Page

User

General tutorials on Python's dictionaries appear at http://www.developer.com/lang/other/article.php/630721 http://www.diveintopython.org/getting_to_know_python/dictionaries.html and elsewhere.

Why Lists Can't Be Dictionary Keys

Newcomers to Python often wonder why, while the language includes both a tuple and a list type, tuples are usable as a dictionary keys, while lists are not. This was a deliberate design decision, and can best be explained by first understanding how Python dictionaries work.

How Dictionaries Work

Dictionaries, in Python, are also known as "mappings", because they "map" or "associate" key objects to value objects:

Toggle line numbers

```python
# retrieve the value for a particular key
value = d[key]
```

Thus, Python mappings must be able to, given a particular key object, determine which (if any) value object is associated with a given key. One simple approach would be to store a list of (key, value) pairs, and then search the list sequentially every time a value was requested. However, this approach would be very slow with a large number of items - in complexity terms, this algorithm would be O(n), where n is the number of items in the mapping.

Python's dictionary implementation reduces the average complexity of dictionary lookups to O(1) by requiring that key objects provide a "hash" function. Such a hash function takes the information in a key object and uses it to produce an integer, called a hash value. This hash value is then used to determine which "bucket" this (key, value) pair should be placed into. Pseudocode for this lookup function might look something like:

Toggle line numbers

```
def lookup(d, key):
    '''dictionary lookup is done in three steps:
       1. A hash value of the key is computed using a hash function.

       2. The hash value addresses a location in d.data which is
          supposed to be an array of "buckets" or "collision lists"
          which contain the (key,value) pairs.

       3. The collision list addressed by the hash value is searched
          sequentially until a pair is found with pair[0] == key.
The
          return value of the lookup is then pair[1].
    '''
    h = hash(key)                  # step 1
    cl = d.data[h]                 # step 2
    for pair in cl:                # step 3
        if key == pair[0]:
            return pair[1]
    else:
        raise KeyError, "Key %s not found." % key
```

For such a lookup algorithm to work *correctly*, the hash functions provided must guarantee that if two keys produce different hash values then the two key objects are not equivalent, that is,

```
for all i1, i2, if hash(i1) != hash(i2), then i1 != i2
```

Otherwise, checking the hash value of a key object might make us look in the wrong bucket and thus never find the associated value.

For such a lookup algorithm to work *efficiently*, most buckets should have only a small number of items (preferably only one). Consider what would happen with the following hash function:

Toggle line numbers

```
def hash(obj):
    return 1
```

Note that this function meets the requirements of a hash function - every time two keys have different hash values, they represent different objects. (This is trivially true because *no* keys have different hash values - they all have the value 1.) But this is a bad hash function because it means that all (key, value) pairs will be placed in a single list, and so each lookup will require searching this entire list. Thus a (very) desirable property of a hash function is that if two keys produce the same hash values, then the key objects are equivalent, that is,

```
for all i1, i2, if hash(i1) == hash(i2), then i1 == i2
```

Hash functions that can approximate this property well will distribute (key, value) pairs evenly across the buckets, and keep lookup time down.

Types Usable as Dictionary Keys

The discussion above should explain why Python requires that:

To be used as a dictionary key, an object must support the hash function (e.g. through `__hash__`), equality comparison (e.g. through `__eq__` or `__cmp__`), and must satisfy the correctness condition above.

Lists as Dictionary Keys

That said, the simple answer to why lists cannot be used as dictionary keys is that lists do not provide a valid `__hash__` method. Of course, the obvious question is, "Why not?"

Consider what kinds of hash functions could be provided for lists.

If lists hashed by id, this would certainly be valid given Python's definition of a hash function -- lists with different hash values would have different ids. But lists are containers, and most other operations on them deal with them as such. So hashing lists by their id instead would produce unexpected behavior such as:

- Looking up different lists with the same contents would produce different results, even though *comparing* lists with the same contents would indicate them as equivalent.
- Using a list literal in a dictionary lookup would be pointless -- it would *always* produce a KeyError.

If lists were hashed by their contents (as tuples are), this, too, would be a valid hash function - lists with different hash values would have different contents. So, again, the problem is not in the definition of the hash function. But what should happen when a list, being used as a dictionary key, is modified? If the modification changed the hash value of the list (because it changed the contents of the list), then the list would be in the wrong "bucket" of the dictionary. This could end up with unexpected errors like:

Toggle line numbers

```
>>> l = [1, 2]
>>> d = {}
>>> d[l] = 42
>>> l.append(3)
>>> d[l]
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
KeyError: [1, 2, 3]
>>> d[[1, 2]]
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
KeyError: [1, 2]
```

where the value `42` is no longer available because the the list that hashes to the same value, `[1, 2]`, is not equivalent to the modified list, and the value that is equivalent to the modified list, `[1, 2, 3]` does not hash to the same value. Since the dictionary doesn't know when a key object is modified, such errors could only be produced at key lookup time, not at object modification time, which could make such errors quite hard to debug.

Having found that both ways of hashing lists have some undesirable side-effects, it should be more obvious why Python takes the stance that:

The builtin list type should not be used as a dictionary key.

Note that since tuples are immutable, they do not run into the troubles of lists - they can be hashed by their contents without worries about modification. Thus, in Python, they provide a valid `__hash__` method, and are thus usable as dictionary keys.

User Defined Types as Dictionary Keys

What about instances of user defined types?

By default, all user defined types are usable as dictionary keys with `hash(object)` defaulting to `id(object)`, and `cmp(object1, object2)` defaulting to `cmp(id(object1), id(object2))`. This same suggestion was discussed above for lists and found unsatisfactory. Why are user defined types different?

1. In the cases where an object must be placed in a mapping, object identity is often much more important than object contents.
2. In the cases where object content really is important, the default settings can be redefined by overridding `__hash__` and `__cmp__` or `__eq__`.

Note that it is often better practice, when an object is to be associated with a value, to simply assign that value as one of the object's attributes.

DictionaryKeys (last edited 2008-11-15 14:01:23 by localhost)

» MoinMoin Powered