpose of the function is to return a value, the function name may be a noun that describes the value (e.g., sqrt rather than calculate_sqrt).

  - Names that identify an individual object (e.g., a parameter, instance variable, or local variable) should be a lowercase noun (e.g., price). Occasionally, we stray from this rule when using a single uppercase letter to designate the name of a data structures (such as tree T).

  - Identifiers that represent a value considered to be a constant are traditionally identified using all capital letters and with underscores to separate words (e.g., MAX_SIZE).

  Recall from our discussion of ***encapsulation*** that identifiers in any context that begin with a single leading underscore (e.g., _secret) are intended to suggest that they are only for "internal" use to a class or module, and not part of a public interface.

- Use comments that add meaning to a program and explain ambiguous or confusing constructs. In-line comments are good for quick explanations; they are indicated in Python following the # character, as in

  ```
  if n % 2 == 1:          # n is odd
  ```

  Multiline block comments are good for explaining more complex code sections. In Python, these are technically multiline string literals, typically delimited with triple quotes (`"""`), which have no effect when executed. In the next section, we discuss the use of block comments for documentation.

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.

2.2.4 Testing and Debugging

Testing is the process of experimentally checking the correctness of a program, while debugging is the process of tracking the execution of a program and discovering the errors in it. Testing and debugging are often the most time-consuming activity in the development of a program.

Testing

A careful testing plan is an essential part of writing a program. While verifying the correctness of a program over all possible inputs is usually infeasible, we should aim at executing the program on a representative subset of inputs. At the very minimum, we should make sure that every method of a class is tested at least once (method coverage). Even better, each code statement in the program should be executed at least once (statement coverage).

Programs often tend to fail on ***special cases*** of the input. Such cases need to be carefully identified and tested. For example, when testing a method that sorts (that is, puts in order) a sequence of integers, we should consider the following inputs:

- The sequence has zero length (no elements).
- The sequence has one element.
- All the elements of the sequence are the same.
- The sequence is already sorted.
- The sequence is reverse sorted.

In addition to special inputs to the program, we should also consider special conditions for the structures used by the program. For example, if we use a Python list to store data, we should make sure that boundary cases, such as inserting or removing at the beginning or end of the list, are properly handled.

While it is essential to use handcrafted test suites, it is also advantageous to run the program on a large collection of randomly generated inputs. The random module in Python provides several means for generating random numbers, or for randomizing the order of collections.

The dependencies among the classes and functions of a program induce a hierarchy. Namely, a component A is above a component B in the hierarchy if A depends upon B, such as when function A calls function B, or function A relies on a parameter that is an instance of class B. There are two main testing strategies, ***top-down*** and ***bottom-up***, which differ in the order in which components are tested.

Top-down testing proceeds from the top to the bottom of the program hierarchy. It is typically used in conjunction with ***stubbing***, a boot-strapping technique that replaces a lower-level component with a ***stub***, a replacement for the component that simulates the functionality of the original. For example, if function A calls function B to get the first line of a file, when testing A we can replace B with a stub that returns a fixed string.

Bottom-up testing proceeds from lower-level components to higher-level components. For example, bottom-level functions, which do not invoke other functions, are tested first, followed by functions that call only bottom-level functions, and so on. Similarly a class that does not depend upon any other classes can be tested before another class that depends on the former. This form of testing is usually described as ***unit testing***, as the functionality of a specific component is tested in isolation of the larger software project. If used properly, this strategy better isolates the cause of errors to the component being tested, as lower-level components upon which it relies should have already been thoroughly tested.

Python provides several forms of support for automated testing. When functions or classes are defined in a module, testing for that module can be embedded in the same file. The mechanism for doing so was described in Section 1.11. Code that is shielded in a conditional construct of the form

```
if __name__ == '__main__':
  # perform tests...
```

will be executed when Python is invoked directly on that module, but not when the module is imported for use in a larger software project. It is common to put tests in such a construct to test the functionality of the functions and classes specifically defined in that module.

More robust support for automation of unit testing is provided by Python's `unittest` module. This framework allows the grouping of individual test cases into larger test suites, and provides support for executing those suites, and reporting or analyzing the results of those tests. As software is maintained, the act of ***regression testing*** is used, whereby all previous tests are re-executed to ensure that changes to the software do not introduce new bugs in previously tested components.

Debugging

The simplest debugging technique consists of using ***print statements*** to track the values of variables during the execution of the program. A problem with this approach is that eventually the print statements need to be removed or commented out, so they are not executed when the software is finally released.

A better approach is to run the program within a ***debugger***, which is a specialized environment for controlling and monitoring the execution of a program. The basic functionality provided by a debugger is the insertion of ***breakpoints*** within the code. When the program is executed within the debugger, it stops at each breakpoint. While the program is stopped, the current value of variables can be inspected.

The standard Python distribution includes a module named `pdb`, which provides debugging support directly within the interpreter. Most IDEs for Python, such as IDLE, provide debugging environments with graphical user interfaces.

2.3 Class Definitions

A class serves as the primary means for abstraction in object-oriented programming. In Python, every piece of data is represented as an instance of some class. A class provides a set of behaviors in the form of ***member functions*** (also known as ***methods***), with implementations that are common to all instances of that class. A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of ***attributes*** (also known as ***fields***, ***instance variables***, or ***data members***).

2.3.1 Example: CreditCard Class

As a first example, we provide an implementation of a CreditCard class based on the design we introduced in Figure 2.3 of Section 2.2.1. The instances defined by the CreditCard class provide a simple model for traditional credit cards. They have identifying information about the customer, bank, account number, credit limit, and current balance. The class restricts charges that would cause a card's balance to go over its spending limit, but it does not charge interest or late payments (we revisit such themes in Section 2.4.1).

Our code begins in Code Fragment 2.1 and continues in Code Fragment 2.2. The construct begins with the keyword, **class**, followed by the name of the class, a colon, and then an indented block of code that serves as the body of the class. The body includes definitions for all methods of the class. These methods are defined as functions, using techniques introduced in Section 1.5, yet with a special parameter, named **self**, that serves to identify the particular instance upon which a member is invoked.

The self Identifier

In Python, the self identifier plays a key role. In the context of the CreditCard class, there can presumably be many different CreditCard instances, and each must maintain its own balance, its own credit limit, and so on. Therefore, each instance stores its own instance variables to reflect its current state.

Syntactically, self identifies the instance upon which a method is invoked. For example, assume that a user of our class has a variable, my_card, that identifies an instance of the CreditCard class. When the user calls my_card.get_balance( ), identifier self, within the definition of the get_balance method, refers to the card known as my_card by the caller. The expression, self._balance refers to an instance variable, named _balance, stored as part of that particular credit card's state.

```
1  class CreditCard:
2    """A consumer credit card."""
3
4    def __init__(self, customer, bank, acnt, limit):
5      """Create a new credit card instance.
6
7      The initial balance is zero.
8
9      customer  the name of the customer (e.g., 'John Bowman')
10     bank      the name of the bank (e.g., 'California Savings')
11     acnt      the acount identifier (e.g., '5391 0375 9387 5309')
12     limit     credit limit (measured in dollars)
13     """
14     self._customer = customer
15     self._bank = bank
16     self._account = acnt
17     self._limit = limit
18     self._balance = 0
19
20   def get_customer(self):
21     """Return name of the customer."""
22     return self._customer
23
24   def get_bank(self):
25     """Return the bank's name."""
26     return self._bank
27
28   def get_account(self):
29     """Return the card identifying number (typically stored as a string)."""
30     return self._account
31
32   def get_limit(self):
33     """Return current credit limit."""
34     return self._limit
35
36   def get_balance(self):
37     """Return current balance."""
38     return self._balance
```

Code Fragment 2.1: The beginning of the CreditCard class definition (continued in Code Fragment 2.2).

```
39    def charge(self, price):
40      """Charge given price to the card, assuming sufficient credit limit.
41
42      Return True if charge was processed; False if charge was denied.
43      """
44      if price + self._balance > self._limit:      # if charge would exceed limit,
45        return False                               # cannot accept charge
46      else:
47        self._balance += price
48        return True
49
50    def make_payment(self, amount):
51      """Process customer payment that reduces balance."""
52      self._balance -= amount
```

Code Fragment 2.2: The conclusion of the CreditCard class definition (continued from Code Fragment 2.1). These methods are indented within the class definition.

We draw attention to the difference between the method signature as declared within the class versus that used by a caller. For example, from a user's perspective we have seen that the get_balance method takes zero parameters, yet within the class definition, self is an explicit parameter. Likewise, the charge method is declared within the class having two parameters (self and price), even though this method is called with one parameter, for example, as my_card.charge(200). The interpretter automatically binds the instance upon which the method is invoked to the self parameter.

The Constructor

A user can create an instance of the CreditCard class using a syntax as:

```
cc = CreditCard('John Doe, '1st Bank', '5391 0375 9387 5309', 1000)
```

Internally, this results in a call to the specially named __init__ method that serves as the ***constructor*** of the class. Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables. In the case of the CreditCard class, each object maintains five instance variables, which we name: _customer, _bank, _account, _limit, and _balance. The initial values for the first four of those five are provided as explicit parameters that are sent by the user when instantiating the credit card, and assigned within the body of the constructor. For example, the command, self._customer = customer, assigns the instance variable self._customer to the parameter customer; note that because customer is ***unqualified*** on the right-hand side, it refers to the parameter in the local namespace.

Encapsulation

By the conventions described in Section 2.2.3, a single leading underscore in the name of a data member, such as _balance, implies that it is intended as ***nonpublic***. Users of a class should not directly access such members.

As a general rule, we will treat all data members as nonpublic. This allows us to better enforce a consistent state for all instances. We can provide accessors, such as get_balance, to provide a user of our class read-only access to a trait. If we wish to allow the user to change the state, we can provide appropriate update methods. In the context of data structures, encapsulating the internal representation allows us greater flexibility to redesign the way a class works, perhaps to improve the efficiency of the structure.

