Chapter 36
Map, Filter and Reduce

36.1 Introduction

Python provides three functions that are widely used to implement functional programming style solutions in combination with collection container types.

These functions are what are known as higher-order functions that take both a collection and a function that will be applied in various ways to that collection.

This chapter introduces three functions `filter()`, `map()` and `reduce()`.

36.2 Filter

The `filter()` function is a higher order function that takes a function to be used to filter out elements from a collection. The result of the `filter()` function is a new iterable containing only those elements selected by the test function.

That is, the function passed into `filter()` is used to test all the elements in the collection that is also passed into filter. Those where the test filter returns `True` are included in the list of values returned. The result returned is a new iterable consisting of all elements of this list that satisfy the given test function. Note that the order of the elements is preserved.

The syntax of the `filter()` function is

```
filter(function, iterable)
```

Note that the second argument to the `filter` function is anything that implements the iterable protocol which includes all Lists, Tuples, Sets and dictionaries or and many other types etc.

The function passed in as the first argument is the test function; it can be a lambda (a function defined in line) or the name of an existing function. The result returned will be an iterable that can be used to create an appropriate collection.

Here are some examples of using `filter` with a simple list of integers:

```
data = [1, 3, 5, 2, 7, 4, 10]
print('data:', data)

# Filter for even numbers using a lambda function
d1 = list(filter(lambda i: i % 2 == 0, data))
print('d1:', d1)

def is_even(i):
    return i % 2 == 0

# Filter for even numbers using a named function
d2 = list(filter(is_even, data))
print('d2:', d2)
```

The output from this is:

```
Data: [1, 3, 5, 2, 7, 4, 10]
d1: [2, 4, 10]
d2: [2, 4, 10]
```

One difference between the two examples is that it is more obvious what the role is of the test function in the second example as it is explicitly named (i.e. `is_even()`), that is the function is testing the integer to see whether it is even or not. The in-line `lambda` function does exactly the same, but it is necessary to understand the test function itself to work out what it is doing.

It is also worth pointing out that defining a named function such as `is_even()` may actually pollute the namespace of the module as there is now a function that others might decide to use even though the original designer of this code never expected anyone else to use the `is_even()` function. This is why lambda functions are often used with `filter()` (and indeed `map()` and `reduce()`).

Of course, you are not just limited to fundamental built in types such as integers, or real numbers or indeed strings; any type can be used. For example, if we have a class `Person` such as:

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return 'Person(' + self.name +
                         ', ' + str(self.age) + ')'
```

Then we can create a list of instances of the class `Person` and then filter out all those over 21:

```
data = [Person('Alun', 54), Person('Niki', 21), Person('Megan',
19)]
for p in data:
    print(p, end=', ')

print('\n-----')

# Use a lambda to filter out People over 21
d3 = list(filter(lambda p: p.age <= 21, data))
for p in d3:
    print(p, end=', ')
```

The output from this is:

```
Person(Alun, 54), Person(Niki, 21), Person(Megan, 19),
-----
Person(Niki, 21), Person(Megan, 19),
```

36.3 Map

`Map` is another higher order function available in Python. `Map` applies the supplied function to all items in the iterable(s) passed to it. It returns a new iterable of the results generated by the applied function.

It is the functional equivalent of a `for` loop applied to an iterable where the results of each iteration round the `for` loop are gathered up.

The `map` function is very widely used within the functional programming world and it is certainly worth becoming familiar with it.

The function signature of map is

```
map(function, iterable, ...)
```

Note that the second argument to the map function is anything that implements the iterable protocol.

The function passed into the `map` function is applied to each item in the iterable passed as the second argument. The result returned from the function is then gathered up into the iterable object returned from `map`.

The following example applies a function that adds one to a number, to a list of integers:

```
data = [1, 3, 5, 2, 7, 4, 10]
print('data:', data)

# Apply the lambda function to each element in the list
# using the map function
d1 = list(map(lambda i: i + 1, data))
print('d1', d1)

def add_one(i):
    return i + 1

# Apply the add_one function to each element in the
# list using the map function
d2 = list(map(add_one, data))
print('d2:', d2)
```

The output of the above example is:

```
data: [1, 3, 5, 2, 7, 4, 10]
d1 [2, 4, 6, 3, 8, 5, 11]
d2: [2, 4, 6, 3, 8, 5, 11]
```

As with the `filter()` function, the function to be applied can either be defined in line as a lambda or it can be named function as in `add_one()`. Either can be used, the advantage of the `add_one()` named function is that it makes the intent of the function explicit; however, it does pollute the namespace of functions defined.

Note that more than one iterable can be passed to the `map` function. If multiple iterables are passed to `map`, then the function passed in must take as many parameters as there are iterables. This feature is useful if you want to merge data held in two or more collections into a single collection.

For example, let us assume that we want to add the numbers in one list to the numbers in another list, we can write a function that takes two parameters and returns the result of adding these two numbers together:

