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).

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.

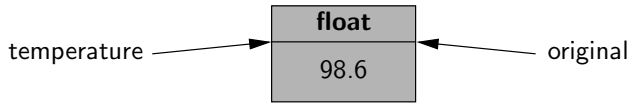

Figure 1.2: Identifiers temperature and original are aliases for the same object.

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.

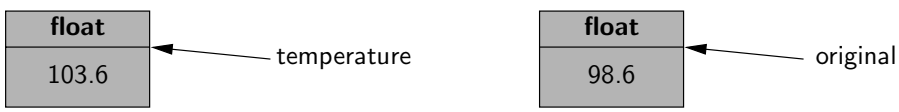

Figure 1.3: The temperature identifier has been assigned to a new value, while original continues to refer to the previously existing value.

5.4.2 Python's String Class

Strings are very important in Python. We introduced their use in Chapter 1, with a discussion of various operator syntaxes in Section 1.3. A comprehensive summary of the named methods of the class is given in Tables A.1 through A.4 of Appendix A. We will not formally analyze the efficiency of each of those behaviors in this section, but we do wish to comment on some notable issues. In general, we let n denote the length of a string. For operations that rely on a second string as a pattern, we let m denote the length of that pattern string.

The analysis for many behaviors is quite intuitive. For example, methods that produce a new string (e.g., capitalize, center, strip) require time that is linear in the length of the string that is produced. Many of the behaviors that test Boolean conditions of a string (e.g., islower) take $O(n)$ time, examining all n characters in the worst case, but short circuiting as soon as the answer becomes evident (e.g., islower can immediately return False if the first character is uppercased). The comparison operators (e.g., $==$, $<$) fall into this category as well.

Pattern Matching

Some of the most interesting behaviors, from an algorithmic point of view, are those that in some way depend upon finding a string pattern within a larger string; this goal is at the heart of methods such as __contains__, find, index, count, replace, and split. String algorithms will be the topic of Chapter 13, and this particular problem known as ***pattern matching*** will be the focus of Section 13.2. A naive implementation runs in $O(mn)$ time case, because we consider the $n-m+1$ possible starting indices for the pattern, and we spend $O(m)$ time at each starting position, checking if the pattern matches. However, in Section 13.2, we will develop an algorithm for finding a pattern of length m within a longer string of length n in $O(n)$ time.

Composing Strings

Finally, we wish to comment on several approaches for composing large strings. As an academic exercise, assume that we have a large string named document, and our goal is to produce a new string, letters, that contains only the alphabetic characters of the original string (e.g., with spaces, numbers, and punctuation removed). It may be tempting to compose a result through repeated concatenation, as follows.

```
# WARNING: do not do this
letters = ''                          # start with empty string
for c in document:
  if c.isalpha( ):
    letters += c                      # concatenate alphabetic character
```

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.

The FIFO strategy is quite simple to implement, as it only requires a queue Q to store references to the pages in the cache. Pages are enqueued in Q when they are referenced by a browser, and then are brought into the cache. When a page needs to be evicted, the computer simply performs a dequeue operation on Q to determine which page to evict. Thus, this policy also requires $O(1)$ additional work per page replacement. Also, the FIFO policy incurs no additional overhead for page requests. Moreover, it tries to take some advantage of temporal locality.

The LRU strategy goes a step further than the FIFO strategy, for the LRU strategy explicitly takes advantage of temporal locality as much as possible, by always evicting the page that was least-recently used. From a policy point of view, this is an excellent approach, but it is costly from an implementation point of view. That is, its way of optimizing temporal and spatial locality is fairly costly. Implementing the LRU strategy requires the use of an adaptable priority queue Q that supports updating the priority of existing pages. If Q is implemented with a sorted sequence based on a linked list, then the overhead for each page request and page replacement is $O(1)$. When we insert a page in Q or update its key, the page is assigned the highest key in Q and is placed at the end of the list, which can also be done in $O(1)$ time. Even though the LRU strategy has constant-time overhead, using the implementation above, the constant factors involved, in terms of the additional time overhead and the extra space for the priority queue Q, make this policy less attractive from a practical point of view.

