Chapter 34
Collection Related Modules

34.1 Introduction

The chapter introduces a feature known as a *list comprehension* in Python.
It then introduces the `collections` and `itertools` modules.

34.2 List Comprehension

This is a very powerful mechanism that can be used to generate new lists.
The syntax of the List Comprehension is:

```
[ <expression> for item in iterable <if optional_condition> ]
```

The new list is formed of the results from the expression. Note that the whole `for` statement and expression is surrounded by the square brackets normally used to create a `List`.

When a List Comprehension is executed it generates a new list by applying the expression to the items in another collection.

For example, given one list of integers, another (new) list can be created using the List Comprehension format:

```
list1 = [1, 2, 3, 4, 5,6]
print('list1:', list1)

list2 = [item + 1 for item in list1]
print('list2:', list2)
```

© Springer Nature Switzerland AG 2019

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_34

which produces the output:

```
list1: [1, 2, 3, 4, 5, 6]
list2: [2, 3, 4, 5, 6, 7]
```

Here the new list is generated by adding 1 to each element in the initial list `list1`. Essentially, we iterate (loop) over all the elements in the initial list and bind each element in the list to the `item` variable in turn. The result of the expression `item + 1` is then captured, in order, in the new list.

This feature is not limited to processing values in a list; any iterable collection can be used such as Tuples or Sets as the source of the values to process.

Another feature of the List Comprehension is the ability to filter the values passed to the expression using the optional if condition.

For example, if we wish to only consider even numbers for the expression then we can use the optional `if` statement to filter out all odd numbers:

```
list3 = [item + 1 for item in list1 if item % 2 == 0]
print('list3:', list3)
```

The output from these two lines of code is:

```
list3: [3, 5, 7]
```

Thus only the even numbers were passed to the expression `item + 1` resulting in only the values 3, 5 and 7 being generated for the new list.

34.3 The Collections Module

The `collections` module extends that basic features of the collection-oriented data types within Python with high performance container data types. It provides many useful containers such as:

Name	Purpose
namedtuple()	Factory function for creating tuple subclasses with named fields
deque	List-like container with fast appends and pops on either end
ChainMap	Dict-like class for creating a single view of multiple mappings
Counter	Dict subclass for counting hashable objects
OrderedDict	Dict subclass that remembers the order entries were added
Defaultdict	Dict subclass that calls a factory function to supply missing values
UserDict	Wrapper around dictionary objects for easier dict subclassing
UserList	Wrapper around list objects for easier list subclassing
UserString	Wrapper around string objects for easier string subclassing

As this is not one of the default modules that are automatically loaded for you by Python; you will need to `import` the collection.

As an example, we will use the `Counter` type to efficiently hold multiple copies of the same element. It is efficient because it only holds one copy of each element but keeps a count of the number of times that element has been added to the collection:

```
import collections

fruit = collections.Counter(['apple', 'orange', 'pear',
'apple', 'orange', 'apple'])
print(fruit)
print(fruit['orange'])
```

The output of this is:

```
Counter({'apple': 3, 'orange': 2, 'pear': 1})
2
```

Which makes the counting behaviour associated with the `Counter` class quite clear.

There are many uses of such a class, for example, it can be used to find out the most frequently used word in an essay; all you have to do is add each word in an essay to the `Counter` and then retrieve the word with the highest count. This can be done using the `Counter` class's `most_common()` method. This method takes a parameter `n` that indicates how many of the *most common* elements should be returned. If `n` is ommitted (or `None`) then the method returns an ordered list of elements. Thus to obtain the most common fruit from the above `Counter` collection we can use:

```
print('fruit.most_common(1):', fruit.most_common(1))
```