```
data1 = [1, 3, 5, 7]
data2 = [2, 4, 6, 8]

result = list(map(lambda x, y: x + y, data1, data2))
print(result)
```

The output printed by this is:

```
[3, 7, 11, 15]
```

As with the `filter` function, it is not only built-in types such as numbers that can be processed by the function supplied to `map`; we can also use user defined types such as the class `Person`. For example, if we wanted to collect all the ages for a list of `Person` we could write:

```
data = [Person('John', 54), Person('Phoebe', 21),
Person('Adam', 19)]
ages = list(map(lambda p: p.age, data))
print(ages)
```

Which creates a list of the ages of the three people:

```
[54, 21, 19]
```

36.4 Reduce

The `reduce()` function is the last higher order function that can be used with collections of data that we will look at.

The `reduce()` function applies a function to an iterable and combines the result returned for each element together into a single result.

This function was part of the core Python 2 language but was not included into the core of Python 3. This is partly because Guido van Rossum believed (probably correctly) that the applicability of `reduce` is quite limited, but where it is useful it is very useful. Although it has to be said that some developers try and shoe horn `reduce()` into situations that just make the implementation very hard to understand—remember always aim to keep it simple.

To use `reduce()` in Python 3 you need to import it from the `functools` module. One point that is sometimes misunderstood with `reduce()` is that the function passed into reduce takes two parameters, which are the previous result and the next value in the sequence; it then returns the result of applying some operation to these parameters.

The signature of the `functools.reduce` function is:

```
functools.reduce(function, iterable[, initializer])
```

Note that optionally you can provide an initialiser that is used to provide an initial value for the result.

One obvious use of `reduce()` is to sum all the values in a list:

```
from functools import reduce

data = [1, 3, 5, 2, 7, 4, 10]
result = reduce(lambda total, value: total + value, data)
print(result)
```

The result printed out for this is 32.

Although it might appear that `reduce()` is only useful for numbers such as integers; it can be used with other types as well. For example, let us assume that we want to calculate the average age for a list of people, we could use reduce to add together all the ages and then divide by the length of the data list we are processing:

```
data = [Person('John', 54), Person('Phoebe', 21),
        Person('Adam', 19)]

total_age = reduce(lambda running_total, person: running_total
+ person.age, data, 0)

average_age = total_age // len(data)
print('Average age:', average_age)
```

In this code example, we have a data list of three people. We then use the `reduce` function to apply a lambda to the data list. The lambda takes a `running_total` and adds a person's age to that total. The value zero is used to initialise this running total. When the lambda is applied to the data list, we will add 54, 21 and 19 together. We then divide the final result returned by the length of the data list (3) using the `//` operator which will use `floor()` division to return a whole integer (rather than a real number such as 3.11). Finally, we print out the average age:

```
Average age: 31
```

36.5 Online Resources

More information on *map*, *filter* and *reduce* can be found using the following online resources:

- http://book.pythontips.com/en/latest/map_filter.html Summary of map, filter and reduce.
- https://www.w3schools.com/python/ref_func_map.asp The W3C schools map() function tutorial.
- https://www.w3schools.com/python/ref_func_filter.asp The W3 schools filter() function tutorial.
- https://pymotw.com/3/functools/index.html The Python Module of the Week page including reduce().
- https://docs.python.org/3.7/library/functools.html The Python Standard Library documentation for functors including `reduce()`.

36.6 Exercises

This exercise aims to allow you to use map and filter with your `Stack` class.

Take the `Stack` that you developed in the last chapter and make it iterable. This can be done by implementing the `__iter__()` method from the iterable protocol. As a list is held internally to the `Stack` this could be implemented by returning an iterable wrapper around the list, for example:

```
    def __iter__(self):
        return iter(self._list)
```

Now define a function that will check to see if a string passed to it starts with 'Job'; if it does return `True` if not return `False`. Call this function `is_job()`.

Also define a function that will prepend the string 'item': to the string passed in and will then return this as the result of the function. Call this function `add_item()`.

You should now be able to use the `filter()` and `map()` functions with the `Stack` class as shown below:

```
stack = Stack()
stack.push('Task1')
stack.push('Task2')
stack.push('Job1')
stack.push('Task3')
stack.push('Job2')
print('stack contents:', stack)

# Apply functions to stack contents using map and filter
new_list = list(map(add_item, stack))
print('new_list:', new_list)

filtered_list = list(filter(is_job, stack))
print('filtered_list: ', filtered_list)
```