Hash Codes in Python

The standard mechanism for computing hash codes in Python is a built-in function
with signature hash(x) that returns an integer value that serves as the hash code for
object x. However, only immutable data types are deemed hashable in Python. This
restriction is meant to ensure that a particular object·s hash code remains constant
during that object·s lifespan. This is an important property for an object·s use as
a key in a hash table. A problem could occur if a key were inserted into the hashtable, yet a later search were pe
Among Python·s built-in data types, the immutable int,·oat ,str,tuple ,a n d
frozenset classes produce robust hash codes, via the hash function, using tech-
niques similar to those discussed earlier in this section. Hash codes for characterstrings are well crafted based o
for tuples are computed with a similar technique based upon a combination of the
hash codes of the individual elements of the tuple. When hashing a frozenset ,t h e
order of the elements should be irrelevant, and so a natural option is to compute theexclusive-or of the individual
for an instance xof a mutable type, such as a list,aTypeError is raised.
Instances of user-de·ned classes are treated as unhashable by default, with a
TypeError raised by the hash function. However, a function that computes hash
codes can be implemented in the form of a special method named
hash
within
a class. The returned hash code should re·ect the immutable attributes of an in-stance. It is common to return a
three numeric red, green, and blue components might implement the method as:

def
hash
(self):
return hash((self.
red,self.
green, self.
blue)) # hash combined tuple

An important rule to obey is that if a class de·nes equivalence through
eq
,
then any implementation of
hash
must be consistent, in that if x= =y ,t h e n
hash(x) == hash(y) . This is important because if two instances are considered
to be equivalent and one is used as a key in a hash table, a search for the secondinstance should result in the di
true, it ensures that hash(5) andhash(5.0) are the same.

collisions for common groups of strings. In particular, "temp01" and"temp10"
collide using this function, as do "stop", "tops", "pots",a n d "spot". A better
hash code should somehow take into consideration the positions of the xi·s. An
alternative hash code, which does exactly this, is to choose a nonzero constant,
a/negationslash=1, and use as a hash code the value
x0an·1+x1an·2+···+xn·2a+xn·1.
Mathematically speaking, this is simply a polynomial in athat takes the compo-
nents (x0,x1,..., xn·1)of an object xas its coef·cients. This hash code is therefore
called a polynomial hash code . By Horner·s rule (see Exercise C-3.50), this poly-
nomial can be computed as
xn·1+a(xn·2+a(xn·3+···+a(x2+a(x1+ax0))···)).
Intuitively, a polynomial hash code uses multiplication by different powers as a
way to spread out the in·uence of each component across the resulting hash code.
Of course, on a typical computer, evaluating a polynomial will be done using
the ·nite bit representation for a hash code; hence, the value will periodically over-·ow the bits used for an intege
should be mindful that such over·ows are occurring and choose the constant aso
that it has some nonzero, low-order bits, which will serve to preserve some of theinformation content even as we
We have done some experimental studies that suggest that 33, 37, 39, and 41
are particularly good choices for awhen working with character strings that are
English words. In fact, in a list of over 50,000 English words formed as the unionof the word lists provided in two
37, 39, or 41 produced less than 7 collisions in each case!
Cyclic-Shift Hash Codes
A variant of the polynomial hash code replaces multiplication by awith a cyclic
shift of a partial sum by a certain number of bits. For example, a 5-bit cyclic shiftof the 32-bit value 00111
10110010110101010001010101000 is achieved by taking
the leftmost ·ve bits and placing those on the rightmost side of the representation,resulting in 101100101101010
. While this operation has little
natural meaning in terms of arithmetic, it accomplishes the goal of varying the bitsof the calculation. In Python, a
32-bit integers.

C-10.32 Perform experiments on our ChainHashMap andProbeHashMap classes
to measure its ef·ciency using random key sets and varying limits on the
load factor (see Exercise R-10.15).
C-10.33 Our implementation of separate chaining in ChainHashMap conserves
memory by representing empty buckets in the table as None , rather than
as empty instances of a secondary structure. Because many of these buck-ets will hold a single item, a better op
the table directly reference the
Item instance, and to reserve use of sec-
ondary containers for buckets that have two or more items. Modify our
implementation to provide this additional optimization.
C-10.34 Computing a hash code can be expensive, especially for lengthy keys. Inour hash table implementation
serting an item, and recompute each item·s hash code each time we resize
our table. Python·s dict class makes an interesting trade-off. The hash
code is computed once, when an item is inserted, and the hash code isstored as an extra ·eld of the item compo
C-10.35 Describe how to perform a removal from a hash table that uses linear
probing to resolve collisions where we do not use a special marker to
represent deleted elements. That is, we must rearrange the contents so thatit appears that the removed entry wa
C-10.36 The quadratic probing strategy has a clustering problem related to the wayit looks for open slots. Namel
checks buckets A[(h(k)+ i
2)mod N],f o r i=1,2,..., N·1.
a. Show that i2mod Nwill assume at most (N+1)/2 distinct values,
for Nprime, as iranges from 1 to N·1. As a part of this justi·ca-
tion, note that i2mod N=(N·i)2mod Nfor all i.
b. A better strategy is to choose a prime Nsuch that Nmod 4 =3a n d
then to check the buckets A[(h(k)±i2)mod N]asiranges from 1
to(N·1)/2, alternating between plus and minus. Show that this
alternate version is guaranteed to check every bucket in A.
C-10.37 Refactor our ProbeHashMap design so that the sequence of secondary
probes for collision resolution can be more easily customized. Demon-strate your new framework by providing se
C-10.38 Design a variation of binary search for performing the multimap opera-tion·nd
all(k) implemented with a sorted search table that includes du-
plicates, and show that it runs in time O(s+logn),w h e r e nis the number
of elements in the dictionary and sis the number of items with given key k.

