Chapter_11_Dictionary

March 6, 2024

A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

A dictionary contains a collection of indices, which are called keys, and a collection of values. Each key is associated with a single value. The association of a key and a value is called a key-value pair or sometimes an item.

In mathematical language, a dictionary represents a mapping from keys to values, so you can also say that each key "maps to" a value. As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

[1]:
```
>>> eng2sp = dict()
>>> eng2sp
```

[1]:
```
{}
```

The squiggly-brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

[2]:
```
eng2sp['one'] = 'uno'
```

[3]:
```
eng2sp
```

[3]:
```
{'one': 'uno'}
```

[4]:
```
eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

[6]:
```
eng2sp
```

[6]:
```
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

The order of the key-value pairs might not be the same. If you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

[7]:
```
eng2sp['two']
```

[7]:
```
'dos'
```

The key 'two' always maps to the value 'dos' so the order of the items doesn't matter.

If the key isn't in the dictionary, you get an exception:

[8]:
```
eng2sp['four']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-8-d6070d49aedb> in <module>
----> 1 eng2sp['four']

KeyError: 'four'
```

[9]:
```
len(eng2sp)
```

[9]: 3

The in operator works on dictionaries, too; it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).

[10]:
```
'one' in eng2sp
```

[10]: True

[11]:
```
'uno' in eng2sp
```

[11]: False

To see whether something appears as a value in a dictionary, you can use the method values, which returns a collection of values, and then use the in operator:

[12]:
```
>>> vals = eng2sp.values()
>>> 'uno' in vals
```

[12]: True

[13]:
```
vals
```

[13]: dict_values(['uno', 'dos', 'tres'])

The in operator uses different algorithms for lists and dictionaries. For lists, it searches the elements of the list in order, as in Section 8.6. As the list gets longer, the search time gets longer in direct proportion.

Python dictionaries use a data structure called a hashtable that has a remarkable property: the in operator takes about the same amount of time no matter how many items are in the dictionary. I explain how that's possible in Section B.4, but the explanation might not make sense until you've read a few more chapters.

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An implementation is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

[15]:
```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

[16]:
```
>>> h = histogram('brontosaurus')
>>> h
```

[16]:
```
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

Dictionaries have a method called get that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. For example:

[17]:
```
>>> h = histogram('a')
>>> h
```

[17]:
```
{'a': 1}
```

[18]:
```
>>> h.get('a', 0)
```

[18]:
```
1
```

[19]:
```
h.get('c', 0)
```

[19]: 0

If you use a dictionary in a for statement, it traverses the keys of the dictionary. For example, print_hist prints each key and the corresponding value:

[20]:
```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

[21]:
```
>>> h = histogram('parrot')
>>> print_hist(h)
```

```
p 1
a 1
r 2
o 1
t 1
```

Again, the keys are in no particular order. To traverse the keys in sorted order, you can use the built-in function sorted:

[22]:
```
>>> for key in sorted(h):
    print(key, h[key])
```

```
a 1
o 1
p 1
r 2
t 1
```

Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a lookup.

But what if you have v and you want to find k? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a reverse lookup; you have to search.

[23]:
```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

The raise statement causes an exception; in this case it causes a LookupError, which is a built-in exception used to indicate that a lookup operation failed.

If we get to the end of the loop, that means v doesn't appear in the dictionary as a value, so we raise an exception.

[24]:
```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
```

[24]:
```
'r'
```

[25]:
```
key = reverse_lookup(h, 3)
```

```
---------------------------------------------------------------------------
LookupError                               Traceback (most recent call last)
<ipython-input-25-a9dd9a7a13ae> in <module>
----> 1 key = reverse_lookup(h, 3)

<ipython-input-23-624b98b92ec0> in reverse_lookup(d, v)
      3         if d[k] == v:
      4             return k
----> 5     raise LookupError()

LookupError:
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message. When you raise an exception, you can provide a detailed error message as an optional argument. For example:

[26]:
```
raise LookupError('value does not appear in the dictionary')
```

```
---------------------------------------------------------------------------
LookupError                               Traceback (most recent call last)
<ipython-input-26-809967c2546f> in <module>
----> 1 raise LookupError('value does not appear in the dictionary')

LookupError: value does not appear in the dictionary
```

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters. Here is a function that inverts a dictionary:

[28]:
```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
```

```
        else:
            inverse[val].append(key)
    return inverse
```

[29]:
```
hist = histogram('parrot')
>>> hist
```

[29]:
```
{'p': 1, 'a': 1, 'r': 2, 'o': 1, 't': 1}
```

[30]:
```
>>> inverse = invert_dict(hist)
>>> inverse
```

[30]:
```
{1: ['p', 'a', 'o', 't'], 2: ['r']}
```

Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

[31]:
```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-e3ea20954cd2> in <module>
      1 t = [1, 2, 3]
      2 d = dict()
----> 3 d[t] = 'oops'

TypeError: unhashable type: 'list'
```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be hashable.

A hash is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly

That's why keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use tuples, which we will see in the next chapter

Since dictionaries are mutable, they can't be used as keys, but they can be used as values.

Count how many times fibonacci(0) and fibonacci(1) are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a memo. Here is a "memoized" version of fibonacci:

[32]:
```
known = {0:0, 1:1}
def fibonacci(n):
    if n in known:
        return known[n]
    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

If you run this version of fibonacci and compare it with the original, you will find that it is much faster.

[33]:
```
fibonacci(10)
```

[33]:
```
55
```

[34]:
```
known
```

[34]:
```
{0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55}
```

In the previous example, known is created outside the function, so it belongs to the special frame called **main**. Variables in **main** are sometimes called global because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next.

It is common to use global variables for flags; that is, boolean variables that indicate ("flag") whether a condition is true. For example, some programs use a flag named verbose to control the level of detail in the output:

[35]:
```
verbose = True
def example1():
    if verbose:
        print('Running example1')
```

If you try to reassign a global variable, you might be surprised. The following example is supposed to keep track of whether the function has been called:

[36]:
```
been_called = False
def example2():
    been_called = True # WRONG
```

But if you run it you will see that the value of been_called doesn't change. The problem is that example2 creates a new local variable named been_called. The local variable goes away when the function ends, and has no effect on the global variable.

To reassign a global variable inside a function you have to declare the global variable before you use it:

[37]:
```
been_called = False
def example2():
    global been_called
    been_called = True
```

The global statement tells the interpreter something like, “In this function, when I say been_called, I mean the global variable; don’t create a local one.”

[38]:
```
count = 0
def example3():
    count = count + 1 # WRONG
```

[39]:
```
example3()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-39-bf3402d0be88> in <module>
----> 1 example3()

<ipython-input-38-06485f11fcd1> in example3()
      1 count = 0
      2 def example3():
----> 3     count = count + 1 # WRONG

UnboundLocalError: local variable 'count' referenced before assignment
```

[40]:
```
def example3():
    global count
    count += 1
```

[41]:
```
example3()
```

[42]:
```
count
```

[42]:
```
1
```

If a global variable refers to a mutable value, you can modify the value without declaring the variable:

[43]:
```
known = {0:0, 1:1}
def example4():
    known[2] = 1
```

[45]:
```
example4()
```

[46]:
```
known
```

[46]:
```
{0: 0, 1: 1, 2: 1}
```

So you can add, remove and replace elements of a global list or dictionary, but if you want to reassign the variable, you have to declare it:

[47]:
```
def example5():
    global known
    known = dict()
```

[ ]:
```

```