W4.0.1 Additional Reading (Dictionaries)

Learning Outcomes

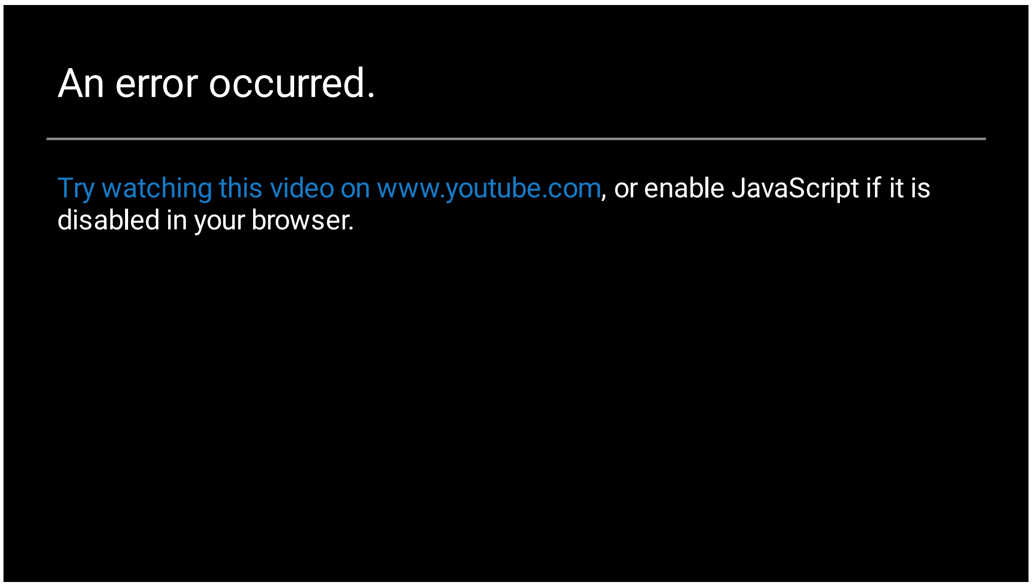
The compound data types we have studied in detail so far — strings and lists — are sequential collections. This means that the items in the collection are ordered from left to right and they use integers as indices to access the values they contain. This also means that looking for a particular value requires scanning the many items in the list until you find the desired value.

Data can sometimes be organized more usefully by associating a key with the value we are looking for. For example, if you are asked for the page number for the start of chapter 5 in a large textbook, you might flip around the book looking for the chapter 5 heading. If the chapter number appears in the header or footer of each page, you might be able to find the page number fairly quickly but it's generally easier and faster to go to the index page and see that chapter 5 starts on page 78.

This sort of direct look up of a value in Python is done with an object called a Dictionary. Dictionaries are a different kind of collection. They are Python's built-in **mapping** type. A map is an unordered, associative collection. The association, or mapping, is from a **key**, which can be of any immutable type (e.g., the chapter name and number in the analogy above), to a **value** (the starting page number), which can be any Python data object. You'll learn how to use these collections in the following lesson.

- To introduce the idea of Key, Value pairs
- To introduce the idea of an unordered sequence
- To understand that dictionary iteration iterates over keys

Getting Started with Dictionaries



An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Source: [Youtube - Google Students]

Let us look at an example of using a dictionary for a simple problem. We will create a dictionary to translate English words into Spanish. For this dictionary, the keys are strings and the values will also be strings.

One way to create a dictionary is to start with the empty dictionary and add **key-value pairs**. The empty dictionary is denoted `{}`.

The first assignment creates an empty dictionary named `eng2sp`. The other assignments add new key-value pairs to the dictionary. The left hand side gives the dictionary and the key being associated. The right hand side gives the value being associated with that key. We can print the current value of the dictionary in the usual way. The key-value pairs of the dictionary are separated by commas. Each pair contains a key and a value separated by a colon.

The order of the pairs may not be what you expected. Python uses complex algorithms, designed for very fast access, to determine where the key-value pairs are stored in a dictionary. For our purposes we can think of this ordering as unpredictable * .

Another way to create a dictionary is to provide a bunch of key-value pairs using the same syntax as the previous output.

It doesn’t matter what order we write the pairs. The values in a dictionary are accessed with keys, not with indices, so there is no need to care about ordering.

Here is how we use a key to look up the corresponding value.

Dictionary Operations

The `del` statement removes a key-value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock. If someone buys all of the pears, we can remove the entry from the dictionary.

Dictionaries are mutable (we will discuss mutability in depth in a future week), as the delete operation above indicates. As we've seen before with lists, this means that the dictionary can be modified by referencing an association on the left hand side of the assignment statement. In the previous example, instead of deleting the entry for `pears`, we could have set the inventory to `0`.

Note
Setting the value associated with `pears` to 0 has a different effect than removing the key-value pair entirely

with `del`. Try printout out the two dictionaries in the examples above.

Similarily, a new shipment of 200 bananas arriving could be handled like this. Notice that there are now 512 bananas— the dictionary has been modified. Note also that the `len` function also works on dictionaries. It returns the number of key-value pairs.