Additional Methods

The most interesting behaviors in our class are charge and make_payment. The charge function typically adds the given price to the credit card balance, to reflect a purchase of said price by the customer. However, before accepting the charge, our implementation verifies that the new purchase would not cause the balance to exceed the credit limit. The make_payment charge reflects the customer sending payment to the bank for the given amount, thereby reducing the balance on the card. We note that in the command, self._balance $-=$ amount, the expression self._balance is qualified with the self identifier because it represents an instance variable of the card, while the unqualified amount represents the local parameter.

Error Checking

Our implementation of the CreditCard class is not particularly robust. First, we note that we did not explicitly check the types of the parameters to charge and make_payment, nor any of the parameters to the constructor. If a user were to make a call such as visa.charge('candy'), our code would presumably crash when attempting to add that parameter to the current balance. If this class were to be widely used in a library, we might use more rigorous techniques to raise a TypeError when facing such misuse (see Section 1.7).

Beyond the obvious type errors, our implementation may be susceptible to logical errors. For example, if a user were allowed to charge a negative price, such as visa.charge(-300), that would serve to *lower* the customer's balance. This provides a loophole for lowering a balance without making a payment. Of course, this might be considered valid usage if modeling the credit received when a customer returns merchandise to a store. We will explore some such issues with the CreditCard class in the end-of-chapter exercises.

Testing the Class

In Code Fragment 2.3, we demonstrate some basic usage of the CreditCard class, inserting three cards into a list named wallet. We use loops to make some charges and payments, and use various accessors to print results to the console.

These tests are enclosed within a conditional, **if** `__name__ == '__main__'`:, so that they can be embedded in the source code with the class definition. Using the terminology of Section 2.2.4, these tests provide ***method coverage***, as each of the methods is called at least once, but it does not provide ***statement coverage***, as there is never a case in which a charge is rejected due to the credit limit. This is not a particular advanced from of testing as the output of the given tests must be manually audited in order to determine whether the class behaved as expected. Python has tools for more formal testing (see discussion of the unittest module in Section 2.2.4), so that resulting values can be automatically compared to the predicted outcomes, with output generated only when an error is detected.

```
53 if __name__ == '__main__':
54   wallet = [ ]
55   wallet.append(CreditCard('John Bowman', 'California Savings',
56                            '5391 0375 9387 5309', 2500) )
57   wallet.append(CreditCard('John Bowman', 'California Federal',
58                            '3485 0399 3395 1954', 3500) )
59   wallet.append(CreditCard('John Bowman', 'California Finance',
60                            '5391 0375 9387 5309', 5000) )
61
62   for val in range(1, 17):
63     wallet[0].charge(val)
64     wallet[1].charge(2*val)
65     wallet[2].charge(3*val)
66
67   for c in range(3):
68     print('Customer =', wallet[c].get_customer( ))
69     print('Bank =', wallet[c].get_bank( ))
70     print('Account =', wallet[c].get_account( ))
71     print('Limit =', wallet[c].get_limit( ))
72     print('Balance =', wallet[c].get_balance( ))
73     while wallet[c].get_balance( ) > 100:
74       wallet[c].make_payment(100)
75       print('New balance =', wallet[c].get_balance( ))
76     print( )
```

Code Fragment 2.3: Testing the CreditCard class.

2.3.2 Operator Overloading and Python's Special Methods

Python's built-in classes provide natural semantics for many operators. For example, the syntax a + b invokes addition for numeric types, yet concatenation for sequence types. When defining a new class, we must consider whether a syntax like a + b should be defined when a or b is an instance of that class.

By default, the + operator is undefined for a new class. However, the author of a class may provide a definition using a technique known as ***operator overloading***. This is done by implementing a specially named method. In particular, the + operator is overloaded by implementing a method named __add__, which takes the right-hand operand as a parameter and which returns the result of the expression. That is, the syntax, a + b, is converted to a method call on object a of the form, a.__add__(b). Similar specially named methods exist for other operators. Table 2.1 provides a comprehensive list of such methods.

When a binary operator is applied to two instances of different types, as in 3 * 'love me', Python gives deference to the class of the *left* operand. In this example, it would effectively check if the int class provides a sufficient definition for how to multiply an instance by a string, via the __mul__ method. However, if that class does not implement such a behavior, Python checks the class definition for the right-hand operand, in the form of a special method named __rmul__ (i.e., "right multiply"). This provides a way for a new user-defined class to support mixed operations that involve an instance of an existing class (given that the existing class would presumably not have defined a behavior involving this new class). The distinction between __mul__ and __rmul__ also allows a class to define different semantics in cases, such as matrix multiplication, in which an operation is noncommutative (that is, A * x may differ from x * A).

Non-Operator Overloads

In addition to traditional operator overloading, Python relies on specially named methods to control the behavior of various other functionality, when applied to user-defined classes. For example, the syntax, str(foo), is formally a call to the constructor for the string class. Of course, if the parameter is an instance of a user-defined class, the original authors of the string class could not have known how that instance should be portrayed. So the string constructor calls a specially named method, foo.__str__(), that must return an appropriate string representation.

Similar special methods are used to determine how to construct an int, float, or bool based on a parameter from a user-defined class. The conversion to a Boolean value is particularly important, because the syntax, **if** foo:, can be used even when foo is not formally a Boolean value (see Section 1.4.1). For a user-defined class, that condition is evaluated by the special method foo.__bool__().

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b a −= b a *= b ...	a.__iadd__(b) a.__isub__(b) a.__imul__(b) ...	
+a	a.__pos__()	
−a	a.__neg__()	
~a	a.__invert__()	
abs(a)	a.__abs__()	
a < b	a.__lt__(b)	
a <= b	a.__le__(b)	
a > b	a.__gt__(b)	
a >= b	a.__ge__(b)	
a == b	a.__eq__(b)	
a != b	a.__ne__(b)	
v in a	a.__contains__(v)	
a[k]	a.__getitem__(k)	
a[k] = v	a.__setitem__(k,v)	
del a[k]	a.__delitem__(k)	
a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)	
len(a)	a.__len__()	
hash(a)	a.__hash__()	
iter(a)	a.__iter__()	
next(a)	a.__next__()	
bool(a)	a.__bool__()	
float(a)	a.__float__()	
int(a)	a.__int__()	
repr(a)	a.__repr__()	
reversed(a)	a.__reversed__()	
str(a)	a.__str__()	

Table 2.1: Overloaded operations, implemented with Python's special methods.

Several other top-level functions rely on calling specially named methods. For example, the standard way to determine the size of a container type is by calling the top-level len function. Note well that the calling syntax, len(foo), is not the traditional method-calling syntax with the dot operator. However, in the case of a user-defined class, the top-level len function relies on a call to a specially named __len__ method of that class. That is, the call len(foo) is evaluated through a method call, foo.__len__(). When developing data structures, we will routinely define the __len__ method to return a measure of the size of the structure.

Implied Methods

As a general rule, if a particular special method is not implemented in a user-defined class, the standard syntax that relies upon that method will raise an exception. For example, evaluating the expression, a + b, for instances of a user-defined class without __add__ or __radd__ will raise an error.

However, there are some operators that have default definitions provided by Python, in the absence of special methods, and there are some operators whose definitions are derived from others. For example, the __bool__ method, which supports the syntax **if** foo:, has default semantics so that every object other than None is evaluated as True. However, for container types, the __len__ method is typically defined to return the size of the container. If such a method exists, then the evaluation of bool(foo) is interpreted by default to be True for instances with nonzero length, and False for instances with zero length, allowing a syntax such as **if** waitlist: to be used to test whether there are one or more entries in the waitlist.

In Section 2.3.4, we will discuss Python's mechanism for providing iterators for collections via the special method, __iter__. With that said, if a container class provides implementations for both __len__ and __getitem__, a default iteration is provided automatically (using means we describe in Section 2.3.4). Furthermore, once an iterator is defined, default functionality of __contains__ is provided.

In Section 1.3 we drew attention to the distinction between expression a **is** b and expression a == b, with the former evaluating whether identifiers a and b are aliases for the same object, and the latter testing a notion of whether the two identifiers reference *equivalent* values. The notion of "equivalence" depends upon the context of the class, and semantics is defined with the __eq__ method. However, if no implementation is given for __eq__, the syntax a == b is legal with semantics of a **is** b, that is, an instance is equivalent to itself and no others.

We should caution that some natural implications are *not* automatically provided by Python. For example, the __eq__ method supports syntax a == b, but providing that method does not affect the evaluation of syntax a != b. (The __ne__ method should be provided, typically returning **not** (a == b) as a result.) Similarly, providing a __lt__ method supports syntax a < b, and indirectly b > a, but providing both __lt__ and __eq__ does *not* imply semantics for a <= b.

2.3.3 Example: Multidimensional Vector Class

To demonstrate the use of operator overloading via special methods, we provide an implementation of a Vector class, representing the coordinates of a vector in a multidimensional space. For example, in a three-dimensional space, we might wish to represent a vector with coordinates $\langle 5, -2, 3 \rangle$. Although it might be tempting to directly use a Python list to represent those coordinates, a list does not provide an appropriate abstraction for a geometric vector. In particular, if using lists, the expression [5, −2, 3] + [1, 4, 2] results in the list [5, −2, 3, 1, 4, 2]. When working with vectors, if $u = \langle 5, -2, 3 \rangle$ and $v = \langle 1, 4, 2 \rangle$, one would expect the expression, u + v, to return a three-dimensional vector with coordinates $\langle 6, 2, 5 \rangle$.

