Page 48

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 speci■ed by explicitly assigning an actual parameter to a formal parameter by name. For example, with the above de■nition 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 de■nitions, 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.

Page 49

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 ■nding 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 ■rst-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 identi■ers x, y, z for arbitrary numeric types, k for an integer, and a, b, and c for arbitrary comparable types. We use the identi■er, 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.

Page 263

242 Chapter 6. Stacks, Queues, and Deques
Using an Array Circularly
In developing a more robust queue implementation, we allow the front of the queue to drift rightward, and we allow the contents of the queue to “wrap around” the end of an underlying array. We assume that our underlying array has ■xed length N that is greater that the actual number of elements in the queue. New elements are enqueued toward the “end” of the current queue, progressing from the front to index N – 1 and continuing at index 0, then 1. Figure 6.6 illustrates such a queue with ■rst element E and last element M.
I J K L M E F G H
0 1 2 f N –1
Figure 6.6: Modeling a queue with a circular array that wraps around the end.
Implementing this circular view is not dif■cult. When we dequeue an element and want to “advance” the front index, we use the arithmetic f = (f + 1) % N. Recall that the % operator in Python denotes the modulo operator, which is computed by taking the remainder after an integral division. For example, 14 divided by 3 has a quotient of 4 with remainder 2, that is, 14 3 = 4 3 . So in Python, 14 // 3 evaluates 2
to the quotient 4, while 14 % 3 evaluates to the remainder 2. The modulo operator is ideal for treating an array circularly. As a concrete example, if we have a list of length 10, and a front index 7, we can advance the front by formally computing (7+1) % 10, which is simply 8, as 8 divided by 10 is 0 with a remainder of 8. Similarly, advancing index 8 results in index 9. But when we advance from index 9 (the last one in the array), we compute (9+1) % 10, which evaluates to index 0 (as 10 divided by 10 has a remainder of zero).
A Python Queue Implementation
A complete implementation of a queue ADT using a Python list in circular fashion is presented in Code Fragments 6.6 and 6.7. Internally, the queue class maintains the following three instance variables:
data: is a reference to a list instance with a ■xed capacity.
size: is an integer representing the current number of elements stored in the queue (as opposed to the length of the data list).
front: is an integer that represents the index within data of the ■rst element of the queue (assuming the queue is not empty).
We initially reserve a list of moderate size for storing data, although the queue formally has size zero. As a technicality, we initialize the front index to zero. When front or dequeue are called with no elements in the queue, we raise an instance of the Empty exception, de■ned in Code Fragment 6.1 for our stack.

Page 120

node class. These two structures rely on different node de■nitions, 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 de■nition 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 ef■cient 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 de■nition must provide a class-level member named slots that is assigned to a ■xed 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 ef■ciency in memory usage, we will use an explicit slots declaration in any nested classes for which we expect many instances.

Page 144

3.3 Asymptotic Analysis

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n, taking a “big-picture” approach. For example, it is often enough just to know that the running time of an algorithm grows proportionally to n.

We analyze algorithms using a mathematical notation for functions that disregards constant factors. Namely, we characterize the running times of algorithms by using functions that map the size of the input, n, to values that correspond to the main factor that determines the growth rate in terms of n. This approach re■ects that each basic step in a pseudo-code description or a high-level language implementation may correspond to a small number of primitive operations. Thus, we can perform an analysis of an algorithm by estimating the number of primitive operations executed up to a constant factor, rather than getting bogged down in language-speci■c or hardware-speci■c analysis of the exact number of operations that execute on the computer.

As a tangible example, we revisit the goal of ■nding the largest element of a Python list; we ■rst used this example when introducing for loops on page 21 of Section 1.4.2. Code Fragment 3.1 presents a function named ■nd max for this task.

```
1 def ■nd max(data):
2 """Return the maximum element from a nonempty Python list."""
3 biggest = data[0] # The initial value to beat
4 for val in data: # For each value:
5 if val > biggest # if it is greater than the best so far,
6 biggest = val # we have found a new best (so far)
7 return biggest # When loop ends, biggest is the max
```

Code Fragment 3.1: A function that returns the maximum value of a Python list.

This is a classic example of an algorithm with a running time that grows proportional to n, as the loop executes once for each data element, with some ■xed number of primitive operations executing for each pass. In the remainder of this section, we provide a framework to formalize this claim.

3.3.1 The “Big-Oh” Notation

Let f (n) and g(n) be functions mapping positive integers to positive real numbers. We say that f (n) is O(g(n)) if there is a real constant c > 0 and an integer constant n0 ≥ 1 such that

f (n) ≤ cg(n), for n ≥ n0 .

This de■nition is often referred to as the “big-Oh” notation, for it is sometimes pronounced as “ f (n) is big-Oh of g(n).” Figure 3.5 illustrates the general de■nition.

Page 33

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 de■nes a speci■c 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 ■rst operand. This feature is useful when constructing Boolean expressions in which we ■rst 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 identi■ers a and b are aliases for the same object. The expression a == b tests a more general notion of equivalence. If identi■ers 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 identi■ers refer to

