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.

Hash Codes
The ·rst action that a hash function performs is to take an arbitrary key kin our
map and compute an integer that is called the hash code fork; 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 ofhash codes. Following that, we di
Treating the Bit Representation as an Integer
To begin, we note that, for any data type Xthat is represented using at most as many
bits as our integer hash codes, we can simply take as a hash code for Xan integer
interpretation of its bits. For example, the hash code for key 314 could simply be
314. The hash code for a ·oating-point number such as 3 .14 could be based upon
an interpretation of the bits of the ·oating-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 ·oating-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 informationpresent in the original key, an
A better approach is to combine in some way the high-order and low-order por-
tions of a 64-bit key to form a 32-bit hash code, which takes all the original bitsinto consideration. A simple imple
binary representation can be viewed as an n-tuple (x
0,x1,..., xn·1)of 32-bit inte-
gers, for example, by forming a hash code for xas·n·1
i=0xi,o ra s x0·x1····· xn·1,
where the ·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 choicesfor character strings or othe
0,x1,..., xn·1), where the order of the xi·s is signi·cant. For example,
consider a 16-bit hash code for a character string sthat sums the Unicode values
of the characters in s. This hash code unfortunately produces lots of unwanted

10.2.1 Hash Functions

The goal of a hash function ,h, is to map each key kto an integer in the range
[0,N·1],w h e r e Nis the capacity of the bucket array for a hash table. Equipped
with such a hash function, h, the main idea of this approach is to use the hash
function value, h(k), as an index into our bucket array, A, instead of the key k
(which may not be appropriate for direct use as an index). That is, we store the
item (k,v)in the bucket A[h(k)].
If there are two or more keys with the same hash value, then two different items
will be mapped to the same bucket in A. In this case, we say that a collision has
occurred. To be sure, there are ways of dealing with collisions, which we willdiscuss later, but the best strategy is
be fast and easy to compute.
It is common to view the evaluation of a hash function, h(k), as consisting of
two portions·a hash code that maps a key kto an integer, and a compression
function that maps the hash code to an integer within a range of indices, [0,N·1],
for a bucket array. (See Figure 10.5.)
-1hash code
120 -2... ...
compression function
120N - 1 ...Arbitrary Objects
Figure 10.5: Two parts of a hash function: a hash code and a compression function.
The advantage of separating the hash function into two such components is that
the hash code portion of that computation is independent of a speci·c hash table
size. This allows the development of a general hash code for each object that canbe used for a hash table of any
currently stored in the map. (See Section 10.2.3.)

Compression Functions
The hash code for a key kwill typically not be suitable for immediate use with a
bucket array, because the integer hash code may be negative or may exceed the ca-
pacity of the bucket array. Thus, once we have determined an integer hash code for
a key object k, there is still the issue of mapping that integer into the range [0,N·1].
This computation, known as a compression function , is the second action per-
formed as part of an overall hash function. A good compression function is one
that minimizes the number of collisions for a given set of distinct hash codes.
The Division Method
A simple compression function is the division method , which maps an integer ito
imod N,
where N, the size of the bucket array, is a ·xed positive integer. Additionally, if we
take Nto be a prime number, then this compression function helps ·spread out· the
distribution of hashed values. Indeed, if Nis not prime, then there is greater risk
that patterns in the distribution of hash codes will be repeated in the distribution ofhash values, thereby causing c
codes {200,205,210,215,220,..., 600}into a bucket array of size 100, then each
hash code will collide with three others. But if we use a bucket array of size 101,
then there will be no collisions. If a hash function is chosen well, it should ensurethat the probability of two differe
Choosing Nto be a prime number is not always enough, however, for if there is
a repeated pattern of hash codes of the form pN +qfor several different p·s, then
there will still be collisions.
The MAD Method
A more sophisticated compression function, which helps eliminate repeated pat-terns in a set of integer keys, is t
This method maps an integer ito
[(ai+b)mod p]mod N,
where Nis the size of the bucket array, pis a prime number larger than N,a n d a
andbare integers chosen at random from the interval [0,p·1], with a>0. This
compression function is chosen in order to eliminate repeated patterns in the set ofhash codes and get us closer
the same as we would have if these keys were ·thrown· into Auniformly at random.

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
101100101101010100010101000 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.