We therefore define a Vector class, in Code Fragment 2.4, that provides a better abstraction for the notion of a geometric vector. Internally, our vector relies upon an instance of a list, named _coords, as its storage mechanism. By keeping the internal list encapsulated, we can enforce the desired public interface for instances of our class. A demonstration of supported behaviors includes the following:

```
v = Vector(5)                  # construct five-dimensional <0, 0, 0, 0, 0>
v[1] = 23                      # <0, 23, 0, 0, 0> (based on use of __setitem__)
v[-1] = 45                     # <0, 23, 0, 0, 45> (also via __setitem__)
print(v[4])                    # print 45 (via __getitem__)
u = v + v                      # <0, 46, 0, 0, 90> (via __add__)
print(u)                       # print <0, 46, 0, 0, 90>
total = 0
for entry in v:                # implicit iteration via __len__ and __getitem__
  total += entry
```

We implement many of the behaviors by trivially invoking a similar behavior on the underlying list of coordinates. However, our implementation of __add__ is customized. Assuming the two operands are vectors with the same length, this method creates a new vector and sets the coordinates of the new vector to be equal to the respective sum of the operands' elements.

It is interesting to note that the class definition, as given in Code Fragment 2.4, automatically supports the syntax u = v + [5, 3, 10, −2, 1], resulting in a new vector that is the element-by-element "sum" of the first vector and the list instance. This is a result of Python's ***polymorphism***. Literally, "polymorphism" means "many forms." Although it is tempting to think of the other parameter of our __add__ method as another Vector instance, we never declared it as such. Within the body, the only behaviors we rely on for parameter other is that it supports len(other) and access to other[j]. Therefore, our code executes when the right-hand operand is a list of numbers (with matching length).

```
1  class Vector:
2    """Represent a vector in a multidimensional space."""
3
4    def __init__(self, d):
5      """Create d-dimensional vector of zeros."""
6      self._coords = [0] * d
7
8    def __len__(self):
9      """Return the dimension of the vector."""
10     return len(self._coords)
11
12   def __getitem__(self, j):
13     """Return jth coordinate of vector."""
14     return self._coords[j]
15
16   def __setitem__(self, j, val):
17     """Set jth coordinate of vector to given value."""
18     self._coords[j] = val
19
20   def __add__(self, other):
21     """Return sum of two vectors."""
22     if len(self) != len(other):              # relies on __len__ method
23       raise ValueError('dimensions must agree')
24     result = Vector(len(self))               # start with vector of zeros
25     for j in range(len(self)):
26       result[j] = self[j] + other[j]
27     return result
28
29   def __eq__(self, other):
30     """Return True if vector has same coordinates as other."""
31     return self._coords == other._coords
32
33   def __ne__(self, other):
34     """Return True if vector differs from other."""
35     return not self == other                 # rely on existing __eq__ definition
36
37   def __str__(self):
38     """Produce string representation of vector."""
39     return '<' + str(self._coords)[1:-1] + '>'   # adapt list representation
```

Code Fragment 2.4: Definition of a simple Vector class.

2.3.4 Iterators

Iteration is an important concept in the design of data structures. We introduced Python's mechanism for iteration in Section 1.8. In short, an ***iterator*** for a collection provides one key behavior: It supports a special method named `__next__` that returns the next element of the collection, if any, or raises a `StopIteration` exception to indicate that there are no further elements.

Fortunately, it is rare to have to directly implement an iterator class. Our preferred approach is the use of the ***generator*** syntax (also described in Section 1.8), which automatically produces an iterator of yielded values.

Python also helps by providing an automatic iterator implementation for any class that defines both `__len__` and `__getitem__`. To provide an instructive example of a low-level iterator, Code Fragment 2.5 demonstrates just such an iterator class that works on any collection that supports both `__len__` and `__getitem__`. This class can be instantiated as `SequenceIterator(data)`. It operates by keeping an internal reference to the data sequence, as well as a current index into the sequence. Each time `__next__` is called, the index is incremented, until reaching the end of the sequence.

```
1  class SequenceIterator:
2    """An iterator for any of Python's sequence types."""
3
4    def __init__(self, sequence):
5      """Create an iterator for the given sequence."""
6      self._seq = sequence              # keep a reference to the underlying data
7      self._k = -1                      # will increment to 0 on first call to next
8
9    def __next__(self):
10     """Return the next element, or else raise StopIteration error."""
11     self._k += 1                      # advance to next index
12     if self._k < len(self._seq):
13       return(self._seq[self._k])      # return the data element
14     else:
15       raise StopIteration( )          # there are no more elements
16
17   def __iter__(self):
18     """By convention, an iterator must return itself as an iterator."""
19     return self
```

Code Fragment 2.5: An iterator class for any sequence type.

2.3.5 Example: Range Class

As the final example for this section, we develop our own implementation of a class that mimics Python's built-in range class. Before introducing our class, we discuss the history of the built-in version. Prior to Python 3 being released, range was implemented as a function, and it returned a list instance with elements in the specified range. For example, range(2, 10, 2) returned the list [2, 4, 6, 8]. However, a typical use of the function was to support a for-loop syntax, such as **for** k **in** range(10000000). Unfortunately, this caused the instantiation and initialization of a list with the range of numbers. That was an unnecessarily expensive step, in terms of both time and memory usage.

The mechanism used to support ranges in Python 3 is entirely different (to be fair, the "new" behavior existed in Python 2 under the name xrange). It uses a strategy known as ***lazy evaluation***. Rather than creating a new list instance, range is a class that can effectively represent the desired range of elements without ever storing them explicitly in memory. To better explore the built-in range class, we recommend that you create an instance as r = range(8, 140, 5). The result is a relatively lightweight object, an instance of the range class, that has only a few behaviors. The syntax len(r) will report the number of elements that are in the given range (27, in our example). A range also supports the __getitem__ method, so that syntax r[15] reports the sixteenth element in the range (as r[0] is the first element). Because the class supports both __len__ and __getitem__, it inherits automatic support for iteration (see Section 2.3.4), which is why it is possible to execute a for loop over a range.

At this point, we are ready to demonstrate our own version of such a class. Code Fragment 2.6 provides a class we name Range (so as to clearly differentiate it from built-in range). The biggest challenge in the implementation is properly computing the number of elements that belong in the range, given the parameters sent by the caller when constructing a range. By computing that value in the constructor, and storing it as self._length, it becomes trivial to return it from the __len__ method. To properly implement a call to __getitem__(k), we simply take the starting value of the range plus k times the step size (i.e., for k=0, we return the start value). There are a few subtleties worth examining in the code:

- To properly support optional parameters, we rely on the technique described on page 27, when discussing a functional version of range.
- We compute the number of elements in the range as
  max(0, (stop − start + step − 1) // step)
  It is worth testing this formula for both positive and negative step sizes.
- The __getitem__ method properly supports negative indices by converting an index −k to len(self)−k before computing the result.

```
1  class Range:
2    """A class that mimic's the built-in range class."""
3
4    def __init__(self, start, stop=None, step=1):
5      """Initialize a Range instance.
6
7      Semantics is similar to built-in range class.
8      """
9      if step == 0:
10       raise ValueError('step cannot be 0')
11
12     if stop is None:                    # special case of range(n)
13       start, stop = 0, start            # should be treated as if range(0,n)
14
15     # calculate the effective length once
16     self._length = max(0, (stop - start + step - 1) // step)
17
18     # need knowledge of start and step (but not stop) to support __getitem__
19     self._start = start
20     self._step = step
21
22   def __len__(self):
23     """Return number of entries in the range."""
24     return self._length
25
26   def __getitem__(self, k):
27     """Return entry at index k (using standard interpretation if negative)."""
28     if k < 0:
29       k += len(self)                    # attempt to convert negative index
30
31     if not 0 <= k < self._length:
32       raise IndexError('index out of range')
33
34     return self._start + k * self._step
```

Code Fragment 2.6: Our own implementation of a Range class.

2.4 Inheritance

A natural way to organize various structural components of a software package is in a ***hierarchical*** fashion, with similar abstract definitions grouped together in a level-by-level manner that goes from specific to more general as one traverses up the hierarchy. An example of such a hierarchy is shown in Figure 2.4. Using mathematical notations, the set of houses is a ***subset*** of the set of buildings, but a ***superset*** of the set of ranches. The correspondence between levels is often referred to as an ***"is a" relationship***, as a house is a building, and a ranch is a house.

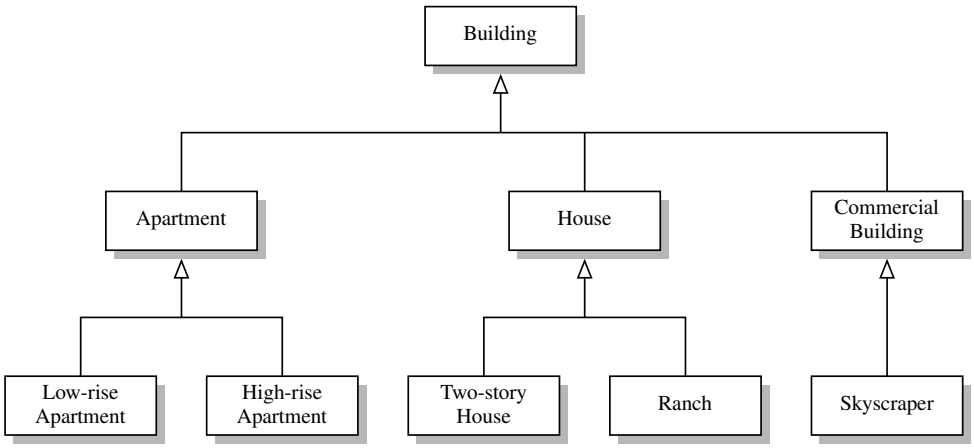


Figure 2.4: An example of an "is a" hierarchy involving architectural buildings.

A hierarchical design is useful in software development, as common functionality can be grouped at the most general level, thereby promoting reuse of code, while differentiated behaviors can be viewed as extensions of the general case, In object-oriented programming, the mechanism for a modular and hierarchical organization is a technique known as ***inheritance***. This allows a new class to be defined based upon an existing class as the starting point. In object-oriented terminology, the existing class is typically described as the ***base class***, ***parent class***, or ***superclass***, while the newly defined class is known as the ***subclass*** or ***child class***.

