Chapter 30
Iterables, Iterators, Generators and Coroutines

30.1 Introduction

There are two protocols that you are very likely to use, or will possibly need to implement at some point or other; these are the *Iterable* protocol and the *Iterator* protocols.

These two closely related protocols are very widely used and supported by a large number of types.

One of the reasons that Iterators and Iterables are import is that they can be used with `for` statements in Python; this makes it very easy to integrate an iterable into code which needs to process a sequence of values in turn. Two further iteration like Python features are Generators and Coroutines which are discussed at the end of this chapter.

30.2 Iteration

30.2.1 Iterables

The *Iterable* protocol is used by types where it is possible to process their contents one at a time in turn. An Iterable is something that will supply an Iterator that can be used to perform this processing. As such it is not the iterator itself; but the provider of the iterator.

There are many iterable types in Python including Lists, Sets, Dictionaries, tuples etc. These are all iterable *containers* that will supply an iterator.

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_30

To be an *iterable* type; it is necessary to implement the `__iter__()` method (which is the only method in the Iterable protocol). This method must supply a reference to the iterator object. This reference could be to the data type itself or it could be to another type that implements the iterator protocol.

30.2.2 Iterators

An *iterator* is an object that will return a sequence of values. Iterators may be finite in length or infinite (although many container-oriented iterators provide a fixed set of values).

The iterator protocol specifies the `__next__()` method. This method is expected to return the next item in the sequence to return or to raise the `StopIteration` exception. This is used to indicate that the iterator has finished supplying values.

30.2.3 The Iteration Related Methods

To summarise then we have

- `__iter__()` from the *Iterable* protocol which is used to return the iterator object,
- `__next__()` from the *Iterator* protocol which is used to obtain the next value in a sequence of values.

Any data type can be both an Iterable and an Iterator; but that is not required. An Iterable could return a different object that will be used to implement the iterator or it can return itself as the iterator—it's the designers choice.

30.2.4 The Iterable Evens Class

To illustrate the ideas behind iterables and iterators we will implement a simple class; this class will be an `Evens` class that is used to supply a set of even values from 0 to some limit. This illustrates that it is not only data containers that can be iterable/iterators.

It also illustrates a type that is both an Iterable and an Iterator.

```
class Evens(object):

    def __init__(self, limit):
        self.limit = limit
        self.val = 0

    # Makes this class iterable
    def __iter__(self):
        return self

    # Makes this class an iterator
    def __next__(self):
        if self.val > self.limit:
            raise StopIteration
        else:
            return_val = self.val
            self.val += 2
            return return_val
```

There are a few things to note about this class

- The `__iter__()` method returns `self`; this is a very common pattern and assumes that the class also implements the iterator protocol
- The `__next__()` method either returns the next value in the sequence or it raises the `StopIteration` exception to indicate that there are no more values available.

30.2.5 Using the Evens Class with a for Loop

Now that we have implemented both the *iterable* and *iterator* protocols for the class `Evens` we can use it with a `for` statement:

```
print('Start')
for i in Evens(6):
    print(i, end=', ')

print('Done')
```

Which generates the output:

```
Start
0, 2, 4, 6, Done
```

This makes it look as if the `Evens` type is a built-in type as it can be used with an existing Python structure; however the `for` loop merely expects to be given an *iterable*; as such `Evens` is compatible with the `for` loop.

30.3 The Itertools Module

The `itertools` module provides a number of useful functions that return iterators constructed in various ways. It can be used to provide an iterator over a selection of values from a data type that is iterable; it can be used to combine iterables together etc.

30.4 Generators

In many cases it is not appropriate (or possible) to obtain all the data to be processed up front (for performance reasons, for memory reasons etc.). Instead lazily creating the data to be iterated over based on some underlying dataset, may be more appropriate.

Generators are a *special* function that can be used to *generate* a sequence of values to be iterated over on demand (that is when the values are needed) rather than produced up front.

The only thing that makes a generator a *generator function* is the use of the `yield` keyword (which was introduced in Python 2.3).

The `yield` keyword can only be used inside a function or a method. Upon its execution the function is suspended, and the value of the `yield` statement is returned as the current *cycle* value. If this is used with a `for` loop, then the loop runs once for this value. Execution of the generator function is then resumed after the loop has cycled once and the next cycle value is obtained.

The *generator* function will keep supplying values until it returns (which means that an infinite sequence of values can be generated).

30.4.1 Defining a Generator Function

A very simple example of a generator function is given below. This function is called the `gen_numbers()` function:

```
def gen_numbers():
    yield 1
    yield 2
    yield 3
```

This is a *generator* function as it has at least one `yield` statement (in fact it has three). Each time the `gen_numbers()` function is called within a `for` statement it will return one of the values associated with a `yield` statement; in this case the value 1, then the value 2 and finally the value 3 before it returns (terminates).

30.4.2 Using a Generator Function in a for Loop

We can use the `gen_numbers()` function with a `for` statement as shown below:
Which produces 1, 2 and 3 as output.

```
for i in gen_numbers():
    print(i)
```

It is common for the body of a generator to have some form of loop itself. This loop is typically used to generate the values that will be *yielded*. However, as is shown above that is not necessary and here a `yield` statement is repeated three times.

Note that `gen_numbers()` is a function but it is a special function as it returns a generator *object*.

This is a generator function returns a generator object which wraps up the *generation* of the values required but this is hidden from the developer.

