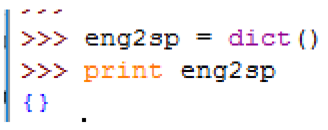
LAB 3:

Dictionaries:

A dictionary is like a list, but it is more general. General in a sense that in a list, indices have to be integers; in a dictionary they can be any type.

You can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

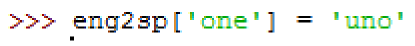
Example: dictionary that maps form English to Spanish



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

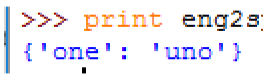
The function ***dict()*** creates a new dictionary data type with no items. Since ***dict*** is the built-in function, you should avoid using it as variable name. In above example the *eng2sp* variable is assigned with empty dictionary type. The curly bracket {} represents an empty-dictionary.

You can add items in the dictionary as shown below:



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

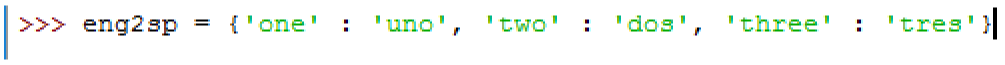
Here a single key-value pair is created which maps form key 'one' to the value 'uno'. If we print the dictionary this time, the output would be like:



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

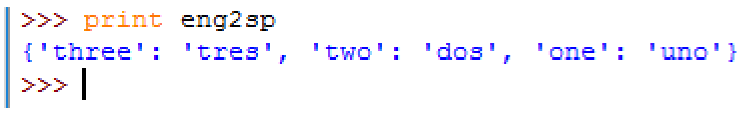
Here is a key-value pair with a colon between the key and value.

Creating dictionary with multiple items/key-value pairs:



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

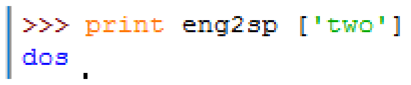
If we print this dictionary eng2sp, the output would be:



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

The order of key-value pair in output is not the same as in input. This order varies form computer to computer. In general, the order of items in a dictionary is unpredictable.

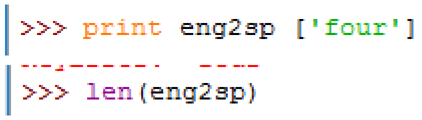
The key-value pairs being out of order doesn't matter because values in dictionary are not indexed with integers but with keys. The keys are used to look up for corresponding values. Ex:



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

The key *'two'* always maps to the value 'dos'. Always. Whenever you use the key *'two'*, it always, as already said maps to its value *'dos'*. So you got the idea why the order of key-value pairs/items doesn't matter.

Try:

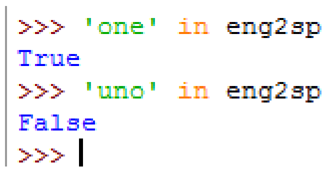


```
>>> print eng2sp ['four']
>>> len(eng2sp)
```

The ***len*** function upon a dictionary returns the number of key-value pairs.

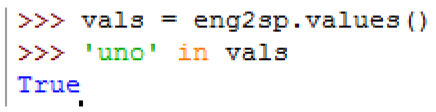
The ***in*** operator in dictionaries:

The ***in*** operator on dictionaries tells you whether something appears as a key. Example below:



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

To check whether something appears as a value in a dictionary. Ex below:



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

The ***values()*** method of dictionary object returns the values as a list. Now we can use in operator in list.

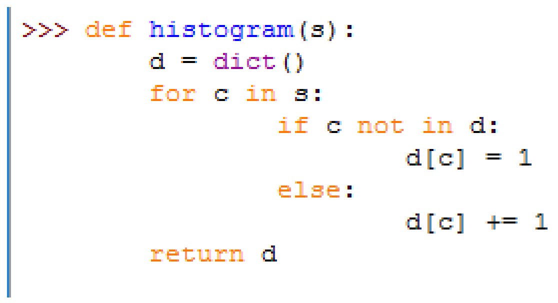
Ex 11.1

Dictionary as a set of counters.