Operation	List	Hash Table	
		expected	worst case
getitem	O(n)	O(1)	O(n)
setitem	O(n)	O(1)	O(n)
delitem	O(n)	O(1)	O(n)
len	O(1)	O(1)	O(1)
iter	O(n)	O(n)	O(n)

Table 10.2: Comparison of the running times of the methods of a map realized by
means of an unsorted list (as in Section 10.1.5) or a hash table. We let ndenote
the number of items in the map, and we assume that the bucket array supporting
the hash table is maintained such that its capacity is proportional to the number ofitems in the map.
In practice, hash tables are among the most ef·cient means for implementing
a map, and it is essentially taken for granted by programmers that their core oper-
ations run in constant time. Python·s dictclass is implemented with hashing, and
the Python interpreter relies on dictionaries to retrieve an object that is referenced
by an identi·er in a given namespace. (See Sections 1.10 and 2.5.) The basic com-mand c=a+b involves two cal
getitem
in the dictionary for the local
namespace to retrieve the values identi·ed as aandb, and a call to
setitem
to store the result associated with name cin that namespace. In our own algorithm
analysis, we simply presume that such dictionary operations run in constant time,independent of the number of e
In a 2003 academic paper [31], researchers discuss the possibility of exploiting
a hash table·s worst-case performance to cause a denial-of-service (DoS) attackof Internet technologies. For ma
they note that an attacker could precompute a very large number of moderate-length
strings that all hash to the identical 32-bit hash code. (Recall that by any of thehashing schemes we describe, ot
In late 2011, another team of researchers demonstrated an implementation of
just such an attack [61]. Web servers allow a series of key-value parameters to beembedded in a URL using a sy
Typically, those key-value pairs are immediately stored in a map by the server,
and a limit is placed on the length and number of such parameters presuming that
storage time in the map will be linear in the number of entries. If all keys wereto collide, that storage requires qua

10.2.3 Load Factors, Rehashing, and E·ciency
In the hash table schemes described thus far, it is important that the load factor,
·=n/N, be kept below 1. With separate chaining, as ·gets very close to 1, the
probability of a collision greatly increases, which adds overhead to our operations,
since we must revert to linear-time list-based methods in buckets that have col-
lisions. Experiments and average-case analyses suggest that we should maintain·<0.9 for hash tables with sepa
With open addressing, on the other hand, as the load factor ·grows beyond 0 .5
and starts approaching 1, clusters of entries in the bucket array start to grow as well.These clusters cause the pr
gest that we should maintain ·<0.5 for an open addressing scheme with linear
probing, and perhaps only a bit higher for other open addressing schemes (for ex-ample, Python·s implementatio
If an insertion causes the load factor of a hash table to go above the speci·ed
threshold, then it is common to resize the table (to regain the speci·ed load factor)
and to reinsert all objects into this new table. Although we need not de·ne a new
hash code for each object, we do need to reapply a new compression function thattakes into consideration the si
scatter the items throughout the new bucket array. When rehashing to a new table, itis a good requirement for th
Indeed, if we always double the size of the table with each rehashing operation, then
we can amortize the cost of rehashing all the entries in the table against the timeused to insert them in the ·rst pl
E·ciency of Hash Tables
Although the details of the average-case analysis of hashing are beyond the scopeof this book, its probabilistic b
array. Thus, to store nentries, the expected number of keys in a bucket would
be·n/N·,w h i c hi s O(1)ifnisO(N).
The costs associated with a periodic rehashing, to resize a table after occasional
insertions or deletions can be accounted for separately, leading to an additional
O(1)amortized cost for
setitem
 and
 getitem
 .
In the worst case, a poor hash function could map every item to the same bucket.
This would result in linear-time performance for the core map operations with sepa-
rate chaining, or with any open addressing model in which the secondary sequenceof probes depends only on th
Table 10.2.

10.6 Exercises

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

Reinforcement

R-10.1 Give a concrete implementation of the popmethod in the context of the MutableMapping class, relying only on the ·ve primary abstract methods of that class.

R-10.2 Give a concrete implementation of the items() method in the context of theMutableMapping class, relying only on the ·ve primary abstract meth-ods of that class. What would its running time be if directly applied to the UnsortedTableMap subclass?

R-10.3 Give a concrete implementation of the items() method directly within the UnsortedTableMap class, ensuring that the entire iteration runs in O(n) time.

R-10.4 What is the worst-case running time for inserting nkey-value pairs into an initially empty map Mthat is implemented with the UnsortedTableMap class?

