6.3 Dictionaries

Lists hold sequences of data of any type. Sometimes we want to do more than that. In many situations it is useful to associate related values. For example, a gradebook associates individual students with lists of grades. A concordance associates a word with places where it occurs. An inventory associates items with counts of how many there are. An associative data structure and operations to manipulate it are built into Python. This structure is called a *dictionary* and it is very useful.

Because lists are sequential, we can use numeric indexes to refer to the individual elements of a list. `L[0]` is the first element of list`L`, `L[1]` is the second element, and so forth. A list with 5 elements always uses the indexes 0, 1, 2, 3, 4. Dictionaries use a similar notation, with slightly different terminology. We say that a dictionary associates *values* with *keys*. The keys play the role of the indexes of a list, only the keys do not need to be numeric. For example, if `"bob"` is a key of dictionary `D`, then `D["bob"]` is the value associated with `"bob"`. We can create this association by assigning a value to `D["bob"]`, such as `D["bob"]= 57` or `D["bob"]= [1, 2, 3]`. We retrieve the value associated with `"bob"` by referring to `D["bob"]`, as in **print**(`D["bob"]`).

Dictionaries can have any immutable type for their keys. The immutability requirement comes from the way dictionaries are implemented internally. In practice, this requirement means that dictionaries cannot have lists as their keys. There are situations where it would be nice to have lists as keys. For example, we might want to represent a date by the list`[month, day, year]` (such as [9, 30, 2009]) and use such dates as keys for a dictionary. For these situations there is an immutable type similar to lists called *tuples*. We will discuss tuples in the next section. Strings, integers, and tuples are the most common keys for dictionaries.

The *empty dictionary* is denoted by the symbol {}. Although there are mechanisms to construct a dictionary from lists, we usually start with the empty dictionary and add items to it one at a time. Since dictionaries aren't sequential, we need to do a little work to print the information from one. Here is an easy way to do this. If `D` is a dictionary, then `D.keys( )` is a list of all of the keys of `D`. Since it is a list, we can process this with a **for**-loop, and then use `D[key]` to access the value associated with each `key`.

For example, if the values associated with each key are printable, we might use the following function to print the dictionary:

```
def PrintDictionary( D ):
    for key in D.keys():
        print(D[key])
```

As with lists, dictionaries are mutable structures, so we can pass them as arguments to functions that modify the contents of the dictionary. Perhaps the biggest issue with dictionaries is the need to check that a key exists before we try to find the value associated with it. This doesn't occur with lists; if a list has 4 entries we know they are indexed with 0, 1, 2, and 3. A dictionary could

have almost anything as its keys, so knowing the size of the dictionary tells us nothing about its keys. Suppose, for example, that we have a program that reads words and records how many times each of them occurs. We store this information in a dictionary Counts, where the words are the keys and numbers of instances of the words are the values. When we read a word, we don't *a priori* know if it is one of the keys or not. We could write

```
Counts[word] = 1
```

but if word has been seen before this will destroy any prior information about the number of times it has already occurred. We could write

```
Counts[word] = Counts[word] + 1
```

but this won't work if word has not been seen before, as there won't be a value for the right-hand side of the assignment. The right way to address this is to use a conditional statement:

```
if word in Counts.keys():
        Counts[word] = Counts[word] + 1
else:
        Counts[word] = 1
```

It is an error to index a dictionary with a something that is not currently a key of the dictionary; if you do this, your program will crash.

Example. We will now develop a simple inventory program that tracks items and counts. This program stores its information in a dictionary with strings (the item names) for the keys of a dictionary and numbers (the counts) for the values. This repeatedly asks the user for an item and a count. The first time we see an item we create a record for it. If the item comes up again, we add its count onto the existing count for that item. For example, we might have the following interaction with this program:

Enter an item or a blank to exit: `hammer`
Enter a count: `10`
Enter an item or a blank to exit: `screwdriver`
Enter a count: `24`
Enter an item or a blank to exit: `saw`
Enter a count: `5`
Enter an item or a blank to exit: `hammer`
Enter a count: `15`
Enter an item or a blank to exit: `saw`
Enter a count: `3`
Enter an item or a blank to exit:
Here is the current inventory:
hammer: 25
saw: 8
screwdriver: 24

Each iteration of the input gives us valuesa for item and count. This information is added to the dictionary in one of two ways. If the item is not already a key in the dictionary, it is added with

```
D[item] = count
```

On the other hand, if item is a key, then we increase its value by count:

```
D[item] = D[item] + count
```

This is very similar to the code above where we counted words; before updating the dictionary we check to see if the new index is already one of the dictionary's keys, and take appropriate action based on the answer to this question. We need such code in almost every program that makes use of dictionaries.

Here is the complete program:

```
# This keeps track of an inventory of items
# and counts.

def AddToDictionary(D, s, n):
# This adds todictionary D the information that n units
# of item s were found.
    if s in D.keys():
        D[s] = D[s] + n
    else:
        D[s] = n

def PrintDictionary(D):
# This prints the keys and values of the dictionary
#  with the keys in alphabetical order
    names = D.keys()
    names.sort()
    for item in names:
        print( "%s: %d" %(item, D[item]) )

def main():
    inventory = {}
    done = False
    while not done:
        item = input("Enter an item or a blank to exit: " )
        if item == "":
            done = True
            print()
        else:
            count = eval( input("Enter a count: " ) )
            AddToDictionary(inventory, item, count)
    PrintDictionary(inventory)

main()
```

Program 6.3.1: Inventory Program