Suppose you are given a string and you want to count how many times each letter appears.

You can create a list 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, and increment the value of an existing item.

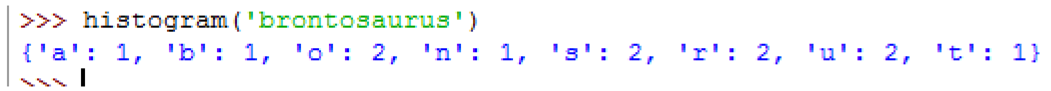
Ex below:



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

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].

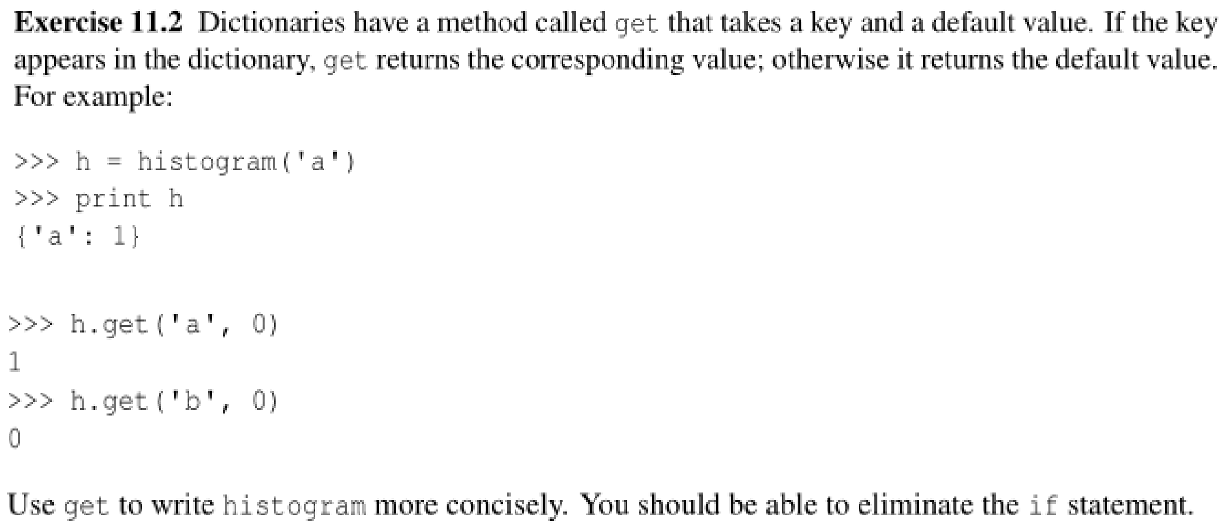
The output would be:



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

The histogram indicates 'a', 'b', 'n' appear once; 'o' appears twice and so on.

Ex 11.2



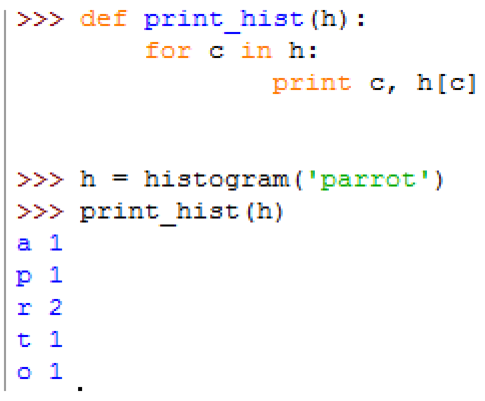
Exercise 11.2 Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```
>>> h = histogram('a')
>>> print h
{'a': 1}

>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

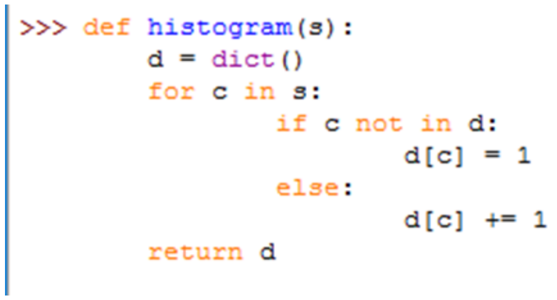
Use `get` to write `histogram` more concisely. You should be able to eliminate the `if` statement.