R-10.5 Reimplement the UnsortedTableMap class from Section 10.1.5, using the PositionalList class from Section 7.4 rather than a Python list.

R-10.6 Which of the hash table collision-handling schemes could tolerate a load factor above 1 and which could not?

R-10.7 OurPosition classes for lists and trees support the
eq
method so that
two distinct position instances are considered equivalent if they refer to thesame underlying node in a structure. I
hash
method that
is consistent with this notion of equivalence. Provide such a
hash
method.

R-10.8 What would be a good hash code for a vehicle identi·cation number thatis a string of numbers and letters

R-10.9 Draw the 11-entry hash table that results from using the hash function,h(i)=(3i+5)mod 11, to hash the key
16, and 5, assuming collisions are handled by chaining.

R-10.10 What is the result of the previous exercise, assuming collisions are han-dled by linear probing?

R-10.11 Show the result of Exercise R-10.9, assuming collisions are handled by quadratic probing, up to the point where the method fails.

10.2 Hash Tables
In this section, we introduce one of the most practical data structures for imple-
menting a map, and the one that is used by Python·s own implementation of the
dictclass. This structure is known as a hash table .
Intuitively, a map Msupports the abstraction of using keys as indices with a
syntax such as M[k] . As a mental warm-up, consider a restricted setting in which
a map with nitems uses keys that are known to be integers in a range from 0 to
N·1f o rs o m e N·n. In this case, we can represent the map using a lookup table
of length N, as diagrammed in Figure 10.3.
0 123456789 1 0
DZ C Q
Figure 10.3: A lookup table with length 11 for a map containing items (1,D), (3,Z),
(6,C), and (7,Q).
In this representation, we store the value associated with key kat index kof the
table (presuming that we have a distinct way to represent an empty slot). Basic mapoperations of
getitem

 ,
setitem
 ,a n d
 delitem
 can be implemented in
O(1)worst-case time.
There are two challenges in extending this framework to the more general set-
ting of a map. First, we may not wish to devote an array of length Nif it is the case
thatN/greatermuchn. Second, we do not in general require that a map·s keys be integers.
The novel concept for a hash table is the use of a hash function to map general
keys to corresponding indices in a table. Ideally, keys will be well distributed in therange from 0 to N·1 by a hash
distinct keys that get mapped to the same index. As a result, we will conceptualize
our table as a bucket array , as shown in Figure 10.4, in which each bucket may
manage a collection of items that are sent to a speci·c index by the hash function.
(To save space, an empty bucket may be replaced by None .)
0 123456789 1 0
(1,D) (25,C)
(3,F)
(14,Z)(39,C)(6,A) (7,Q)
Figure 10.4: A bucket array of capacity 11 with items (1,D), (25,C), (3,F), (14,Z),
(6,A), (39,C), and (7,Q), using a simple hash function.

10.2.2 Collision-Handling Schemes

The main idea of a hash table is to take a bucket array, A, and a hash function, h,a n d
use them to implement a map by storing each item (k,v)in the ·bucket· A[h(k)].
This simple idea is challenged, however, when we have two distinct keys, k1andk2,
such that h(k1)=h(k2). The existence of such collisions prevents us from simply
inserting a new item (k,v)directly into the bucket A[h(k)]. It also complicates our
procedure for performing insertion, search, and deletion operations.

Separate Chaining

A simple and ef·cient way for dealing with collisions is to have each bucket A[j]
store its own secondary container, holding items (k,v)such that h(k)=j. A natural
choice for the secondary container is a small map instance implemented using a list,
as described in Section 10.1.5. This collision resolution rule is known as separate
chaining , and is illustrated in Figure 10.6.

A123456789 1 0 01 112
123825
9054
28413618 10

Figure 10.6: A hash table of size 13, storing 10 items with integer keys, with colli-
sions resolved by separate chaining. The compression function is h(k)=kmod 13.
For simplicity, we do not show the values associated with the keys.

In the worst case, operations on an individual bucket take time proportional to
the size of the bucket. Assuming we use a good hash function to index the nitems
of our map in a bucket array of capacity N, the expected size of a bucket is n/N.
Therefore, if given a good hash function, the core map operations run in O(·n/N·).
The ratio ·=n/N, called the load factor of the hash table, should be bounded by
a small constant, preferably below 1. As long as ·isO(1), the core operations on
the hash table run in O(1)expected time.