Since these different page replacement policies have different trade-offs between implementation difficulty and the degree to which they seem to take advantage of localities, it is natural for us to ask for some kind of comparative analysis of these methods to see which one, if any, is the best.

From a worst-case point of view, the FIFO and LRU strategies have fairly unattractive competitive behavior. For example, suppose we have a cache containing m pages, and consider the FIFO and LRU methods for performing page replacement for a program that has a loop that repeatedly requests $m+1$ pages in a cyclic order. Both the FIFO and LRU policies perform badly on such a sequence of page requests, because they perform a page replacement on every page request. Thus, from a worst-case point of view, these policies are almost the worst we can imagine—they require a page replacement on every page request.

This worst-case analysis is a little too pessimistic, however, for it focuses on each protocol's behavior for one bad sequence of page requests. An ideal analysis would be to compare these methods over all possible page-request sequences. Of course, this is impossible to do exhaustively, but there have been a great number of experimental simulations done on page-request sequences derived from real programs. Based on these experimental comparisons, the LRU strategy has been shown to be usually superior to the FIFO strategy, which is usually better than the random strategy.

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.

The other methods discussed in Table A.4 serve a dual purpose to join, as they begin with a string and produce a sequence of substrings based upon a given delimiter. For example, the call `'red and green and blue'.split(' and ')` produces the result `['red', 'green', 'blue']`. If no delimiter (or None) is specified, split uses whitespace as a delimiter; thus, `'red and green and blue'.split()` produces `['red', 'and', 'green', 'and', 'blue']`.

String Formatting

The format method of the str class composes a string that includes one or more formatted arguments. The method is invoked with a syntax s.format(arg0, arg1, ...), where s serves as a ***formatting string*** that expresses the desired result with one or more placeholders in which the arguments will be substituted. As a simple example, the expression `'{} had a little {}'.format('Mary', 'lamb')` produces the result `'Mary had a little lamb'`. The pairs of curly braces in the formatting string are the placeholders for fields that will be substituted into the result. By default, the arguments sent to the function are substituted using positional order; hence, `'Mary'` was the first substitute and `'lamb'` the second. However, the substitution patterns may be explicitly numbered to alter the order, or to use a single argument in more than one location. For example, the expression `'{0}, {0}, {0} your {1}'.format('row', 'boat')` produces the result `'row, row, row your boat'`.

All substitution patterns allow use of annotations to pad an argument to a particular width, using a choice of fill character and justification mode. An example of such an annotation is `'{:-^20}'.format('hello')`. In this example, the hyphen (`-`) serves as a fill character, the caret (`^`) designates a desire for the string to be centered, and 20 is the desired width for the argument. This example results in the string `'-------hello--------'`. By default, space is used as a fill character and an implied $<$ character dictates left-justification; an explicit $>$ character would dictate right-justification.

There are additional formatting options for numeric types. A number will be padded with zeros rather than spaces if its width description is prefaced with a zero. For example, a date can be formatted in traditional "YYYY/MM/DD" form as `'{}/{:02}/{:02}'.format(year, month, day)`. Integers can be converted to binary, octal, or hexadecimal by respectively adding the character `b`, `o`, or `x` as a suffix to the annotation. The displayed precision of a floating-point number is specified with a decimal point and the subsequent number of desired digits. For example, the expression `'{:.3}'.format(2/3)` produces the string `'0.667'`, rounded to three digits after the decimal point. A programmer can explicitly designate use of fixed-point representation (e.g., `'0.667'`) by adding the character `f` as a suffix, or scientific notation (e.g., `'6.667e-01'`) by adding the character `e` as a suffix.

P-9.55 Write a program that can process a sequence of stock buy and sell orders as described in Exercise C-9.50.

P-9.56 Let S be a set of n points in the plane with distinct integer x- and y-coordinates. Let T be a complete binary tree storing the points from S at its external nodes, such that the points are ordered left to right by increasing x-coordinates. For each node v in T, let $S(v)$ denote the subset of S consisting of points stored in the subtree rooted at v. For the root r of T, define $top(r)$ to be the point in $S = S(r)$ with maximum y-coordinate. For every other node v, define $top(r)$ to be the point in S with highest y-coordinate in $S(v)$ that is not also the highest y-coordinate in $S(u)$, where u is the parent of v in T (if such a point exists). Such labeling turns T into a ***priority search tree***. Describe a linear-time algorithm for turning T into a priority search tree. Implement this approach.