Notice that there are now 512 bananas—the dictionary has been modified. Note also that the `len` function also works on dictionaries. It returns the number of key-value pairs.

Dictionary Methods

Dictionaries have a number of useful built-in methods. The following table provides a summary and more details can be found in the Python Documentation.

Method	Parameters	Description
keys	none	Returns a view of the keys in the dictionary
values	none	Returns a view of the values in the dictionary
items	none	Returns a view of the key-value pairs in the dictionary
get	key	Returns the value associated with key; None otherwise
get	key,alt	Returns the value associated with key; alt otherwise

As we saw earlier with strings and lists, dictionary methods use dot notation, which specifies the name of the method to the right of the dot and the name of the object on which to apply the method immediately to the left of the dot. For example, if `x` is a variable whose value is a dictionary, `x.keys` is the method object, and `x.keys()` invokes the method, returning a view of the value.

Iterating Over Dictionaries

There are three ways to iterate over the contents of a dictionary. Let's take a moment to examine them.

The first technique involves iterating over the keys of the dictionary using the `keys` method. The `keys` method returns a collection of the keys in the dictionary.

```
inventory = {'apples': 430, 'bananas': 312, 'pears': 217, 'oranges': 525}

for akey in inventory.keys():     # the order in which we get the keys is not defined
    print("Got key", akey, "which maps to value", inventory[akey])

ks = list(inventory.keys())       # Make a list of all of the keys
print(ks)
print(ks[0])                      # Display the first key
```

Note the first line of the for loop:

```
for akey in inventory.keys():
```

Each time through the loop, the loop variable `akey` is assigned a different key in the dictionary. In the loop body, the value associated with the key is accessed by indexing the dictionary with `akey` using the expression `inventory[akey]`. Note that the order in which the keys are assigned in the loop is not predictable. If you want to visit the keys in alphabetic order, you must use the `sorted` function to produce a sorted collection of keys, like this:

```
for akey in sorted(inventory.keys()):
```

It's so common to iterate over the keys in a dictionary that you can omit the `keys` method call in the `for` loop — iterating over a dictionary implicitly iterates over its keys.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

for k in inventory:
    print("Got key", k)
```

The `values` method returns a collection of the values in the dictionary. Here's an example that displays a list of the values:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

print(list(inventory.values()))

for v in inventory.values():
    print("Got", v)
```

The `items` method returns a collection of tuples containing each key and its associated value. Take a look at this example that iterates over the dictionary using the `items` method:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

print(list(inventory.items()))

for k, v in inventory.items():
    print("Got", k, "that maps to", v)
```

ake a close look at the first line of the for loop:

```
for k, v in inventory.items():
```

Each time through the loop, `k` receives a key from the dictionary, and `v` receives its associated value. That avoids the need to index the dictionary inside the loop body to access the value associated with the key.

Safely Retrieving Values

Looking up a value in a dictionary is a potentially dangerous operation. When using the `[]` operator to access a key, if the key is not present, a runtime error occurs. There are two ways to deal with this problem.

The first approach is to use the `in` and `not in` operators, which can test if a key is in the dictionary:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
print('apples' in inventory)
print('cherries' in inventory)

if 'bananas' in inventory:
    print(inventory['bananas'])
else:
    print("We have no bananas")
```

The second approach is to use the `get` method. `get` retrieves the value associated with a key, similar to the `[]` operator. The important difference is that `get` will not cause a runtime error if the key is not present. It will instead return the value `None`. There exists a variation of `get` that allows a second parameter that serves as an alternative return value in the case where the key is not present. This can be seen in the final example below. In this case, since “cherries” is not a key, `get` returns 0 (instead of None).

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

print(inventory.get("apples"))
print(inventory.get("cherries"))

print(inventory.get("cherries",0))
```

TIP: When to use a Dictionary

Now that you have experience using lists and dictionaries, you will have to decide which one is best to use in each situation. The following guidelines will help you recognize when a dictionary will be beneficial:

- When a piece of data consists of a set of properties of a single item, a dictionary is often better. You could try to keep track mentally that the zip code property is at index 2 in a list, but your code will be easier to read and you will make fewer mistakes if you can look up mydiction['zipcode'] than if you look up mylst[2].
- When you have a collection of data pairs, and you will often have to look up one of the pairs based on its first value, it is better to use a dictionary than a list of (key, value) tuples. With a dictionary, you can find the value for any (key, value) tuple by looking up the key. With a list of tuples you would need to iterate through the list, examining each pair to see if it had the key that you want.
- On the other hand, if you will have a collection of data pairs where multiple pairs share the same first data element, then you can't use a dictionary, because a dictionary requires all the keys to be distinct from each other.

Glossary

dictionary
A collection of key-value pairs that maps from keys to values. The keys can be any immutable type, and the values can be any type.

key
A data item that is *mapped to* a value in a dictionary. Keys are used to look up values in a dictionary.

value
The value that is associated with each key in a dictionary.

key-value pair
One of the pairs of items in a dictionary. Values are looked up in a dictionary by key.

mapping type
A mapping type is a data type comprised of a collection of keys and associated values. Python’s only built-in mapping type is the dictionary. Dictionaries implement the associative array abstract data type.