There are two ways in which a subclass can differentiate itself from its superclass. A subclass may ***specialize*** an existing behavior by providing a new implementation that ***overrides*** an existing method. A subclass may also ***extend*** its superclass by providing brand new methods.

Python's Exception Hierarchy

Another example of a rich inheritance hierarchy is the organization of various exception types in Python. We introduced many of those classes in Section 1.7, but did not discuss their relationship with each other. Figure 2.5 illustrates a (small) portion of that hierarchy. The BaseException class is the root of the entire hierarchy, while the more specific Exception class includes most of the error types that we have discussed. Programmers are welcome to define their own special exception classes to denote errors that may occur in the context of their application. Those user-defined exception types should be declared as subclasses of Exception.

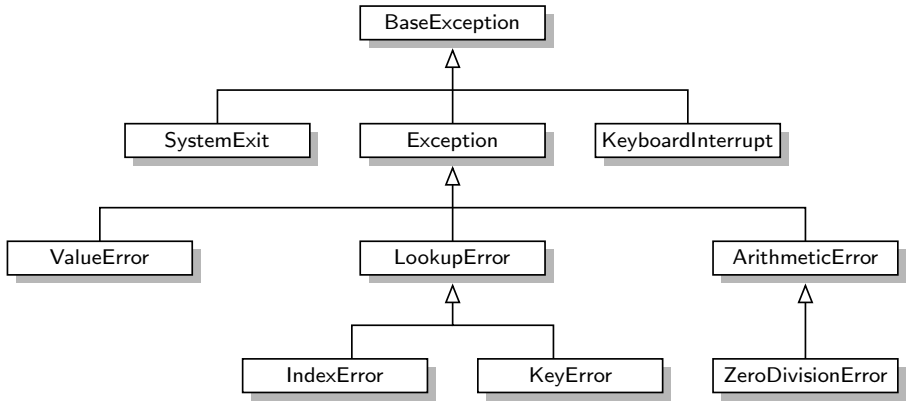


Figure 2.5: A portion of Python's hierarchy of exception types.

2.4.1 Extending the CreditCard Class

To demonstrate the mechanisms for inheritance in Python, we revisit the CreditCard class of Section 2.3, implementing a subclass that, for lack of a better name, we name PredatoryCreditCard. The new class will differ from the original in two ways: (1) if an attempted charge is rejected because it would have exceeded the credit limit, a $5 fee will be charged, and (2) there will be a mechanism for assessing a monthly interest charge on the outstanding balance, based upon an Annual Percentage Rate (APR) specified as a constructor parameter.

In accomplishing this goal, we demonstrate the techniques of specialization and extension. To charge a fee for an invalid charge attempt, we *override* the existing charge method, thereby specializing it to provide the new functionality (although the new version takes advantage of a call to the overridden version). To provide support for charging interest, we extend the class with a new method named process_month.

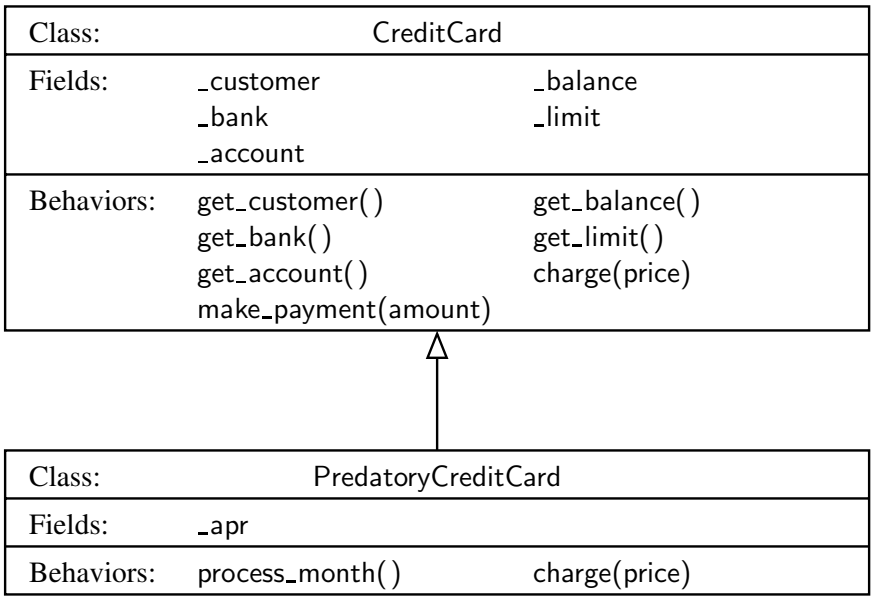


Figure 2.6: Diagram of an inheritance relationship.

Figure 2.6 provides an overview of our use of inheritance in designing the new PredatoryCreditCard class, and Code Fragment 2.7 gives a complete Python implementation of that class.

To indicate that the new class inherits from the existing CreditCard class, our definition begins with the syntax, **class** PredatoryCreditCard(CreditCard). The body of the new class provides three member functions: __init__, charge, and process_month. The __init__ constructor serves a very similar role to the original CreditCard constructor, except that for our new class, there is an extra parameter to specify the annual percentage rate. The body of our new constructor relies upon making a call to the inherited constructor to perform most of the initialization (in fact, everything other than the recording of the percentage rate). The mechanism for calling the inherited constructor relies on the syntax, **super**( ). Specifically, at line 15 the command

```
super( ).__init__(customer, bank, acnt, limit)
```

calls the __init__ method that was inherited from the CreditCard superclass. Note well that this method only accepts four parameters. We record the APR value in a new field named _apr.

In similar fashion, our PredatoryCreditCard class provides a new implementation of the charge method that overrides the inherited method. Yet, our implementation of the new method relies on a call to the inherited method, with syntax **super**( ).charge(price) at line 24. The return value of that call designates whether

```
class PredatoryCreditCard(CreditCard):
  """An extension to CreditCard that compounds interest and fees."""

  def __init__(self, customer, bank, acnt, limit, apr):
    """Create a new predatory credit card instance.

    The initial balance is zero.

    customer  the name of the customer (e.g., 'John Bowman')
    bank      the name of the bank (e.g., 'California Savings')
    acnt      the acount identifier (e.g., '5391 0375 9387 5309')
    limit     credit limit (measured in dollars)
    apr       annual percentage rate (e.g., 0.0825 for 8.25% APR)
    """
    super().__init__(customer, bank, acnt, limit)      # call super constructor
    self._apr = apr

  def charge(self, price):
    """Charge given price to the card, assuming sufficient credit limit.

    Return True if charge was processed.
    Return False and assess $5 fee if charge is denied.
    """
    success = super().charge(price)               # call inherited method
    if not success:
      self._balance += 5                          # assess penalty
    return success                                # caller expects return value

  def process_month(self):
    """Assess monthly interest on outstanding balance."""
    if self._balance > 0:
      # if positive balance, convert APR to monthly multiplicative factor
      monthly_factor = pow(1 + self._apr, 1/12)
      self._balance *= monthly_factor
```

Code Fragment 2.7: A subclass of CreditCard that assesses interest and fees.

the charge was successful. We examine that return value to decide whether to assess a fee, and in turn we return that value to the caller of method, so that the new version of charge has a similar outward interface as the original.

The process_month method is a new behavior, so there is no inherited version upon which to rely. In our model, this method should be invoked by the bank, once each month, to add new interest charges to the customer's balance. The most challenging aspect in implementing this method is making sure we have working knowledge of how an annual percentage rate translates to a monthly rate. We do not simply divide the annual rate by twelve to get a monthly rate (that would be too predatory, as it would result in a higher APR than advertised). The correct computation is to take the twelfth-root of 1 + self._apr, and use that as a multiplicative factor. For example, if the APR is 0.0825 (representing 8.25%), we compute $\sqrt[12]{1.0825} \approx 1.006628$, and therefore charge 0.6628% interest per month. In this way, each $100 of debt will amass $8.25 of compounded interest in a year.

Protected Members

Our PredatoryCreditCard subclass directly accesses the data member self._balance, which was established by the parent CreditCard class. The underscored name, by convention, suggests that this is a *nonpublic* member, so we might ask if it is okay that we access it in this fashion. While general users of the class should not be doing so, our subclass has a somewhat privileged relationship with the superclass. Several object-oriented languages (e.g., Java, C++) draw a distinction for nonpublic members, allowing declarations of ***protected*** or ***private*** access modes. Members that are declared as protected are accessible to subclasses, but not to the general public, while members that are declared as private are not accessible to either. In this respect, we are using _balance as if it were protected (but not private).

Python does not support formal access control, but names beginning with a single underscore are conventionally akin to protected, while names beginning with a double underscore (other than special methods) are akin to private. In choosing to use protected data, we have created a dependency in that our PredatoryCreditCard class might be compromised if the author of the CreditCard class were to change the internal design. Note that we could have relied upon the public get_balance( ) method to retrieve the current balance within the process_month method. But the current design of the CreditCard class does not afford an effective way for a subclass to change the balance, other than by direct manipulation of the data member. It may be tempting to use charge to add fees or interest to the balance. However, that method does not allow the balance to go above the customer's credit limit, even though a bank would presumably let interest compound beyond the credit limit, if warranted. If we were to redesign the original CreditCard class, we might add a nonpublic method, _set_balance, that could be used by subclasses to affect a change without directly accessing the data member _balance.

2.4.2 Hierarchy of Numeric Progressions

As a second example of the use of inheritance, we develop a hierarchy of classes for iterating numeric progressions. A numeric progression is a sequence of numbers, where each number depends on one or more of the previous numbers. For example, an ***arithmetic progression*** determines the next number by adding a fixed constant to the previous value, and a ***geometric progression*** determines the next number by multiplying the previous value by a fixed constant. In general, a progression requires a first value, and a way of identifying a new value based on one or more previous values.

To maximize reusability of code, we develop a hierarchy of classes stemming from a general base class that we name Progression (see Figure 2.7). Technically, the Progression class produces the progression of whole numbers: 0, 1, 2, ... However, this class is designed to serve as the base class for other progression types, providing as much common functionality as possible, and thereby minimizing the burden on the subclasses.