Preface vii

Contents and Organization

The chapters for this book are organized to provide a pedagogical path that starts
with the basics of Python programming and object-oriented design. We then add
foundational techniques like algorithm analysis and recursion. In the main portionof the book, we present fundam

1.Python Primer
2.Object-Oriented Programming
3.Algorithm Analysis
4.Recursion
5.Array-Based Sequences
6.Stacks, Queues, and Deques
7.Linked Lists
8.Trees
9.Priority Queues
10.Maps, Hash Tables, and Skip Lists
11.Search Trees
12.Sorting and Selection
13.Text Processing
14.Graph Algorithms
15.Memory Management and B-Trees
A.Character Strings in Python
B.Useful Mathematical Facts

A more detailed table of contents follows this preface, beginning on page xi.

Prerequisites

We assume that the reader is at least vaguely familiar with a high-level program-ming language, such as C, C++

·Variables and expressions.
·Decision structures (such as if-statements and switch-statements).
·Iteration structures (for loops and while loops).
·Functions (whether stand-alone or object-oriented methods).

For readers who are familiar with these concepts, but not with how they are ex-pressed in Python, we provide a p
give a comprehensive treatment of Python.

```
def
 or
(self,o t h e r ) : # supports syntax S |T
···Return a new set that is the union of two existing sets.···
result = type( self)( ) # create new instance of concrete class
forein self :
result.add(e)
foreinother:
result.add(e)
return result
```

Code Fragment 10.15: An implementation of the MutableSet.
or
method,
which computes the union of two existing sets.

An implementation of the behavior that computes a new set as a union of two others is given in the form of the
or
special method, in Code Fragment 10.15.

An important subtlety in this implementation is the instantiation of the resultingset. Since the MutableSet class is
must belong to a concrete subclass. When computing the union of two such concrete instances, the result should presumably be an instance of the same class as the operands. The function type(self) returns a reference to the actual class of the instance identi·ed as self, and the subsequent parentheses in expression type(self)() call the default constructor for that class.

In terms of ef·ciency, we analyze such set operations while letting ndenote the size of Sand mdenote the size of set Tfor an operation such as S|T.I f the concrete sets are implemented with hashing, the expected running time of the implementation in Code Fragment 10.15 is O(m+n), because it loops over both sets, performing constant-time operations in the form of a containment check and a possible insertion into the result.

Our implementation of the in-place version of a union is given in Code Fragment 10.16, in the form of the
ior
special method that supports syntax S|=T.

Notice that in this case, we do not create a new set instance, instead we modify andreturn the existing set, after
in-place version of the union has expected running time O(m)where mis the size of the second set, because we only have to loop through that second set.

```
def
ior
(self,o t h e r ) : # supports syntax S |=T
···Modify this set to be the union of itself an another set.···
foreinother:
self.add(e)
return self # technical requirement of in-place operator
```

Code Fragment 10.16: An implementation of the MutableSet.
ior
method,
which performs an in-place union of one set with another.

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 additionalparameters using the followi

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

where a,b,a n d nare 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 unpredictablerandom sequences, this kind of 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 de·ning a Random class; instances of that class serve as generators with independent state. This allows different aspects of a program to rely on their ownpseudo-random number genera 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 ·oating-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.
shu·e(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.

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.
·oordiv
(b); alternatively b.
r·oordiv
(a)
a%b
a.
mod
(b); alternatively b.
rmod
(a)
a
b
a.
pow
(b); alternatively b.
rpow
(a)
a<<b
a.
lshift

another occurrence. The ef·ciency of the Boyer-Moore algorithm relies on creating a lookup table that quickly determines where a mismatched character occurs elsewhere in the pattern. In particular, we de·ne a function last (c)as

·Ifcis inP,last (c)is the index of the last (rightmost) occurrence of cinP. Otherwise, we conventionally de·ne last (c)=·1.

If we assume that the alphabet is of ·xed, ·nite size, and that characters can be converted to indices of an array (for example, by using their character code), the lastfunction can be easily implemented as a lookup table with worst-case O(1)-time access to the value last (c). However, the table would have length equal to the size of the alphabet (rather than the size of the pattern), and time would be required to initialize the entire table.

We prefer to use a hash table to represent the lastfunction, with only those characters from the pattern occurring in the structure. The space usage for thisapproach is proportional to the nu
the pattern, and thus O(m). The expected lookup time remains independent of the problem (although the worst-case bound is O(m)). Our complete implementation of the Boyer-Moore pattern-matching algorithm is given in Code Fragment 13.2.

```
def·nd
boyer
 moore(T, P):
···Return the lowest index of T at which substring P begins (or else -1).···
n, m = len(T), len(P) # introduce convenient notations
ifm= =0 : return 0 # trivial search for empty string
last = {} # build ·last· dictionary
forkinrange(m):
l a s t [P [ k ]]=k # later occurrence overwrites
# align end of pattern at index m-1 of text
i=m ·1 # an index into T
k=m ·1 # an index into P
while i<n:
ifT[i] == P[k]: # a matching character
ifk= =0 :
return i # pattern begins at index i of text
else:
i·=1 # examine previous character
k·=1 #o fb o t hTa n dP
else:
j = last.get(T[i], ·1) # last(T[i]) is -1 if not found
i+ =m ·min(k, j + 1) # case analysis for jump step
k=m ·1 # restart at end of pattern
return ·1
```

Code Fragment 13.2: An implementation of the Boyer-Moore algorithm.