P-9.57 One of the main applications of priority queues is in operating systems—for ***scheduling jobs*** on a CPU. In this project you are to build a program that schedules simulated CPU jobs. Your program should run in a loop, each iteration of which corresponds to a ***time slice*** for the CPU. Each job is assigned a priority, which is an integer between -20 (highest priority) and 19 (lowest priority), inclusive. From among all jobs waiting to be processed in a time slice, the CPU must work on a job with highest priority. In this simulation, each job will also come with a ***length*** value, which is an integer between 1 and 100, inclusive, indicating the number of time slices that are needed to process this job. For simplicity, you may assume jobs cannot be interrupted—once it is scheduled on the CPU, a job runs for a number of time slices equal to its length. Your simulator must output the name of the job running on the CPU in each time slice and must process a sequence of commands, one per time slice, each of which is of the form "add job *name* with length n and priority p" or "no new job this slice".

P-9.58 Develop a Python implementation of an adaptable priority queue that is based on an unsorted list and supports location-aware entries.

Chapter Notes

Knuth's book on sorting and searching [65] describes the motivation and history for the selection-sort, insertion-sort, and heap-sort algorithms. The heap-sort algorithm is due to Williams [103], and the linear-time heap construction algorithm is due to Floyd [39]. Additional algorithms and analyses for heaps and heap-sort variations can be found in papers by Bentley [15], Carlsson [24], Gonnet and Munro [45], McDiarmid and Reed [74], and Schaffer and Sedgewick [88].

Hash Codes

The first action that a hash function performs is to take an arbitrary key k in our map and compute an integer that is called the ***hash code*** for k; this integer need not be in the range $[0, N-1]$, and may even be negative. We desire that the set of hash codes assigned to our keys should avoid collisions as much as possible. For if the hash codes of our keys cause collisions, then there is no hope for our compression function to avoid them. In this subsection, we begin by discussing the theory of hash codes. Following that, we discuss practical implementations of hash codes in Python.

Treating the Bit Representation as an Integer

To begin, we note that, for any data type X that is represented using at most as many bits as our integer hash codes, we can simply take as a hash code for X an integer interpretation of its bits. For example, the hash code for key 314 could simply be 314. The hash code for a floating-point number such as 3.14 could be based upon an interpretation of the bits of the floating-point representation as an integer.

For a type whose bit representation is longer than a desired hash code, the above scheme is not immediately applicable. For example, Python relies on 32-bit hash codes. If a floating-point number uses a 64-bit representation, its bits cannot be viewed directly as a hash code. One possibility is to use only the high-order 32 bits (or the low-order 32 bits). This hash code, of course, ignores half of the information present in the original key, and if many of the keys in our map only differ in these bits, then they will collide using this simple hash code.

A better approach is to combine in some way the high-order and low-order portions of a 64-bit key to form a 32-bit hash code, which takes all the original bits into consideration. A simple implementation is to add the two components as 32-bit numbers (ignoring overflow), or to take the exclusive-or of the two components. These approaches of combining components can be extended to any object x whose binary representation can be viewed as an n-tuple $(x_0, x_1, \ldots, x_{n-1})$ of 32-bit integers, for example, by forming a hash code for x as $\sum_{i=0}^{n-1} x_i$, or as $x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}$, where the $\oplus$ symbol represents the bitwise exclusive-or operation (which is ^ in Python).

Polynomial Hash Codes

The summation and exclusive-or hash codes, described above, are not good choices for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{n-1})$, where the order of the x_i's is significant. For example, consider a 16-bit hash code for a character string s that sums the Unicode values of the characters in s. This hash code unfortunately produces lots of unwanted

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.

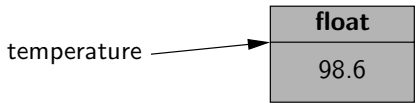

Figure 1.1: The identifier temperature references an instance of the float class having value 98.6.

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.