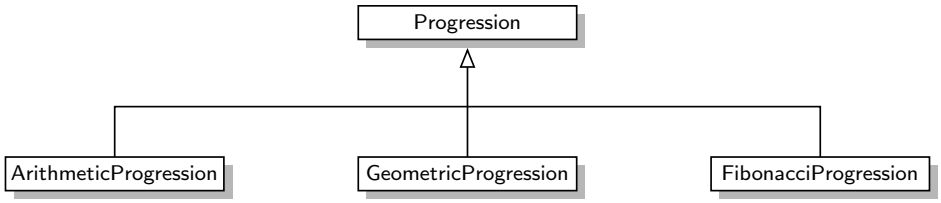


Figure 2.7: Our hierarchy of progression classes.

Our implementation of the basic Progression class is provided in Code Fragment 2.8. The constructor for this class accepts a starting value for the progression (0 by default), and initializes a data member, self._current, to that value.

The Progression class implements the conventions of a Python ***iterator*** (see Section 2.3.4), namely the special __next__ and __iter__ methods. If a user of the class creates a progression as seq = Progression( ), each call to next(seq) will return a subsequent element of the progression sequence. It would also be possible to use a for-loop syntax, **for** value **in** seq:, although we note that our default progression is defined as an infinite sequence.

To better separate the mechanics of the iterator convention from the core logic of advancing the progression, our framework relies on a nonpublic method named _advance to update the value of the self._current field. In the default implementation, _advance adds one to the current value, but our intent is that subclasses will override _advance to provide a different rule for computing the next entry.

For convenience, the Progression class also provides a utility method, named print_progression, that displays the next n values of the progression.

```
class Progression:
  """Iterator producing a generic progression.

  Default iterator produces the whole numbers 0, 1, 2, ...
  """

  def __init__(self, start=0):
    """Initialize current to the first value of the progression."""
    self._current = start

  def _advance(self):
    """Update self._current to a new value.

    This should be overridden by a subclass to customize progression.

    By convention, if current is set to None, this designates the
    end of a finite progression.
    """
    self._current += 1

  def __next__(self):
    """Return the next element, or else raise StopIteration error."""
    if self._current is None:        # our convention to end a progression
      raise StopIteration()
    else:
      answer = self._current         # record current value to return
      self._advance()                # advance to prepare for next time
      return answer                  # return the answer

  def __iter__(self):
    """By convention, an iterator must return itself as an iterator."""
    return self

  def print_progression(self, n):
    """Print next n values of the progression."""
    print(' '.join(str(next(self)) for j in range(n)))
```

Code Fragment 2.8: A general numeric progression class.

An Arithmetic Progression Class

Our first example of a specialized progression is an arithmetic progression. While the default progression increases its value by one in each step, an arithmetic progression adds a fixed constant to one term of the progression to produce the next. For example, using an increment of 4 for an arithmetic progression that starts at 0 results in the sequence $0, 4, 8, 12, \ldots$.

Code Fragment 2.9 presents our implementation of an ArithmeticProgression class, which relies on Progression as its base class. The constructor for this new class accepts both an increment value and a starting value as parameters, although default values for each are provided. By our convention, ArithmeticProgression(4) produces the sequence $0, 4, 8, 12, \ldots$, and ArithmeticProgression(4, 1) produces the sequence $1, 5, 9, 13, \ldots$.

The body of the ArithmeticProgression constructor calls the super constructor to initialize the _current data member to the desired start value. Then it directly establishes the new _increment data member for the arithmetic progression. The only remaining detail in our implementation is to override the _advance method so as to add the increment to the current value.

```
1  class ArithmeticProgression(Progression):          # inherit from Progression
2    """Iterator producing an arithmetic progression."""
3
4    def __init__(self, increment=1, start=0):
5      """Create a new arithmetic progression.
6
7      increment  the fixed constant to add to each term (default 1)
8      start      the first term of the progression (default 0)
9      """
10     super().__init__(start)                        # initialize base class
11     self._increment = increment
12
13   def _advance(self):                              # override inherited version
14     """Update current value by adding the fixed increment."""
15     self._current += self._increment
```

Code Fragment 2.9: A class that produces an arithmetic progression.

A Geometric Progression Class

Our second example of a specialized progression is a geometric progression, in which each value is produced by multiplying the preceding value by a fixed constant, known as the ***base*** of the geometric progression. The starting point of a geometric progression is traditionally 1, rather than 0, because multiplying 0 by any factor results in 0. As an example, a geometric progression with base 2 proceeds as $1, 2, 4, 8, 16, \ldots$.

Code Fragment 2.10 presents our implementation of a GeometricProgression class. The constructor uses 2 as a default base and 1 as a default starting value, but either of those can be varied using optional parameters.

```
1  class GeometricProgression(Progression):          # inherit from Progression
2    """Iterator producing a geometric progression."""
3
4    def __init__(self, base=2, start=1):
5      """Create a new geometric progression.
6
7      base        the fixed constant multiplied to each term (default 2)
8      start       the first term of the progression (default 1)
9      """
10     super().__init__(start)
11     self._base = base
12
13   def _advance(self):                             # override inherited version
14     """Update current value by multiplying it by the base value."""
15     self._current *= self._base
```

Code Fragment 2.10: A class that produces a geometric progression.

A Fibonacci Progression Class

As our final example, we demonstrate how to use our progression framework to produce a ***Fibonacci progression***. We originally discussed the Fibonacci series on page 41 in the context of generators. Each value of a Fibonacci series is the sum of the two most recent values. To begin the series, the first two values are conventionally 0 and 1, leading to the Fibonacci series $0, 1, 1, 2, 3, 5, 8, \ldots$. More generally, such a series can be generated from any two starting values. For example, if we start with values 4 and 6, the series proceeds as $4, 6, 10, 16, 26, 42, \ldots$.

```
1  class FibonacciProgression(Progression):
2    """Iterator producing a generalized Fibonacci progression."""
3
4    def __init__(self, first=0, second=1):
5      """Create a new fibonacci progression.
6
7      first        the first term of the progression (default 0)
8      second       the second term of the progression (default 1)
9      """
10     super().__init__(first)                  # start progression at first
11     self._prev = second - first              # fictitious value preceding the first
12
13   def _advance(self):
14     """Update current value by taking sum of previous two."""
15     self._prev, self._current = self._current, self._prev + self._current
```

Code Fragment 2.11: A class that produces a Fibonacci progression.

We use our progression framework to define a new FibonacciProgression class, as shown in Code Fragment 2.11. This class is markedly different from those for the arithmetic and geometric progressions because we cannot determine the next value of a Fibonacci series solely from the current one. We must maintain knowledge of the two most recent values. The base Progression class already provides storage of the most recent value as the _current data member. Our FibonacciProgression class introduces a new member, named _prev, to store the value that proceeded the current one.

With both previous values stored, the implementation of _advance is relatively straightforward. (We use a simultaneous assignment similar to that on page 45.) However, the question arises as to how to initialize the previous value in the constructor. The desired first and second values are provided as parameters to the constructor. The first should be stored as _current so that it becomes the first one that is reported. Looking ahead, once the first value is reported, we will do an assignment to set the new current value (which will be the second value reported), equal to the first value plus the "previous." By initializing the previous value to (second − first), the initial advancement will set the new current value to first + (second − first) = second, as desired.

Testing Our Progressions

To complete our presentation, Code Fragment 2.12 provides a unit test for all of our progression classes, and Code Fragment 2.13 shows the output of that test.

```
if __name__ == '__main__':
  print('Default progression:')
  Progression().print_progression(10)

  print('Arithmetic progression with increment 5:')
  ArithmeticProgression(5).print_progression(10)

  print('Arithmetic progression with increment 5 and start 2:')
  ArithmeticProgression(5, 2).print_progression(10)

  print('Geometric progression with default base:')
  GeometricProgression().print_progression(10)

  print('Geometric progression with base 3:')
  GeometricProgression(3).print_progression(10)

  print('Fibonacci progression with default start values:')
  FibonacciProgression().print_progression(10)

  print('Fibonacci progression with start values 4 and 6:')
  FibonacciProgression(4, 6).print_progression(10)
```

Code Fragment 2.12: Unit tests for our progression classes.

```
Default progression:
0 1 2 3 4 5 6 7 8 9
Arithmetic progression with increment 5:
0 5 10 15 20 25 30 35 40 45
Arithmetic progression with increment 5 and start 2:
2 7 12 17 22 27 32 37 42 47
Geometric progression with default base:
1 2 4 8 16 32 64 128 256 512
Geometric progression with base 3:
1 3 9 27 81 243 729 2187 6561 19683
Fibonacci progression with default start values:
0 1 1 2 3 5 8 13 21 34
Fibonacci progression with start values 4 and 6:
4 6 10 16 26 42 68 110 178 288
```

Code Fragment 2.13: Output of the unit tests from Code Fragment 2.12.

2.4.3 Abstract Base Classes

When defining a group of classes as part of an inheritance hierarchy, one technique for avoiding repetition of code is to design a base class with common functionality that can be inherited by other classes that need it. As an example, the hierarchy from Section 2.4.2 includes a Progression class, which serves as a base class for three distinct subclasses: ArithmeticProgression, GeometricProgression, and FibonacciProgression. Although it is possible to create an instance of the Progression base class, there is little value in doing so because its behavior is simply a special case of an ArithmeticProgression with increment 1. The real purpose of the Progression class was to centralize the implementations of behaviors that other progressions needed, thereby streamlining the code that is relegated to those subclasses.

In classic object-oriented terminology, we say a class is an ***abstract base class*** if its only purpose is to serve as a base class through inheritance. More formally, an abstract base class is one that cannot be directly instantiated, while a ***concrete class*** is one that can be instantiated. By this definition, our Progression class is technically concrete, although we essentially designed it as an abstract base class.