Page 427

406 Chapter 10. Maps, Hash Tables, and Skip Lists
10.1.3 Python's MutableMapping Abstract Base Class
Section 2.4.3 provides an introduction to the concept of an abstract base class and the role of such classes in Python's collections module. Methods that are declared to be abstract in such a base class must be implemented by concrete subclasses. However, an abstract base class may provide concrete implementation of other methods that depend upon use of the presumed abstract methods. (This is an example of the template method design pattern.)
The collections module provides two abstract base classes that are relevant to our current discussion: the Mapping and MutableMapping classes. The Mapping class includes all nonmutating methods supported by Python's dict class, while the MutableMapping class extends that to include the mutating methods. What we de■ne as the map ADT in Section 10.1.1 is akin to the MutableMapping abstract base class in Python's collections module.
The signi■cance of these abstract base classes is that they provide a framework to assist in creating a user-de■ned map class. In particular, the MutableMapping class provides concrete implementations for all behaviors other than the ■rst ■ve outlined in Section 10.1.1: getitem , setitem , delitem , len , and iter . As we implement the map abstraction with various data structures, as long as we provide the ■ve core behaviors, we can inherit all other derived behaviors by simply declaring MutableMapping as a parent class.
To better understand the MutableMapping class, we provide a few examples of how concrete behaviors can be derived from the ■ve core abstractions. For example, the contains method, supporting the syntax k in M, could be implemented by making a guarded attempt to retrieve self[k] to determine if the key exists.
def contains (self, k):
try:
self[k] # access via getitem (ignore result)
return True
except KeyError:
return False # attempt failed
A similar approach might be used to provide the logic of the setdefault method.
def setdefault(self, k, d):
try:
return self[k] # if getitem succeeds, return value
except KeyError: # otherwise:
self[k] = d # set default value with setitem
return d # and return that newly assigned value
We leave as exercises the implementations of the remaining concrete methods of the MutableMapping class.

Page 46

1.5. Functions 25
These assignment statements establish identi■er data as an alias for grades and target as a name for the string literal A . (See Figure 1.7.)
grades data target
list str
... A
Figure 1.7: A portrayal of parameter passing in Python, for the function call count(grades, A). Identi■ers data and target are formal parameters de■ned 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 identi■er prizes in the caller's scope is assigned to the object that is identi■ed 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 ef■cient, 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 identi■ed 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

Page 221

Using a ■xed increment for each resize, and thus an arithmetic progression of intermediate array sizes, results in an overall time that is quadratic in the number of operations, as shown in the following proposition. Intuitively, even an increase in 1000 cells per resize will become insigni■cant for large data sets.
Proposition 5.2: Performing a series of n append operations on an initially empty dynamic array using a ■xed increment with each resize takes Ω(n2) time.
Justi■cation: Let c > 0 represent the ■xed increment in capacity that is used for each resize event. During the series of n append operations, time will have been spent initializing arrays of size c, 2c, 3c, . . . , mc for m = n/c■, and therefore, the overall time would be proportional to c + 2c + 3c + · · · + mc. By Proposition 3.3, this sum is
c (c + 1)
n n
m m
m(m + 1) n2
∑ ci = c · ∑ i = c
2
≥ c
2
≥
2c
.
i=1 i=1
Therefore, performing the n append operations takes Ω(n2) time.
A lesson to be learned from Propositions 5.1 and 5.2 is that a subtle difference in an algorithm design can produce drastic differences in the asymptotic performance, and that a careful analysis can provide important insights into the design of a data structure.
Memory Usage and Shrinking an Array
Another consequence of the rule of a geometric increase in capacity when appending to a dynamic array is that the ■nal array size is guaranteed to be proportional to the overall number of elements. That is, the data structure uses O(n) memory. This is a very desirable property for a data structure.
If a container, such as a Python list, provides operations that cause the removal of one or more elements, greater care must be taken to ensure that a dynamic array guarantees O(n) memory usage. The risk is that repeated insertions may cause the underlying array to grow arbitrarily large, and that there will no longer be a proportional relationship between the actual number of elements and the array capacity after many elements are removed.

A robust implementation of such a data structure will shrink the underlying array, on occasion, while maintaining the O(1) amortized bound on individual operations. However, care must be taken to ensure that the structure cannot rapidly oscillate between growing and shrinking the underlying array, in which case the amortized bound would not be achieved. In Exercise C-5.16, we explore a strategy in which the array capacity is halved whenever the number of actual element falls below one fourth of that capacity, thereby guaranteeing that the array capacity is at most four times the number of elements; we explore the amortized analysis of such a strategy in Exercises C-5.17 and C-5.18.

Page 226

0 1 2 k n–1
Figure 5.16: Creating room to insert a new element at index k of a dynamic array.
that process depends upon the index of the new element, and thus the number of other elements that must be shifted. That loop copies the reference that had been at index n – 1 to index n, then the reference that had been at index n – 2 to n – 1, continuing until copying the reference that had been at index k to k + 1, as illustrated in Figure 5.16. Overall this leads to an amortized O(n – k + 1) performance for inserting at index k.
When exploring the ef■ciency of Python’s append method in Section 5.3.3, we performed an experiment that measured the average cost of repeated calls on varying sizes of lists (see Code Fragment 5.4 and Table 5.2). We have repeated that experiment with the insert method, trying three different access patterns:
• In the ■rst case, we repeatedly insert at the beginning of a list,
for n in range(N):
data.insert(0, None)
• In a second case, we repeatedly insert near the middle of a list,
for n in range(N):
data.insert(n // 2, None)
• In a third case, we repeatedly insert at the end of the list,
for n in range(N):
data.insert(n, None)
The results of our experiment are given in Table 5.5, reporting the average time per operation (not the total time for the entire loop). As expected, we see that inserting at the beginning of a list is most expensive, requiring linear time per operation. Inserting at the middle requires about half the time as inserting at the beginning, yet is still Ω(n) time. Inserting at the end displays O(1) behavior, akin to append.
N
100 1,000 10,000 100,000 1,000,000
k=0 0.482 0.765 4.014 36.643 351.590
k = n // 2 0.451 0.577 2.191 17.873 175.383
k=n 0.420 0.422 0.395 0.389 0.397
Table 5.5: Average running time of insert(k, val), measured in microseconds, as observed over a sequence of N calls, starting with an empty list. We let n denote the size of the current list (as opposed to the ■nal list).