Dictionaries and Sets

Motivation

In Computer Science a large focus of the discipline concerns itself with information – both manipulating and managing data. One of Python's strengths is being good at offering facilities for manipulating data. Two key collections of data that Python supports are: dictionaries and sets. In computing, a **collection** is a container or a grouping of some variable number of data items that share significance to the software problem under review.

A Python **dictionary** is an important software structure for dealing with data. A dictionary allows a more complex, and often useful indexing scheme to manage a collection of elements. Unlike a sequence (such as a list) that is accessed by using an integer index, a dictionary can be indexed by a string referred to as a *key*. The information (value) at the location in the dictionary is associated with that key making for logical **key:value** pairs to be constructed.

Review and Looking Forward

Last week you looked at processing strings in more depth. Along with lists and tuples, strings contain sequences of values. In computing, a sequence structure contains an indexed order of items. A sequence is essentially a succession of items that are bound together in a collection of values. The simplest sequence is a string, which is a sequence of characters. Next you might say is the list, which is a sequence of possibly variable type items. You have seen that even though strings and lists are both sequences, they have differences (e.g. a string can be converted to all uppercase, a new item can be appended to a list). Regardless of the type of sequence, remember two important functions are consistently supported: **index()** and **count()**.

This week we will look at another Python collection mechanism, called a **dictionary**. Collections, also sometimes referred to as container data types , are data stores in which multiple elements are grouped into a single unit usually based on a natural grouping (e.g. names and phone numbers in a phonebook). A collection has an underlying built-in support structure that is used for efficient data manipulation and storage. Python's general purpose built-in containers include **list**, **tuple**, **dict** and **set**. You have learned about **list** and **tuple**. Now it is time to learn about **dict** and **set**.

I think that you will particularly enjoy working with dictionaries in Python...Let's get started!

Dictionary

A dictionary in Python is a collection of related data pairs. For instance, if you wanted to store the names and phone numbers of 5 key contacts you could store this information in a dictionary.

To declare a dictionary, write dictionary_name = {key : value} with the requirement that dictionary keys must be unique. *Note:* Use curly brackets **{ }** when declaring a dictionary. Multiple pairs are separated by a comma.

```
>>> phone_book = {'Bob' : '408-555-1111', 'Ned' : '415-636-5555'}>>>
type(phone_book)
<class 'dict'>
>>> phone_book
{'Ned': '415-636-5555', 'Bob': '408-555-1111'}
```

A dictionary can also be declared using the **dict()** method, assigning key value pairs. Note: Using the **dict()** method to declare a dictionary, use round brackets **( )** and not to put quotation marks for the key.

```
>>> guest_book = dict(White = 2, Raman = 4)
>>> type(guest_book)
<class 'dict'>
>>> guest_book
{'Raman': 4, 'White': 2}
```

Caution: Order of entries may not be as you may expect! *Hint*: This is an unordered collection – as such, the items in a dictionary are not necessarily stored in the same order as the way you declare them.

A dictionary can also be declared without assigning any initial values to it, creating an empty dictionary.

```
>>> empty_book = {}
>>> empty_book
{}
```

To access individual items in the dictionary, use the dictionary key .

```
>>> phone_number = phone_book['Ned']
>>> print(phone_number)
415-636-5555
>>> num_guests = guest_book['White'] + guest_book['Raman']
>>> num_guests
6
```

To modify an item in the dictionary, re-assign the dictionary key of the item to be modified to the new data.

```
>>>phone_book
{'Ned': '415-636-5555', 'Bob': '408-555-1111'}
>>> phone_book['Bob'] = '408-555-2222'
>>> phone_book
{'Ned': '415-636-5555', 'Bob': '408-555-2222'}
```

To add an item to a dictionary, write as follows: dictionary_name[key] = item.

```
>>> empty_book['new_key'] = 'new_item'
>>> empty_book
{'new_key': 'new_item'}
>>> guest_book['Lin'] = 3
>>> guest_book
{'Raman': 4, 'White': 2, 'Lin': 3}
```

To remove an item in a dictionary, use the **del** keyword as follows: del dictionary_name[key] .

```
>>> guest_book
{'Raman': 4, 'White': 2, 'Lin': 3}
>>> del guest_book['White']
>>> guest_book
```

```
{'Raman': 4, 'Lin': 3}
```

Trying to a non-existing item in a dictionary will generate an error. To prevent this, check first if an item is in the dictionary using the **in** operator before issuing a **del()** statement.

```
>>> guest_book
{'Lin': 3, 'Raman': 4}
>>> del guest_book['Nobody']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Nobody'
>>> 'Nobody' in guest_book
False
>>> if 'Nobody' in guest_book:
...     del guest_book['Nobody']
...
>>>
```