In statically typed languages such as Java and C++, an abstract base class serves as a formal type that may guarantee one or more ***abstract methods***. This provides support for polymorphism, as a variable may have an abstract base class as its declared type, even though it refers to an instance of a concrete subclass. Because there are no declared types in Python, this kind of polymorphism can be accomplished without the need for a unifying abstract base class. For this reason, there is not as strong a tradition of defining abstract base classes in Python, although Python's abc module provides support for defining a formal abstract base class.

Our reason for focusing on abstract base classes in our study of data structures is that Python's collections module provides several abstract base classes that assist when defining custom data structures that share a common interface with some of Python's built-in data structures. These rely on an object-oriented software design pattern known as the ***template method pattern***. The template method pattern is when an abstract base class provides concrete behaviors that rely upon calls to other abstract behaviors. In that way, as soon as a subclass provides definitions for the missing abstract behaviors, the inherited concrete behaviors are well defined.

As a tangible example, the collections.Sequence abstract base class defines behaviors common to Python's list, str, and tuple classes, as sequences that support element access via an integer index. More so, the collections.Sequence class provides concrete implementations of methods, count, index, and __contains__ that can be inherited by any class that provides concrete implementations of both __len__ and __getitem__. For the purpose of illustration, we provide a sample implementation of such a Sequence abstract base class in Code Fragment 2.14.

```
1  from abc import ABCMeta, abstractmethod          # need these definitions
2
3  class Sequence(metaclass=ABCMeta):
4    """Our own version of collections.Sequence abstract base class."""
5
6    @abstractmethod
7    def __len__(self):
8      """Return the length of the sequence."""
9
10   @abstractmethod
11   def __getitem__(self, j):
12     """Return the element at index j of the sequence."""
13
14   def __contains__(self, val):
15     """Return True if val found in the sequence; False otherwise."""
16     for j in range(len(self)):
17       if self[j] == val:                          # found match
18         return True
19     return False
20
21   def index(self, val):
22     """Return leftmost index at which val is found (or raise ValueError)."""
23     for j in range(len(self)):
24       if self[j] == val:                          # leftmost match
25         return j
26     raise ValueError('value not in sequence')    # never found a match
27
28   def count(self, val):
29     """Return the number of elements equal to given value."""
30     k = 0
31     for j in range(len(self)):
32       if self[j] == val:                          # found a match
33         k += 1
34     return k
```

Code Fragment 2.14: An abstract base class akin to collections.Sequence.

This implementation relies on two advanced Python techniques. The first is that we declare the ABCMeta class of the abc module as a ***metaclass*** of our Sequence class. A metaclass is different from a superclass, in that it provides a template for the class definition itself. Specifically, the ABCMeta declaration assures that the constructor for the class raises an error.

The second advanced technique is the use of the @abstractmethod decorator immediately before the __len__ and __getitem__ methods are declared. That declares these two particular methods to be abstract, meaning that we do not provide an implementation within our Sequence base class, but that we expect any concrete subclasses to support those two methods. Python enforces this expectation, by disallowing instantiation for any subclass that does not override the abstract methods with concrete implementations.

The rest of the Sequence class definition provides tangible implementations for other behaviors, under the assumption that the abstract __len__ and __getitem__ methods will exist in a concrete subclass. If you carefully examine the source code, the implementations of methods __contains__, index, and count do not rely on any assumption about the self instances, other than that syntax len(self) and self[j] are supported (by special methods __len__ and __getitem__, respectively). Support for iteration is automatic as well, as described in Section 2.3.4.

In the remainder of this book, we omit the formality of using the abc module. If we need an "abstract" base class, we simply document the expectation that subclasses provide assumed functionality, without technical declaration of the methods as abstract. But we will make use of the wonderful abstract base classes that are defined within the collections module (such as Sequence). To use such a class, we need only rely on standard inheritance techniques.

For example, our Range class, from Code Fragment 2.6 of Section 2.3.5, is an example of a class that supports the __len__ and __getitem__ methods. But that class does not support methods count or index. Had we originally declared it with Sequence as a superclass, then it would also inherit the count and index methods. The syntax for such a declaration would begin as:

```
class Range(collections.Sequence):
```

Finally, we emphasize that if a subclass provides its own implementation of an inherited behaviors from a base class, the new definition overrides the inherited one. This technique can be used when we have the ability to provide a more efficient implementation for a behavior than is achieved by the generic approach. As an example, the general implementation of __contains__ for a sequence is based on a loop used to search for the desired value. For our Range class, there is an opportunity for a more efficient determination of containment. For example, it is evident that the expression, 100000 **in** Range(0, 2000000, 100), should evaluate to True, even without examining the individual elements of the range, because the range starts with zero, has an increment of 100, and goes until 2 million; it must include 100000, as that is a multiple of 100 that is between the start and stop values. Exercise C-2.27 explores the goal of providing an implementation of Range.__contains__ that avoids the use of a (time-consuming) loop.

2.5 Namespaces and Object-Orientation

A ***namespace*** is an abstraction that manages all of the identifiers that are defined in a particular scope, mapping each name to its associated value. In Python, functions, classes, and modules are all first-class objects, and so the "value" associated with an identifier in a namespace may in fact be a function, class, or module.

In Section 1.10 we explored Python's use of namespaces to manage identifiers that are defined with global scope, versus those defined within the local scope of a function call. In this section, we discuss the important role of namespaces in Python's management of object-orientation.

2.5.1 Instance and Class Namespaces

We begin by exploring what is known as the ***instance namespace***, which manages attributes specific to an individual object. For example, each instance of our CreditCard class maintains a distinct balance, a distinct account number, a distinct credit limit, and so on (even though some instances may coincidentally have equivalent balances, or equivalent credit limits). Each credit card will have a dedicated instance namespace to manage such values.

There is a separate ***class namespace*** for each class that has been defined. This namespace is used to manage members that are to be *shared* by all instances of a class, or used without reference to any particular instance. For example, the make_payment method of the CreditCard class from Section 2.3 is not stored independently by each instance of that class. That member function is stored within the namespace of the CreditCard class. Based on our definition from Code Fragments 2.1 and 2.2, the CreditCard class namespace includes the functions: __init__, get_customer, get_bank, get_account, get_balance, get_limit, charge, and make_payment. Our PredatoryCreditCard class has its own namespace, containing the three methods we defined for that subclass: __init__, charge, and process_month.

Figure 2.8 provides a portrayal of three such namespaces: a class namespace containing methods of the CreditCard class, another class namespace with methods of the PredatoryCreditCard class, and finally a single instance namespace for a sample instance of the PredatoryCreditCard class. We note that there are two different definitions of a function named charge, one in the CreditCard class, and then the overriding method in the PredatoryCreditCard class. In similar fashion, there are two distinct __init__ implementations. However, process_month is a name that is only defined within the scope of the PredatoryCreditCard class. The instance namespace includes all data members for the instance (including the _apr member that is established by the PredatoryCreditCard constructor).

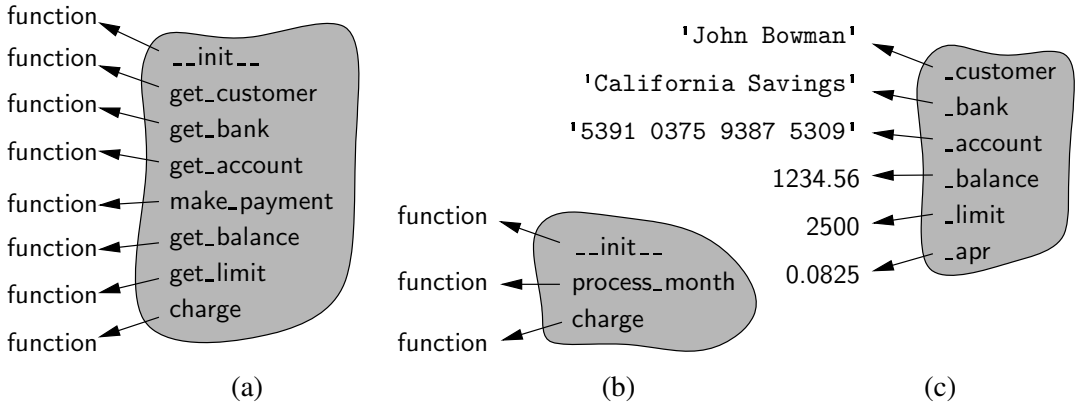


Figure 2.8: Conceptual view of three namespaces: (a) the class namespace for CreditCard; (b) the class namespace for PredatoryCreditCard; (c) the instance namespace for a PredatoryCreditCard object.

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.

Class Data Members

A class-level data member is often used when there is some value, such as a constant, that is to be shared by all instances of a class. In such a case, it would be unnecessarily wasteful to have each instance store that value in its instance namespace. As an example, we revisit the PredatoryCreditCard introduced in Section 2.4.1. That class assesses a $5 fee if an attempted charge is denied because of the credit limit. Our choice of $5 for the fee was somewhat arbitrary, and our coding style would be better if we used a named variable rather than embedding the literal value in our code. Often, the amount of such a fee is determined by the bank's policy and does not vary for each customer. In that case, we could define and use a class data member as follows:

```
class PredatoryCreditCard(CreditCard):
  OVERLIMIT_FEE = 5                    # this is a class-level member

  def charge(self, price):
    success = super( ).charge(price)
    if not success:
      self._balance += PredatoryCreditCard.OVERLIMIT_FEE
    return success
```

The data member, OVERLIMIT_FEE, is entered into the PredatoryCreditCard class namespace because that assignment takes place within the immediate scope of the class definition, and without any qualifying identifier.

Nested Classes

It is also possible to nest one class definition within the scope of another class. This is a useful construct, which we will exploit several times in this book in the implementation of data structures. This can be done by using a syntax such as

```
class A:          # the outer class
  class B:        # the nested class
    ...
```

In this case, class B is the nested class. The identifier B is entered into the namespace of class A associated with the newly defined class. We note that this technique is unrelated to the concept of inheritance, as class B does not inherit from class A.

