Appendix H

Dictionaries in Brief

H.1 About dictionaries

Chapter 6 and Appendix G look at lists as a way to collect related data. Collections of data are an important aspect of programs, and lists are one way that data can be collected. Dictionaries are another important and flexible type of collection which we'll look at in this appendix.

While lists number each item using an index, dictionaries allow you to name each item. (In this appendix we'll use strings as dictionary item names, though other Python types can be used.) And like lists, you can use any Python type as the dictionary's values, so dictionaries are a very flexible type of collection.

Dictionaries store a collection of ***keys***, and each key is associated with a value. In other words, just as an index is used to access an item in a list, a key is the access to a dictionary's value. As keys are used to access items, each key needs to be unique in a given dictionary.

To make this more tangible, let's consider a program that will keep track of products, including product ID and price. The program might allow you to enter a product ID and find out the corresponding price for that product. (In practice, the data might be stored in a file or a database.)

In order to store product IDs and prices, you could use two parallel lists. Then in order to look up a price for a given product ID, the program could search the list of product IDs, determine the index if a match is found, and then use the index to locate the corresponding price. The data organization in this list arrangement would look like figure H-1.

productIDs:

'P-101'	'P-102'	…

prices:

9.95	12.95	…

Figure H-1: Product price information as parallel lists

Two parallel lists are OK but not ideal. First, the data are related but split into two variables. Also, searching for a particular product ID requires Python to potentially scan the entire product ID list. For a large set of data this would be inefficient.

In contrast with parallel lists, a dictionary organization could keep pairs of data, with the product ID as the key, and the price as the value. Figure H-2 shows this arrangement.

prices:

key	value
'P-101'	9.95
'P-102'	12.95
…	…

Figure H-2: Price information as key/value pairs

The dictionary arrangement has a few benefits over the list arrangement. First, lookup is faster in dictionaries, since they are structured so that a scan of potentially all entries is not needed as is the case for lists. Second, the data in the dictionary is more tightly connected – in this example one instead of two variables are needed. Lastly, if we want to capture additional information, for example product names, we can use a dictionary of dictionaries, which would allow us to name each of the data fields (chapter 9 covers this arrangement).

In the next section we'll look at how to create dictionaries. Then in section H.3, we'll see how data can be found in a dictionary.

H.2 Creating Dictionaries

Like lists, dictionaries can have their items indicated when they are created, or can start empty and have their contents subsequently added.

In order to define a dictionary, curly braces `{ }` are used. For example, for the price data in the prior section, these could be defined as:

```
prices = {'P-101': 9.95,
          'P-102': 12.95}
```

In the above statement, the key and value are separated by a colon, and each key/value pair is separated by a comma.

Alternatively, an empty dictionary can be created, and items added to it. To create an empty dictionary, use curly braces, for example:

```
prices = { }
```

Key/value pairs can be added to a dictionary using the syntax:

```
dictionaryVariable[key] = value
```

for example,

```
prices['P-101'] = 9.95
```


H.3 Finding data in a dictionary

Once a dictionary has been populated, it is easy and efficient to locate data using a key. The dictionary variable, with square brackets containing the key can be used to locate the data value associated with that key. The general syntax is:

```
value = dictionaryVariable[key]
```

Note this syntax is similar to how items in a list are found using an index. However, for dictionaries, the value within square brackets is not limited to an integer as it is when working with lists.

Note if you try to access a product ID that is not in the dictionary, Python will generate a `KeyError`. This is because dictionary lookups with a key that is not in the dictionary will result in an error. This is similar to using an index of a list that is past the end of the list. To protect against this error, we can use the `in` phrase to check if a key is in a dictionary. In general

```
<key> in <dictionaryVariable>
```

returns `True` if the <key> is contained in the <dictionary>. Similarly, `not in` can be used, to see if a key isn't in a dictionary:

```
<key> not in <dictionaryVariable>
```

H.4 Modelling a Data Tree with a Dictionary

A tree of data is sometimes used as a data structure. A data tree can be stored using a dictionary, by setting the values in the dictionary to dictionaries. In other words, this is a dictionary of dictionaries. This is described further in Chapter 9.