30.4.3 When Do the Yield Statements Execute?

It is interesting to consider what happens within the generator function; it is actually suspended each time a `yield` statement supplies a value and is only resumed when the next request for a value is received. This can be seen by adding some additional `print` statements to the `gen_numbers()` function:

```
def gen_numbers2():
    print('Start')
    yield 1
    print('Continue')
    yield 2
    print('Final')
    yield 3
    print('End')

for i in gen_numbers():
    print(i)
```

When we run this code snippet, we get

```
Start
1
Continue
2
Final
3
End
```

Thus the generator executes the `yield` statements on an as needed basis and not all at once.

30.4.4 An Even Number Generator

We could have used a generator to produce a set of even numbers up to a specific limit, as we did earlier with the `Evens` class, but without the need to create a class (and implement the two special methods `__iter__()` and `__next__()`). For example:

```
def evens_up_to(limit):
    value = 0
    while value <= limit:
        yield value
        value += 2

for i in evens_up_to(6):
    print(i, end=', ')
```

This produces

```
0, 2, 4, 6,
```

This illustrates the potential benefit of a generator over an iterator; the `evens_up_to()` function is a lot simpler and concise then the `Evens` iterable class.

30.4.5 Nesting Generator Functions

You can even nest generator functions as each call to the generator function is encapsulated in its own generator object which captures all the state information needed by that generator invocation. For example:

```
for i in evens_up_to(4):
    print('i:', i)

    for j in evens_up_to(6):
        print('j:', j, end=', ')

    print('')
```

Which generates:

```
i: 0
j: 0, j: 2, j: 4, j: 6,
i: 2
j: 0, j: 2, j: 4, j: 6,
i: 4
j: 0, j: 2, j: 4, j: 6,
```

As you can see from this the loop variable `i` is bound to the values produced by the first call to `evens_up_to()` (which produces a sequence up to 4) while the `j` loop variable is bound to the values produced by the second call to `evens_up_to()` (which produces a sequence of values up to 6).

30.4.6 Using Generators Outside a for Loop

You do not need a `for` loop to work with a generator function; the generator object actually returned by the generator function supports the `next()` function. This function takes a generator object (returned from the generator function) and returns the next value in sequence.

```
evens = evens_up_to(4)
print(next(evens), end=', ')
print(next(evens), end=', ')
print(next(evens))
```

This produces

```
0, 2, 4
```

Subsequent calls to `next(evens)` return no value; if required the generator can throw an error/exception.

30.5 Coroutines

Coroutines were introduced in Python 2.5 but are still widely misunderstood.

Much documentation introduces Coroutines by saying that they are similar to Generators, however there is a fundamental difference between Generators and Coroutines:

- generators are data producers,
- coroutines are data *consumers.*

That is coroutines *consume* data produced by something else; where as a generator produces a sequence of values that something else can process.

The `send()` function is used to send values to a coroutine. These data items are made available within the coroutine; which will wait for values to be supplied to it. When a value is supplied then some behaviour can be triggered. Thus, when a coroutine consumes a value it triggers some behaviour to be processed.

Part of the confusion between generators and coroutines is that the `yield` keyword is reused within a coroutine; it is used within a coroutine to cause the coroutine to wait until a value has been sent. It will then supply this value to the coroutine.

It is also necessary to *prime* a Coroutine using with `next()` or `send(None)` functions. This advances the Coroutine to the call to `yield` where it will then wait until a value is sent to it.

A coroutine may continue forever unless `close()` is sent to it. It is possible to pick up on the coroutine being closed by catching the `GeneratorExit` exception; you can then trigger some shut down behaviour if required.

An example of a coroutine is given by the `grep()` function below:

```
def grep(pattern):
    print('Looking for', pattern)
    try:
        while True:
            line = (yield)
            if pattern in line:
                print(line)
    except GeneratorExit:
        print('Exiting the Co-routine')
```

This coroutine will wait for input data; when data is sent to the coroutine, then that data will be assigned to the `line` variable. It will then check to see if the pattern used to initialise the coroutine function is present in the `line`; if it is it will print the `line`; it will then loop and wait for the next data item to be sent to the coroutine. If while it is waiting the coroutine is closed, then it will catch the `GeneratorExit` exception and print out a suitable message.

The `grep()` coroutine is used below, notice that the coroutine function returns a coroutine object that can be used to submit data:

```
print('Starting')
# Initialise the coroutine
g = grep('Python')

# prime the coroutine
next(g)

# Send data to the coroutine
g.send('Java is cool')
g.send('C++ is cool')
g.send('Python is cool')

# now close the coroutine
g.close()
print('Done')
```

The output from this is:

```
Starting
Looking for Python
Python is cool
Exiting the Co-routine
Done
```

30.6 Online Resources

See the following for further information

- https://docs.python.org/3/library/stdtypes.html#iterator-types for iterator types.

30.7 Exercises

These exercises focusses on the creation of a generator.

Write a *prime number* generator; you can use the prime number program you wrote earlier in the book but convert it into a generator. The generator should take a *limit* to give the maximum size of the loop you use to generate the prime numbers. You could call this `prime_number_generator()`.

You should be able to run the following code:

```
number = input('Please input the number:')
if number.isnumeric():
    num = int(number)
    if num <= 2:
        print('Number must be greater than 2')
    else:
        for prime in prime_number_generator(num):
            print(prime, end=', ')
else:
    print('Must be a positive integer')
```

If the user enters 27 then the output would be:

```
Please input the number:27
2, 3, 5, 7, 11, 13, 17, 19, 23,
```

Now create the `infinite_prime_number_generator()`; this generator does not have a limit and will keep generating prime numbers until it is no longer used.

You should be able to use this prime number generator as follows:

```
prime = infinite_prime_number_generator()
print(next(prime))
print(next(prime))
print(next(prime))
print(next(prime))
print(next(prime))
```