Nesting one class in the scope of another makes clear that the nested class exists for support of the outer class. Furthermore, it can help reduce potential name conflicts, because it allows for a similarly named class to exist in another context. For example, we will later introduce a data structure known as a ***linked list*** and will define a nested node class to store the individual components of the list. We will also introduce a data structure known as a ***tree*** that depends upon its own nested

node class. These two structures rely on different node definitions, and by nesting those within the respective container classes, we avoid ambiguity.

Another advantage of one class being nested as a member of another is that it allows for a more advanced form of inheritance in which a subclass of the outer class overrides the definition of its nested class. We will make use of that technique in Section 11.2.1 when specializing the nodes of a tree structure.

Dictionaries and the __slots__ Declaration

By default, Python represents each namespace with an instance of the built-in dict class (see Section 1.2.3) that maps identifying names in that scope to the associated objects. While a dictionary structure supports relatively efficient name lookups, it requires additional memory usage beyond the raw data that it stores (we will explore the data structure used to implement dictionaries in Chapter 10).

Python provides a more direct mechanism for representing instance namespaces that avoids the use of an auxiliary dictionary. To use the streamlined representation for all instances of a class, that class definition must provide a class-level member named __slots__ that is assigned to a fixed sequence of strings that serve as names for instance variables. For example, with our CreditCard class, we would declare the following:

```
class CreditCard:
  __slots__ = '_customer', '_bank', '_account', '_balance', '_limit'
```

In this example, the right-hand side of the assignment is technically a tuple (see discussion of automatic packing of tuples in Section 1.9.3).

When inheritance is used, if the base class declares __slots__, a subclass must also declare __slots__ to avoid creation of instance dictionaries. The declaration in the subclass should only include names of supplemental methods that are newly introduced. For example, our PredatoryCreditCard declaration would include the following declaration:

```
class PredatoryCreditCard(CreditCard):
  __slots__ = '_apr'                    # in addition to the inherited members
```

We could choose to use the __slots__ declaration to streamline every class in this book. However, we do not do so because such rigor would be atypical for Python programs. With that said, there are a few classes in this book for which we expect to have a large number of instances, each representing a lightweight construct. For example, when discussing nested classes, we suggest linked lists and trees as data structures that are often comprised of a large number of individual nodes. To promote greater efficiency in memory usage, we will use an explicit __slots__ declaration in any nested classes for which we expect many instances.

2.5.2 Name Resolution and Dynamic Dispatch

In the previous section, we discussed various namespaces, and the mechanism for establishing entries in those namespaces. In this section, we examine the process that is used when *retrieving* a name in Python's object-oriented framework. When the dot operator syntax is used to access an existing member, such as obj.foo, the Python interpreter begins a name resolution process, described as follows:

1. The instance namespace is searched; if the desired name is found, its associated value is used.
2. Otherwise the class namespace, for the class to which the instance belongs, is searched; if the name is found, its associated value is used.
3. If the name was not found in the immediate class namespace, the search continues upward through the inheritance hierarchy, checking the class namespace for each ancestor (commonly by checking the superclass class, then its superclass class, and so on). The first time the name is found, its associate value is used.
4. If the name has still not been found, an AttributeError is raised.

As a tangible example, let us assume that mycard identifies an instance of the PredatoryCreditCard class. Consider the following possible usage patterns.

- mycard._balance (or equivalently, self._balance from within a method body): the _balance method is found within the *instance namespace* for mycard.
- mycard.process_month( ): the search begins in the instance namespace, but the name process_month is not found in that namespace. As a result, the PredatoryCreditCard class namespace is searched; in this case, the name is found and that method is called.
- mycard.make_payment(200): the search for the name, make_payment, fails in the instance namespace and in the PredatoryCreditCard namespace. The name is resolved in the namespace for superclass CreditCard and thus the inherited method is called.
- mycard.charge(50): the search for name charge fails in the instance namespace. The next namespace checked is for the PredatoryCreditCard class, because that is the true type of the instance. There is a definition for a charge function in that class, and so that is the one that is called.

In the last case shown, notice that the existence of a charge function in the PredatoryCreditCard class has the effect of ***overriding*** the version of that function that exists in the CreditCard namespace. In traditional object-oriented terminology, Python uses what is known as ***dynamic dispatch*** (or ***dynamic binding***) to determine, at run-time, which implementation of a function to call based upon the type of the object upon which it is invoked. This is in contrast to some languages that use ***static dispatching***, making a compile-time decision as to which version of a function to call, based upon the declared type of a variable.

2.6 Shallow and Deep Copying

In Chapter 1, we emphasized that an assignment statement foo = bar makes the name foo an ***alias*** for the object identified as bar. In this section, we consider the task of making a ***copy*** of an object, rather than an alias. This is necessary in applications when we want to subsequently modify either the original or the copy in an independent manner.

Consider a scenario in which we manage various lists of colors, with each color represented by an instance of a presumed color class. We let identifier warmtones denote an existing list of such colors (e.g., oranges, browns). In this application, we wish to create a new list named palette, which is a copy of the warmtones list. However, we want to subsequently be able to add additional colors to palette, or to modify or remove some of the existing colors, without affecting the contents of warmtones. If we were to execute the command

```
palette = warmtones
```

this creates an alias, as shown in Figure 2.9. No new list is created; instead, the new identifier palette references the original list.

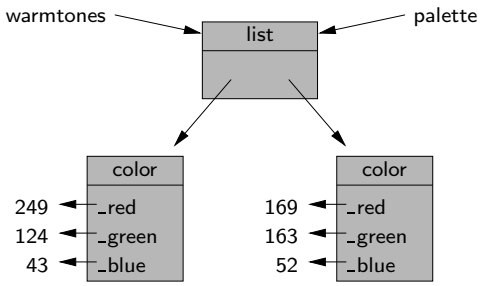


Figure 2.9: Two aliases for the same list of colors.

Unfortunately, this does not meet our desired criteria, because if we subsequently add or remove colors from "palette," we modify the list identified as warmtones.

We can instead create a new instance of the list class by using the syntax:

```
palette = list(warmtones)
```

In this case, we explicitly call the list constructor, sending the first list as a parameter. This causes a new list to be created, as shown in Figure 2.10; however, it is what is known as a ***shallow copy***. The new list is initialized so that its contents are precisely the same as the original sequence. However, Python's lists are ***referential*** (see page 9 of Section 1.2.3), and so the new list represents a sequence of references to the same elements as in the first.

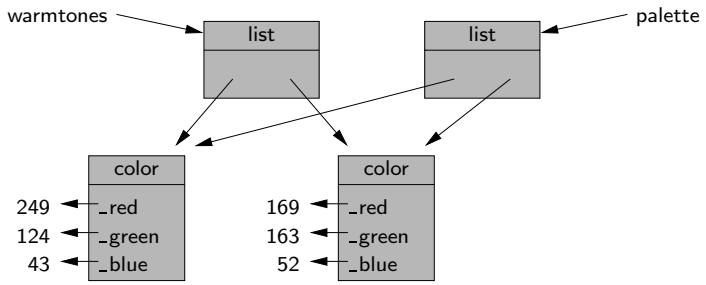


Figure 2.10: A shallow copy of a list of colors.

This is a better situation than our first attempt, as we can legitimately add or remove elements from palette without affecting warmtones. However, if we edit a color instance from the palette list, we effectively change the contents of warmtones. Although palette and warmtones are distinct lists, there remains indirect aliasing, for example, with palette[0] and warmtones[0] as aliases for the same color instance.

We prefer that palette be what is known as a ***deep copy*** of warmtones. In a deep copy, the new copy references its own *copies* of those objects referenced by the original version. (See Figure 2.11.)

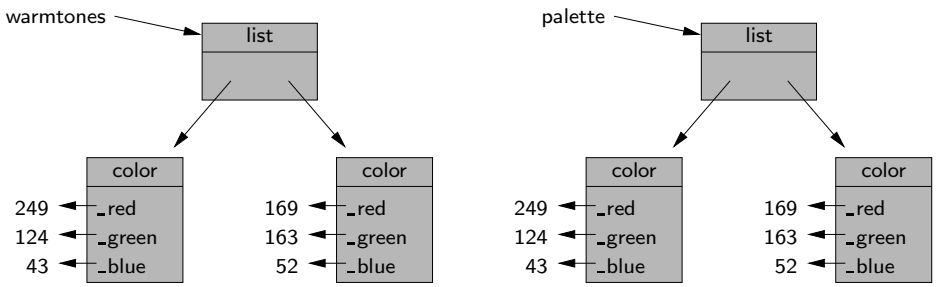


Figure 2.11: A deep copy of a list of colors.

Python's copy Module

To create a deep copy, we could populate our list by explicitly making copies of the original color instances, but this requires that we know how to make copies of colors (rather than aliasing). Python provides a very convenient module, named copy, that can produce both shallow copies and deep copies of arbitrary objects.

This module supports two functions: the copy function creates a shallow copy of its argument, and the deepcopy function creates a deep copy of its argument. After importing the module, we may create a deep copy for our example, as shown in Figure 2.11, using the command:

```
palette = copy.deepcopy(warmtones)
```

2.7 Exercises

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

Reinforcement

R-2.1 Give three examples of life-critical software applications.

R-2.2 Give an example of a software application in which adaptability can mean the difference between a prolonged lifetime of sales and bankruptcy.

R-2.3 Describe a component from a text-editor GUI and the methods that it encapsulates.

R-2.4 Write a Python class, Flower, that has three instance variables of type str, int, and float, that respectively represent the name of the flower, its number of petals, and its price. Your class must include a constructor method that initializes each variable to an appropriate value, and your class should include methods for setting the value of each type, and retrieving the value of each type.

R-2.5 Use the techniques of Section 1.7 to revise the charge and make_payment methods of the CreditCard class to ensure that the caller sends a number as a parameter.

R-2.6 If the parameter to the make_payment method of the CreditCard class were a negative number, that would have the effect of *raising* the balance on the account. Revise the implementation so that it raises a ValueError if a negative value is sent.

