Chapter 11

Dictionaries

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.

You can think of a dictionary as a mapping between a set of indices and a set of values. Each index, which is called a **key**, corresponds to a value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

As an example, we will build a dictionary that maps from English words to Spanish words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items.

```
>>> eng2sp = dict()
>>> print eng2sp
{}
```

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

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key `'one'` to the value `'uno'`. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print eng2sp
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

But if you print eng2sp, you might be surprised:

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The key-value pairs are not in order, but that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print eng2sp['two']
'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:

```
>>> print eng2sp['four']
KeyError: 'four'
```

The len function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

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

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

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

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

The in operator uses different algorithms for lists and dictionaries. For lists, it uses a search algorithm, as in Section 9.3. As the list gets longer, the search time gets longer in direct proportion. For dictionaries, Python uses an algorithm called a **hashtable** that has a remarkable property: the in operator takes about the same amount of time no matter how many items there are in a dictionary. I won't explain how that's possible, but you can look it up.

11.1 Dictionary as a set of counters

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.

Here is what the code might look like:

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

The name of the function is **histogram**, which is a statistical term for a set of counters (or frequencies).

The first line of the function creates an empty dictionary. The `for` loop traverses the string. Each time through the loop, if the character `c` is not in the dictionary, we create a new item with key `c` and the initial value 1 (since we have seen this letter once). If `c` is already in the dictionary we increment `d[c]`.

Here's how it works:

```
>>> h = histogram('brontosaurus')
>>> print h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters `'a'` and `'b'` appear once each; `'o'` appears twice, and so on.

Exercise 11.1. *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:*

```
>>> h = histogram('a')
>>> print h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

Use `get` *to write* `histogram` *more concisely. You should be able to eliminate the* `if` *statement.*

11.2 Looping and dictionaries

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:

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

Here's what the output looks like:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Again, the keys are in no particular order.

Exercise 11.2. *Dictionaries have a method called* `keys` *that returns the keys of the dictionary, in no particular order, as a list.*

Modify `print_hist` *to print the keys and their values in alphabetical order, using* `keys` *and* `sort`.

11.3 Reverse lookup

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.

Here is a function that takes a value and returns the first key that maps to that value:

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

This function is yet another example of the search pattern we have seen before, but it uses a feature we haven't seen before, raise. The raise statement causes an exception; in this case it causes a ValueError, which generally indicates that there is something wrong with the value of a parameter.

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.

Here is an example of a successful reverse lookup:

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> print k
r
```

And an unsuccessful one:

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in reverse_lookup
ValueError
```

The result when you raise an exception is the same as when Python raises one: it prints a traceback and an error message.

The raise statement takes a detailed error message as an optional argument. For example:

```
>>> raise ValueError, 'value does not appear in the dictionary'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: 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.

Exercise 11.3. *Modify* reverse_lookup *so that it builds and returns a list of* all *keys that map to* v*, or an empty list if there are none.*

11.4 Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you were 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:

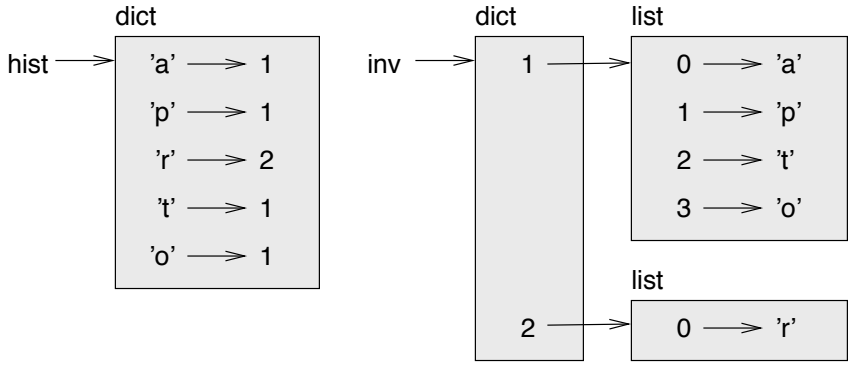
```
def invert_dict(d):
    inv = {}
    for key in d:
        val = d[key]
        if val not in inv:
            inv[val] = [key]
        else:
            inv[val].append(key)
    return inv
```

Each time through the loop, `key` gets a key from `d` and `val` gets the corresponding value. If `val` is not in `inv`, that means we haven't seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```
>>> hist = histogram('parrot')
>>> print hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inv = invert_dict(hist)
>>> print inv
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

And here is a diagram showing `hist` and `inv`:



A dictionary is represented as a box with the type `dict` above it and the key-value pairs inside. If the values are integers, floats or strings, I usually draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple.

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

```
>>> t = [1, 2, 3]
>>> d = {}
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

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 uses 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 the 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.

Exercise 11.4. *Read the documentation of the dictionary method* `setdefault` *and use it to write a more concise version of* `invert_dict`.

11.5 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

Scale down the input: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first `n` lines.

If there is an error, you can reduce `n` to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

Check summaries and types: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of run-time errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value, which is often smaller than the value itself.

Write self-checks: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of number, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a "sanity check" because it detects results that are "insane."

Another kind of check compares the results of two different computations to see if they are consistent. This is called a "consistency check."

11.6 Glossary

dictionary: A mapping from a set of keys to their corresponding values.

key-value pair: The representation of the mapping from a key to a value.

item: Another name for a key-value pair.

key: An object that appears in a dictionary as the first part of a key-value pair.

value: An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word "value."

implementation: A way of performing a computation.

hashtable: The algorithm used to implement Python dictionaries.

hash function: A function used by a hashtable to compute the location for a key.

hashable: A type that has a hash function. Immutable types like integers, floats and strings are hashable; mutable types like lists and dictionaries are not.

lookup: A dictionary operation that takes a key and finds the corresponding value.

reverse lookup: A dictionary operation that takes a value and finds one or more keys that map to it.

singleton: A list (or other sequence) with a single element.

call graph: A diagram that shows every frame created during the execution of a program, with an arrow from each caller to each callee.

histogram: A set of counters.

hint: A computed value stored to avoid unnecessary future computation.

global variable: A variable defined outside a function. Global variables can be accessed from any function.

11.7 Exercises

Exercise 11.5. *Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called* `is_anagram` *that takes two strings and returns True if they are anagrams.*

Exercise 11.6. *Write a function named* `has_duplicates` *that takes a list as a parameter and that returns* `True` *if there is any object that appears more than once in the list, and* `False` *otherwise.*

Exercise 11.7. *Write a function that takes the file name of a word list (see Section 19.1) as a parameter and searches for all pairs of words that are rotations of each other. The function* `rotate_word` *is described in Exercise 19.9.*