Which generates:

```
fruit.most_common(1): [('apple', 3)]
```

You can also perform some mathematical operations with multiple `Counter` objects. For example, you can add and subtract `Counter` objects. You can also obtain a combination of Counters that combines the maximum values from two `Counter` objects. You can also generate an intersection of two Counters. These are all illustrated below:

```
fruit1 = collections.Counter(['apple', 'orange', 'pear',
'orange'])
fruit2 = collections.Counter(['banana', 'apple', 'apple'])

print('fruit1:', fruit1)
print('fruit2:', fruit2)

print('fruit1 + fruit2:', fruit1 + fruit2)
print('fruit1 - fruit2:', fruit1 - fruit2)
# Union (max(fruit1[n], fruit2[n])
print('fruit1 | fruit2:', fruit1 | fruit2)
# Intersection (min(fruit1[n], fruit2[n])
print('fruit1 & fruit2:', fruit1 & fruit2)
```

Which produces:

```
fruit1: Counter({'orange': 2, 'apple': 1, 'pear': 1})
fruit2: Counter({'apple': 2, 'banana': 1})
fruit1 + fruit2: Counter({'apple': 3, 'orange': 2, 'pear': 1,
'banana': 1})
fruit1 - fruit2: Counter({'orange': 2, 'pear': 1})
fruit1 | fruit2: Counter({'apple': 2, 'orange': 2, 'pear': 1,
'banana': 1})
fruit1 & fruit2: Counter({'apple': 1})
```

Once a `Counter` object has been created you can test it to see if an item is present using the `in` keyword, for example:

```
print('apple' in fruit)
```

You can also add items to a `Counter` object by accessing the value using the item as the key, for example:

```
fruit['apple'] = 1 # initialises the number of apples
fruit['apple'] =+ 1 # Adds one to the number of apples
fruit['apple'] =- 1 # Subtracts 1 from the number of apples
```

34.4 The Itertools Module

The `itertools` module is another module that it is worth being familiar with. This module provides a number of useful functions that return iterators constructed in various ways. As there are many different options available it is worth looking at the documentation for a complete list of available functions.

To give you a flavour of some of the facilities available look at the following listing:

```
import itertools

# Connect two iterators together
r1 = list(itertools.chain([1, 2, 3], [2, 3, 4]))
print(r1)

# Create iterator with element repeated specified number of
#  times (possibly infinite)
r2 = list(itertools.repeat('hello', 5))
print(r2)

# Create iterator with elements from first iterator starting
#  where predicate function fails
values = [1, 3, 5, 7, 9, 3, 1]
r3 = list(itertools.dropwhile(lambda x: x < 5, values))
print(r3)

# Create iterator with elements from supplied iterator between
#  the two indexes (use 'None' for second index to go to end)
r4 = list(itertools.islice(values, 3, 6))
print(r4)
```

The output from this code is:

```
[1, 2, 3, 2, 3, 4]
['hello', 'hello', 'hello', 'hello', 'hello']
[5, 7, 9, 3, 1]
[7, 9, 3]
```

34.5 Online Resources

Online resources on the itertools library are listed below:

- https://docs.python.org/3.7/library/itertools.html The standard library documentation on the itertools module.
- https://pymotw.com/3/itertools/index.html The Python module of the week page for itertools.

34.6 Exercises

The aim of this exercise is to create a concordance program in Python using the collections.Counter class.

For the purposes of this exercise a concordance is an alphabetical list of the words present in a text or texts with a count of the number of times that the word occurs.

Your concordance program should include the following steps:

- Ask the user to input a sentence.
- Slit the sentence into individual words (you can use the `split()` method of the string class for this.
- Use the `Counter` class to generate a list of all the words in the sentence and the number of times they occur.
- Produce an alphabetic list of the words with their counts. You can obtain a sorted list of the keys using the `sorted()` function. You can then use a collection. `OrderedDict` to generate a dictionary that maintains the order in which keys were added.
- Print out the alphabetically ordered list.

An example of how the program might operate is given below:

```
Please enter text to be analysed: cat sat mat hat cat hat cat
Unordered Counter
Counter({'cat': 3, 'hat': 2, 'sat': 1, 'mat': 1})
Ordered Word Count
OrderedDict([('cat', 3), ('hat', 2), ('mat', 1), ('sat', 1)])
```