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:



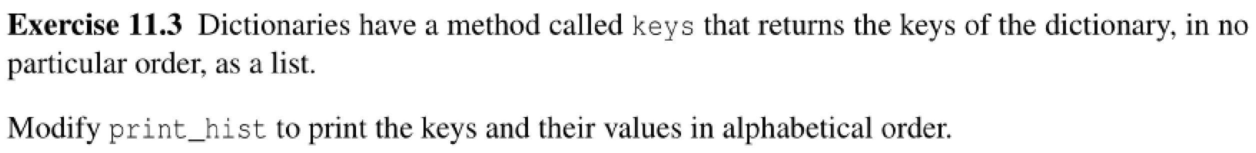
```
>>> def print_hist(h):
        for c in h:
                print c, h[c]


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



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

Notice: keys are not in particular order.

Ex:



Exercise 11.3 Dictionaries have a method called `keys` that returns the keys of the dictionary, in no particular order, as a list.

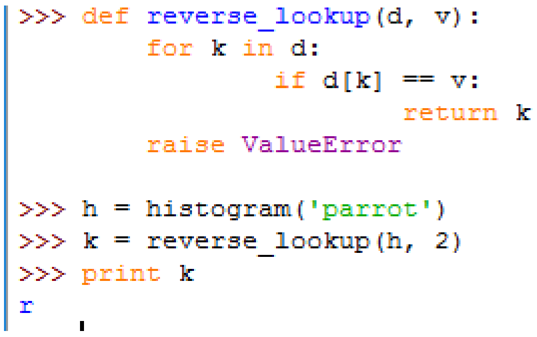
Modify `print_hist` to print the keys and their values in alphabetical order.

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.

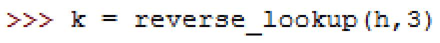
Here is a function that takes a value and returns the first key that maps to that value:



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

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

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.

Try:



```
>>> k = reverse_lookup(h,3)
```

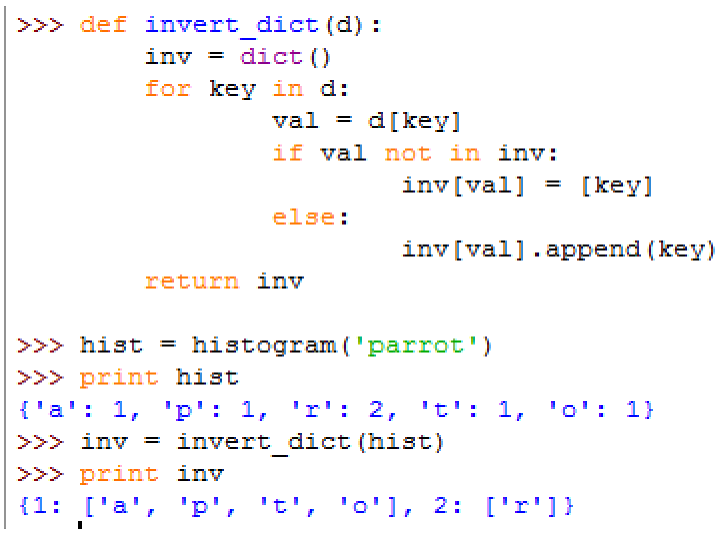
Ex:

Exercise 11.4 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.

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.

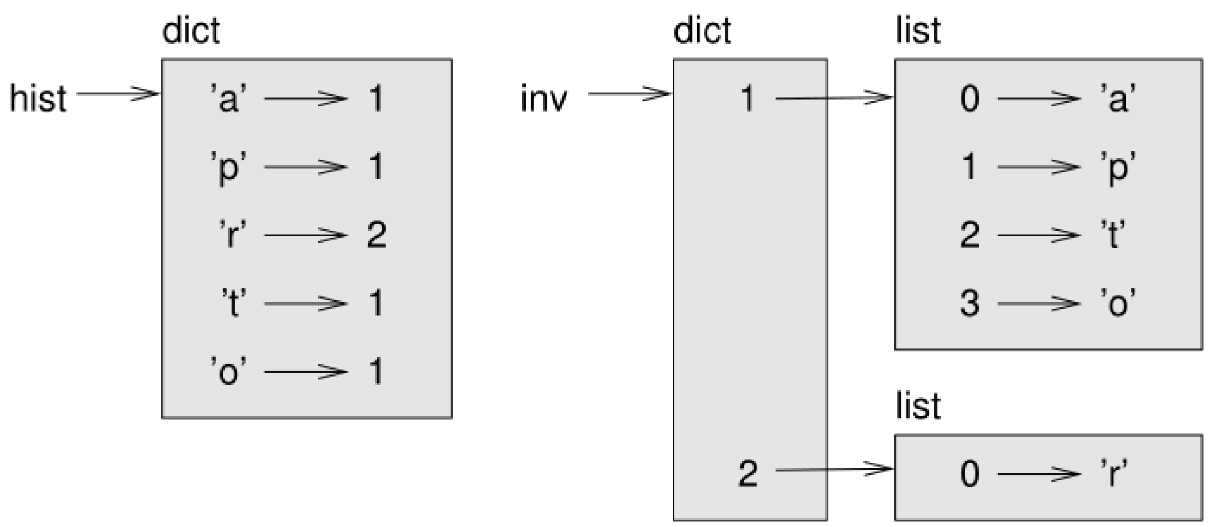
Here is the function that inverts a dictionary:



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

>>> 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']}
```