Dictionary keys and data values can be of different types.

```
>>> mixed_dict = {'fp': 1.25, 1.75 : "One Point Seventy Five", 3: "%",
3.5:1}
>>> print(mixed_dict)
{3.5: 1, 'fp': 1.25, 3: '%', 1.75: 'One Point Seventy Five'}
>>> print(mixed_dict["fp"])
1.25
>>> print(mixed_dict[1.75])
One Point Seventy Five
>>> print(mixed_dict[3])
%
>>>
>>> print(mixed_dict[3.5])
```

```
1
```

Using the **clear()** method, all elements of the dictionary can be removed, returning an empty dictionary.

```
>>> mixed_dict.clear()
>>> print(mixed_dict)
{}
```

The **get()**method will return a value for a given key. If the key is not found it will return the **None**.

```
>>> demo_dict = {1: "one", 2: "two"}
>>> demo_dict.get(2)
'two'
>>> see_ret = demo_dict.get(3)
>>> if see_ret == None:
...     print('y')
...
y
```

Alternatively, when using the **get()**method, you can state the value to return if the key is not found.

```
>>>>>> demo_dict.get(3, "Key 3 is not found")
'Key 3 is not found'
>>> demo_dict.get(2, "Key 2 is not found")
'two'
```

The **keys()** method will return a list of the dictionary's keys.

```
>>> demo_dict.keys()
```

The **values()** method returns a list of the dictionary's values.

```
>>> demo_dict.values()
```

```
dict_values(['one', 'two'])
```

The dictionary pairs can be captured as tuples in a list using the **items()** method.

```
>>> guest_tuple = guest_book.items()
>>> type(guest_tuple)
<class 'dict_items'>
>>> guest_tuple
dict_items([('Lin', 3), ('Raman', 4)])
```

The **update()** adds one dictionary's key-values pairs to another. Duplicates are removed.

```
>>> demo_dict.update(guest_book)
>>>demo_dict
{1: 'one', 2: 'two', 'Raman': 4, 'Lin': 3}
```

Set

A Python **set** is another type of data container available. A set is a unordered collection of unique values. Because a set is unordered, operations such as indexing and slicing are not provided. However, membership (using **in**), size (using **len()** function) and looping on membership (**for** index in set) are all supported. A set is created by placing all the items inside curly braces {}, separated by a comma.

```
>>> set_one = {1,3,5,7,9}
>>> type(set_one)
<class 'set'>
```

Alternatively a set can be created by using the built-in function **set().** A set can be made from a list by passing the list as an argument to the set function.

```
set_two = set([4,8])
>>> type(set_two)
<class 'set'>
```

```
>>> set_two
{8, 4}
>>> set_three = set(range(1,3))
>>> set_three
{1, 2}
```

Caution: Using empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the **set()** function without any argument.

```
>>> empty_set_err = {}
>>> type(empty_set_err)
<class 'dict'>
>>> empty_set = set()
>>> type(empty_set)
<class 'set'>
```

A set does not retain duplicates.

```
>>> set_four = {1, 1}
>>> set_four
{1}
```

An item can be added to a set using the **add ()** method.

```
>>> set_four.add(2)
>>> set_four
{1, 2}
```

Multiple items can be added using the **update()** method.

```
>>> set_four.update([3,4])
>>> set_four
{1, 2, 3, 4}
```

A particular item can be removed from a set using methods discard() and remove(). While using **discard()**, if the item does not exist the set is unchanged; using **remove()** will raise an error.

```
>>> set_four.discard(5)
>>> set_four
{1, 2, 3, 4}
>>> set_four.remove(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 5
>>> set_four.remove(4)
>>> set_four
{1, 2, 3}
```

All of the examples so far have shown integers belonging to a set, but other types can belong as well: floating point numbers, string, and even tuples (yet not lists).

```
>>> set_str = {"red", "green", "blue"}
>>> set_str
{'green', 'red', 'blue'}
```

Sets can be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

```
>>> set_one
{9, 1, 3, 5, 7}
>>> set_two
{8, 4}
>>> set_three
{1, 2}
>>> set_four
{1, 2, 3}
>>> set_three < set_four
```

```
True
>>> set1 | set_three
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'set1' is not defined
>>> set_one | set_three
{1, 2, 3, 5, 7, 9}
>>> set_three & set_four
{1, 2}
>>> set_one - set_four
{9, 5, 7}
```

That's it for now…*Congratulations*, you are ready to use sets and dictionaries!