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 performed for that key based on a different hash codethan that which it had when inserted; the wrong bucket would be searched.

Among Python’s built-in data types, the immutable int,float ,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 on a technique similar to polynomial hash codes,except using exclusive-or computations rather than additions. If we repeat the ex-periment described in Table 10.1 using Python’s built-in hash codes, we find thatonly 8 strings out of the set of more than 230,000 collide with another. Hash codes 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 hash codes without any shifting. If hash(x) is called for an instance xof a mutable type, such as a list,aTypeError is raised.

Instances of user-defined 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 reflect the immutable attributes of an in-stance. It is common to return a hash code that is itself based on the computed hashof the combination of such attributes. For example, a Color class that maintains

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 defines 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 discovery of the first. It is therefore important that thehash code for the second match the hash code for the first, so that the proper bucketis examined. This rule extends to any well-defined comparisons between objectsof different classes. For example, since Python treats the expression 5= =5 . 0 as

true, it ensures that hash(5) andhash(5.0) are the same.

Rezultat za stranicu 434

Hash Codes

The first 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 discuss practical implementations of hash codes inPython.

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 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 informationpresent in

the original key, and if many of the keys in our map only differ in thesebits, then they will collide using

this simple hash code.

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 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 xwhose

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 ˆin

Python).

Polynomial Hash Codes

The summation and exclusive-or hash codes, described above, are not good choicesfor

character strings or other variable-length objects that can be viewed as tuples ofthe form (x

0,x1,..., xn−1), where the order of the xi’s is significant. 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

Rezultat za stranicu 433

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 to try to avoid them in the first place. We saythat a hash function is “good” if it maps the keys in our map so as to sufficientlyminimize collisions. For practical reasons, we also would like a hash function to

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 specific hash table

size. This allows the development of a general hash code for each object that canbe used for a hash table of any size; only the compression function depends uponthe table size. This is particularly convenient, because the underlying bucket arrayfor a hash table may be dynamically resized, depending on the number of items

currently stored in the map. (See Section 10.2.3.)

Rezultat za stranicu 438

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 fixed 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 collisions. For example, if we insert keys with hash

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 different keys getting hashed to the same bucket is 1 /N.

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 the Multiply-Add-and-Divide (or “MAD”) method.

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 to having a “good” hash function, that is, one such thatthe probability any two different keys collide is 1 /N. This good behavior would be

the same as we would have if these keys were “thrown” into Auniformly at random.

Rezultat za stranicu 435

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

$x_0a^{n-1}+x_1a^{n-2}+\cdots+x_{n-2}a+x_{n-1}$.

Mathematically speaking, this is simply a polynomial in athat takes the compo-

nents $(x_0,x_1,\ldots,x_{n-1})$of an object xas its coefficients. 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

$x_{n-1}+a(x_{n-2}+a(x_{n-3}+\cdots+a(x_2+a(x_1+ax_0))\cdots))$.

Intuitively, a polynomial hash code uses multiplication by different powers as a

way to spread out the influence of each component across the resulting hash code.

Of course, on a typical computer, evaluating a polynomial will be done using

the finite bit representation for a hash code; hence, the value will periodically over-flow the bits used for an integer. Since we are more interested in a good spread ofthe object xwith respect to other keys, we simply ignore such overflows. Still, we should be mindful that such overflows 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 are in an overflow situation.

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 variants of Unix, we found that taking ato be 33,

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 five bits and placing those on the rightmost side of the representation,resulting in

10110010110101010001010100000111

. While this operation has little

natural meaning in terms of arithmetic, it accomplishes the goal of varying the bitsof the calculation. In Python, a cyclic shift of bits can be accomplished throughcareful use of the bitwise operators <<and>>,

taking care to truncate results to

32-bit integers.

Rezultat za stranicu 443

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 efficient 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 identifier in a given namespace. (See Sections 1.10 and 2.5.) The basic com-mand c=a+b involves two calls to

getitem

in the dictionary for the local namespace to retrieve the values identified 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 entries in the namespace. (Admittedly, the numberof entries in a typical namespace can almost surely be bounded by a constant.)

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 many published algorithms that compute hash codes, 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, other than double hashing, if two keys are mappedto the same hash code, they will be inseparable in the collision resolution.)

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 syntax such as ?key1=val1&key2=val2&key3=val3 .

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 quadratic time (causing the server to perform aninordinate amount of work). In spring of 2012, Python developers distributed asecurity patch that introduces randomization into the computation of hash codes

for strings, making it less tractable to reverse engineer a set of colliding strings.

Rezultat za stranicu 442

10.2.3 Load Factors, Rehashing, and Efficiency

In the hash table schemes described thus far, it is important that the load factor,

$\lambda = n/N$, be kept below 1. With separate chaining, as λgets 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$\lambda < 0.9$ for hash

tables with separate chaining.

With open addressing, on the other hand, as the load factor λgrows beyond 0 .5

and starts approaching 1, clusters of entries in the bucket array start to grow as well.These clusters cause the probing strategies to “bounce around” the bucket array fora considerable amount of time before they find an empty slot. In Exercise C-10.36,we explore the degradation of quadratic probing when $\lambda \geq 0.5$. Experiments sug-

gest that we should maintain $\lambda < 0.5$ for an open addressing scheme with linear

probing, and perhaps only a bit higher for other open addressing schemes (for ex-ample, Python’s implementation of open addressing enforces that $\lambda < 2/3$).

If an insertion causes the load factor of a hash table to go above the specified

threshold, then it is common to resize the table (to regain the specified load factor) and to reinsert all objects into this new table. Although we need not define a new

hash code for each object, we do need to reapply a new compression function thattakes into consideration the size of the new table. Each rehashing will generally

scatter the items throughout the new bucket array. When rehashing to a new table, itis a good requirement for the new array's size to be at least double the previous size.

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 first place (as with dynamic arrays; see Section 5.3).

Efficiency of Hash Tables

Although the details of the average-case analysis of hashing are beyond the scopeof this book, its probabilistic basis is quite intuitive. If our hash function is good,then we expect the entries to be uniformly distributed in the Ncells of the bucket

array. Thus, to store nentries, the expected number of keys in a bucket would

be$\lceil n/N \rceil$,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 the hash code. A summary of these costs is given in

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 five 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 five primary abstract methods 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. For positions to be allowed as keysin a hash table, there must be a definition for the

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 identification number thatis a string of numbers and letters of the form “9X9XX99X9XX999999,”where a “9” represents a digit and an “X” represents a letter?

R-10.9 Draw the 11-entry hash table that results from using the hash function,$h(i)=(3i+5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20,

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.

Rezultat za stranicu 432

410 Chapter 10. Maps, Hash Tables, and Skip Lists

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

$N-1$f o rs o m e $N \geq n$. 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 function, but in practice there may be two or more

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 specific 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.

Rezultat za stranicu 439

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 efficient 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 $h(k) = k \bmod 13$.

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 n items

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 $O(\lceil n/N \rceil)$.

The ratio $\lambda = n/N$, called the load factor of the hash table, should be bounded by

a small constant, preferably below 1. As long as λ is $O(1)$, the core operations on

the hash table run in $O(1)$ expected time.