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 the state diagram showing hist and inv:



Lists can be values in a dictionary but they cannot be keys. For further explanation, refer to corresponding chapters in Think Python book.

Matrix: list of lists:

Nested lists are often used to represent matrices. For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

might be represented as:

```
>>> mx = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

mx is a list with three elements, where each element is a row of the matrix. We can select an entire row from the matrix in the usual way:

```
>>> mx[1]
[4, 5, 6]
```

Or we can extract a single element from the matrix using the double-index form:

```
>>> mx[1][2]
6
```

The first index selects the row, and the second index selects the column. Although this way of representing matrices is common, it is not the only possibility. A small variation is to use a list of columns instead of a list of rows. Later we will see a more radical alternative using a dictionary.

Sparse matrix using dictionary:

Just before we use list of lists to represent the matrix. It is very much good idea to use list of lists to create matrix mostly containing nonzero elements. But in case of sparse matrix:

In numerical analysis, a **sparse matrix** is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered **dense**. The number of zero-valued elements divided by the total number of elements (e.g., m × n for an m × n matrix) is called the **sparsity** of the matrix (which is equal to 1 minus the **density** of the matrix).

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

The list representation contains a lot of zeroes:

```
matrix = [[0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0],
          [0, 2, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 3, 0]]
```

An alternative is to use a dictionary. For the keys, we can use tuples that contain the row and column numbers. Here is the dictionary representation of the same matrix:

```
>>> matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
```

We only need three key: value pairs, one for each nonzero element of the matrix. Each key is a tuple, and each value is an integer.

To access an element of the matrix, we could use the [] operator:

```
>>> matrix[(0, 3)]
1
```

Notice that the syntax for the dictionary representation is not the same as the syntax for the nested list representation. Instead of two integer indices, we use one index, which is a tuple of integers.

There is one problem. If we specify an element that is zero, we get an error, because there is no entry in the dictionary with that key:

```
>>> matrix[(1, 3)]
KeyError: (1, 3)
```

The get method solves this problem:

```
>>> matrix.get((0, 3), 0)
1
```

The first argument is the key; the second argument is the value get should return if the key is not in the dictionary:

```
>>> matrix.get((1, 3), 0)
0
```