R-2.7 The CreditCard class of Section 2.3 initializes the balance of a new account to zero. Modify that class so that a new account can be given a nonzero balance using an optional fifth parameter to the constructor. The four-parameter constructor syntax should continue to produce an account with zero balance.

R-2.8 Modify the declaration of the first for loop in the CreditCard tests, from Code Fragment 2.3, so that it will eventually cause exactly one of the three credit cards to go over its credit limit. Which credit card is it?

R-2.9 Implement the __sub__ method for the Vector class of Section 2.3.3, so that the expression u−v returns a new vector instance representing the difference between two vectors.

R-2.10 Implement the __neg__ method for the Vector class of Section 2.3.3, so that the expression −v returns a new vector instance whose coordinates are all the negated values of the respective coordinates of v.

R-2.11 In Section 2.3.3, we note that our Vector class supports a syntax such as v = u + [5, 3, 10, −2, 1], in which the sum of a vector and list returns a new vector. However, the syntax v = [5, 3, 10, −2, 1] + u is illegal. Explain how the Vector class definition can be revised so that this syntax generates a new vector.

R-2.12 Implement the __mul__ method for the Vector class of Section 2.3.3, so that the expression v * 3 returns a new vector with coordinates that are 3 times the respective coordinates of v.

R-2.13 Exercise R-2.12 asks for an implementation of __mul__, for the Vector class of Section 2.3.3, to provide support for the syntax v * 3. Implement the __rmul__ method, to provide additional support for syntax 3 * v.

R-2.14 Implement the __mul__ method for the Vector class of Section 2.3.3, so that the expression u * v returns a scalar that represents the dot product of the vectors, that is, $\sum_{i=1}^{d} u_i \cdot v_i$.

R-2.15 The Vector class of Section 2.3.3 provides a constructor that takes an integer d, and produces a d-dimensional vector with all coordinates equal to 0. Another convenient form for creating a new vector would be to send the constructor a parameter that is some iterable type representing a sequence of numbers, and to create a vector with dimension equal to the length of that sequence and coordinates equal to the sequence values. For example, Vector([4, 7, 5]) would produce a three-dimensional vector with coordinates <4, 7, 5>. Modify the constructor so that either of these forms is acceptable; that is, if a single integer is sent, it produces a vector of that dimension with all zeros, but if a sequence of numbers is provided, it produces a vector with coordinates based on that sequence.

R-2.16 Our Range class, from Section 2.3.5, relies on the formula

$$\max(0, (\text{stop} - \text{start} + \text{step} - 1) \mathbin{//} \text{step})$$

to compute the number of elements in the range. It is not immediately evident why this formula provides the correct calculation, even if assuming a positive step size. Justify this formula, in your own words.

R-2.17 Draw a class inheritance diagram for the following set of classes:

- Class Goat extends object and adds an instance variable _tail and methods milk( ) and jump( ).
- Class Pig extends object and adds an instance variable _nose and methods eat(food) and wallow( ).
- Class Horse extends object and adds instance variables _height and _color, and methods run( ) and jump( ).
- Class Racer extends Horse and adds a method race( ).
- Class Equestrian extends Horse, adding an instance variable _weight and methods trot( ) and is_trained( ).

R-2.18 Give a short fragment of Python code that uses the progression classes from Section 2.4.2 to find the 8^{th} value of a Fibonacci progression that starts with 2 and 2 as its first two values.

R-2.19 When using the ArithmeticProgression class of Section 2.4.2 with an increment of 128 and a start of 0, how many calls to next can we make before we reach an integer of 2^{63} or larger?

R-2.20 What are some potential efficiency disadvantages of having very deep inheritance trees, that is, a large set of classes, A, B, C, and so on, such that B extends A, C extends B, D extends C, etc.?

R-2.21 What are some potential efficiency disadvantages of having very shallow inheritance trees, that is, a large set of classes, A, B, C, and so on, such that all of these classes extend a single class, Z?

R-2.22 The collections.Sequence abstract base class does not provide support for comparing two sequences to each other. Modify our Sequence class from Code Fragment 2.14 to include a definition for the __eq__ method, so that expression seq1 == seq2 will return True precisely when the two sequences are element by element equivalent.

R-2.23 In similar spirit to the previous problem, augment the Sequence class with method __lt__, to support lexicographic comparison seq1 < seq2.

Creativity

C-2.24 Suppose you are on the design team for a new e-book reader. What are the primary classes and methods that the Python software for your reader will need? You should include an inheritance diagram for this code, but you do not need to write any actual code. Your software architecture should at least include ways for customers to buy new books, view their list of purchased books, and read their purchased books.

C-2.25 Exercise R-2.12 uses the __mul__ method to support multiplying a Vector by a number, while Exercise R-2.14 uses the __mul__ method to support computing a dot product of two vectors. Give a single implementation of Vector.__mul__ that uses run-time type checking to support both syntaxes u * v and u * k, where u and v designate vector instances and k represents a number.

C-2.26 The SequenceIterator class of Section 2.3.4 provides what is known as a forward iterator. Implement a class named ReversedSequenceIterator that serves as a reverse iterator for any Python sequence type. The first call to next should return the last element of the sequence, the second call to next should return the second-to-last element, and so forth.

C-2.27 In Section 2.3.5, we note that our version of the Range class has implicit support for iteration, due to its explicit support of both __len__ and __getitem__. The class also receives implicit support of the Boolean test, "k **in** r" for Range r. This test is evaluated based on a forward iteration through the range, as evidenced by the relative quickness of the test 2 **in** Range(10000000) versus 9999999 **in** Range(10000000). Provide a more efficient implementation of the __contains__ method to determine whether a particular value lies within a given range. The running time of your method should be independent of the length of the range.

C-2.28 The PredatoryCreditCard class of Section 2.4.1 provides a process_month method that models the completion of a monthly cycle. Modify the class so that once a customer has made ten calls to charge in the current month, each additional call to that function results in an additional $1 surcharge.

C-2.29 Modify the PredatoryCreditCard class from Section 2.4.1 so that a customer is assigned a minimum monthly payment, as a percentage of the balance, and so that a late fee is assessed if the customer does not subsequently pay that minimum amount before the next monthly cycle.

C-2.30 At the close of Section 2.4.1, we suggest a model in which the CreditCard class supports a nonpublic method, _set_balance(b), that could be used by subclasses to affect a change to the balance, without directly accessing the _balance data member. Implement such a model, revising both the CreditCard and PredatoryCreditCard classes accordingly.

C-2.31 Write a Python class that extends the Progression class so that each value in the progression is the absolute value of the difference between the previous two values. You should include a constructor that accepts a pair of numbers as the first two values, using 2 and 200 as the defaults.

C-2.32 Write a Python class that extends the Progression class so that each value in the progression is the square root of the previous value. (Note that you can no longer represent each value with an integer.) Your constructor should accept an optional parameter specifying the start value, using $65,536$ as a default.

Projects

P-2.33 Write a Python program that inputs a polynomial in standard algebraic notation and outputs the first derivative of that polynomial.

P-2.34 Write a Python program that inputs a document and then outputs a bar-chart plot of the frequencies of each alphabet character that appears in that document.

P-2.35 Write a set of Python classes that can simulate an Internet application in which one party, Alice, is periodically creating a set of packets that she wants to send to Bob. An Internet process is continually checking if Alice has any packets to send, and if so, it delivers them to Bob's computer, and Bob is periodically checking if his computer has a packet from Alice, and, if so, he reads and deletes it.

P-2.36 Write a Python program to simulate an ecosystem containing two types of creatures, ***bears*** and ***fish***. The ecosystem consists of a river, which is modeled as a relatively large list. Each element of the list should be a Bear object, a Fish object, or None. In each time step, based on a random process, each animal either attempts to move into an adjacent list location or stay where it is. If two animals of the same type are about to collide in the same cell, then they stay where they are, but they create a new instance of that type of animal, which is placed in a random empty (i.e., previously None) location in the list. If a bear and a fish collide, however, then the fish dies (i.e., it disappears).

P-2.37 Write a simulator, as in the previous project, but add a Boolean gender field and a floating-point strength field to each animal, using an Animal class as a base class. If two animals of the same type try to collide, then they only create a new instance of that type of animal if they are of different genders. Otherwise, if two animals of the same type and gender try to collide, then only the one of larger strength survives.

P-2.38 Write a Python program that simulates a system that supports the functions of an e-book reader. You should include methods for users of your system to "buy" new books, view their list of purchased books, and read their purchased books. Your system should use actual books, which have expired copyrights and are available on the Internet, to populate your set of available books for users of your system to "purchase" and read.

P-2.39 Develop an inheritance hierarchy based upon a Polygon class that has abstract methods area( ) and perimeter( ). Implement classes Triangle, Quadrilateral, Pentagon, Hexagon, and Octagon that extend this base class, with the obvious meanings for the area( ) and perimeter( ) methods. Also implement classes, IsoscelesTriangle, EquilateralTriangle, Rectangle, and Square, that have the appropriate inheritance relationships. Finally, write a simple program that allows users to create polygons of the various types and input their geometric dimensions, and the program then outputs their area and perimeter. For extra effort, allow users to input polygons by specifying their vertex coordinates and be able to test if two such polygons are similar.

Chapter Notes

For a broad overview of developments in computer science and engineering, we refer the reader to *The Computer Science and Engineering Handbook* [96]. For more information about the Therac-25 incident, please see the paper by Leveson and Turner [69].

The reader interested in studying object-oriented programming further, is referred to the books by Booch [17], Budd [20], and Liskov and Guttag [71]. Liskov and Guttag also provide a nice discussion of abstract data types, as does the survey paper by Cardelli and Wegner [23] and the book chapter by Demurjian [33] in the *The Computer Science and Engineering Handbook* [96]. Design patterns are described in the book by Gamma *et al.* [41].

Books with specific focus on object-oriented programming in Python include those by Goldwasser and Letscher [43] at the introductory level, and by Phillips [83] at a more advanced level,