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 syn-
tax 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 polymor-
phic 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 avail-
able 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 compar-
ison, 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 oper-
ate 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 23

1.1 Python Overview

Building data structures and algorithms requires that we communicate detailed in-
structions 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 sec-
ond 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 signi■cant in-
compatibilities between Python 2 and Python 3. This book is based on Python 3
(more speci■cally, 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 experi-
ence, although not necessarily using Python. This book does not provide a com-
plete 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 com-
mand, evaluates that command, and reports the result of the command. While the
interpreter can be used interactively (especially when debugging), a programmer
typically de■nes a series of commands in advance and saves those commands in a
plain text ■le known as source code or a script. For Python, source code is conven-
tionally stored in a ■le named with the .py suf■x (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 prede■ned script saved in a
■le (e.g., demo.py) are executed by invoking the interpreter with the ■lename as
an argument (e.g., python demo.py), or using an additional -i ■ag 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.

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 46

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 42

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 ■le (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 identi■er
```

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 identi■es 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 iden-ti■er, 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 identi■er. If the element of the original data happens to be mutable, the val identi■er can be used to invoke its methods. But a reassignment of identi■er 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 ■nding 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.

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).

6.1.1 The Stack Abstract Data Type

Stacks are the simplest of all data structures, yet they are also among the most important. They are used in a host of different applications, and as a tool for many more sophisticated data structures and algorithms. Formally, a stack is an abstract data type (ADT) such that an instance S supports the following two methods:

S.push(e): Add element e to the top of stack S.

S.pop( ): Remove and return the top element from the stack S; an error occurs if the stack is empty.

Additionally, let us de■ne the following accessor methods for convenience:

S.top( ): Return a reference to the top element of stack S, without removing it; an error occurs if the stack is empty.

S.is empty( ): Return True if stack S does not contain any elements.

len(S): Return the number of elements in stack S; in Python, we implement this with the special method len .

By convention, we assume that a newly created stack is empty, and that there is no a priori bound on the capacity of the stack. Elements added to the stack can have arbitrary type.

Example 6.3: The following table shows a series of stack operations and their effects on an initially empty stack S of integers.

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is empty( )	False	[5]
S.pop( )	5	[]
S.is empty( )	True	[]
S.pop( )	“error”	[]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

Page 33

1.3 Expressions, Operators, and Precedence

In the previous section, we demonstrated how names can be used to identify ex-
isting 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 seman-
tics 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 opera-
tors 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 75

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 ■nd 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, . . . , 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 ef■ciency at this point, however, as this topic is something that will be addressed later in this book.

Page 35

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 m n , 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 $0 \le r < m$. As a consequence, we ■nd 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 $m < r \le 0$. 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 ■oating-point operands, with the expression q = n // m being the integral ■oor 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 (pre■x unary operator)

& bitwise and

| bitwise or

ˆ bitwise exclusive-or

<< shift bits left, ■lling in with zeros

>> shift bits right, ■lling 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