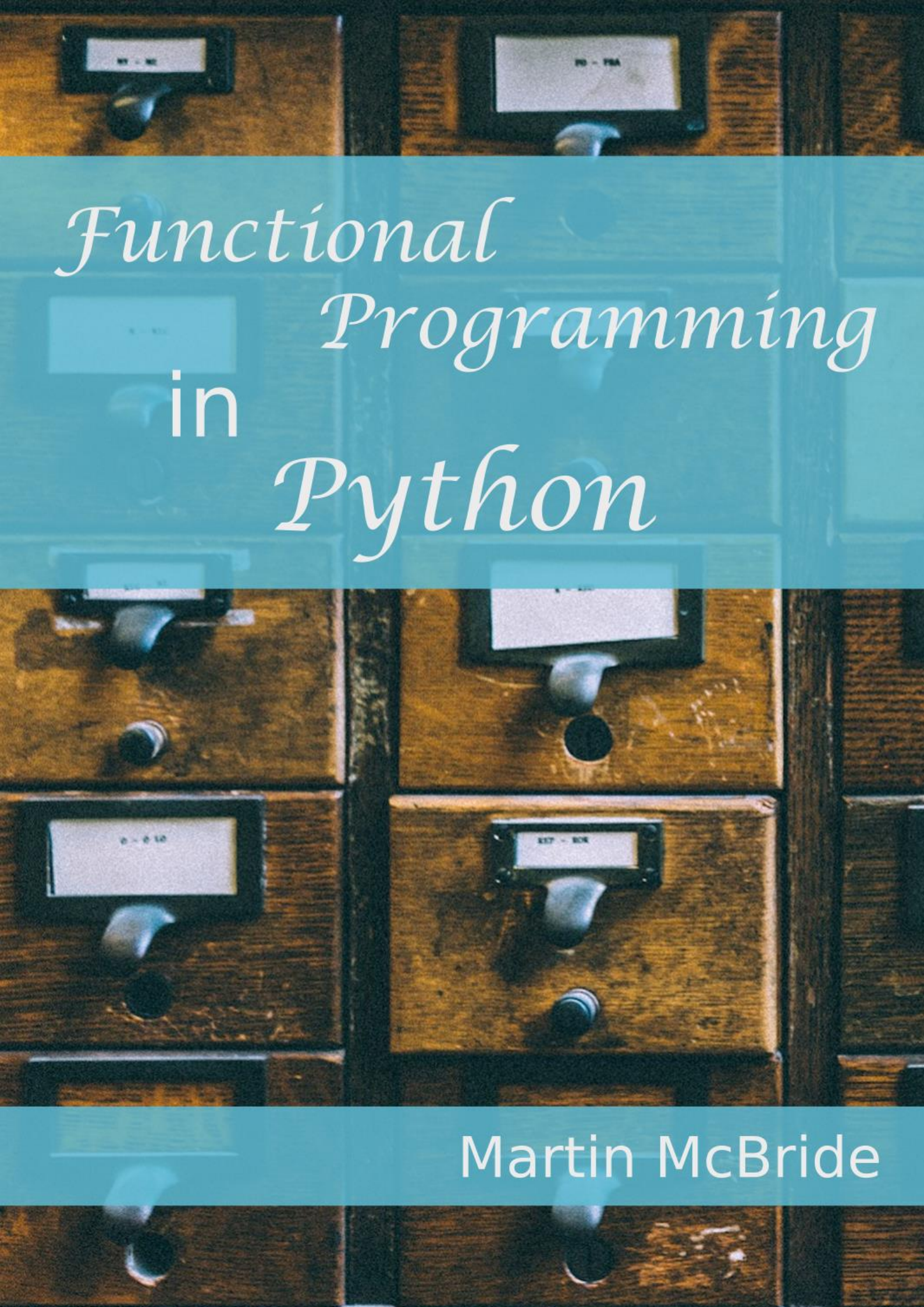

Functional
Programming
in
Python
Martin McBride

Functional Programming in Python
by Martin McBride

Published by Axlesoft Ltd
info@axlesoft.com

Visit pythoninformer.com for more details
@pythoninformer

Contents

1 Introduction

Python supports several programming *paradigms* – procedural, object oriented, and functional. Of these, functional programming is probably the least understood and the least used. But it can be a powerful tool, especially as it can be integrated seamlessly with procedural and OOP code.

This book explains what functional programming is, how it is used, and the features of Python that support it. All features are illustrated with example code.

No prior knowledge of functional programming is assumed, and you don't need to be an advanced Python programmer to use this book. Any language features used are fully described. All that is required is a basic knowledge of Python.

The examples are developed for Python 3.5 or higher, although most will work with earlier 3.x versions too.

1.1 Programming paradigms

A programming paradigm is a general approach to developing software. There aren't usually fixed rules about is or isn't part of a particular paradigm, but rather there are certain patterns, characteristics and models that tend to be used. This is especially true of Python, since it supports several paradigms with no real dividing line between them. Here are the paradigms available in Python:

Procedural programming is the most basic form of coding. Code is structured hierarchically into blocks (such as if statements, loops and functions). It is arguably the simplest form of coding. However, it can be difficult to write and maintain large and complex software due to its lack of enforced structure.

Object oriented programming (OOP) structures code into *objects*. An object typically represents a real item in the program, such as a file or a window on the screen, and it groups all the data and code associated with that item within a single software structure. Software is structured according to the relationships and interactions between different objects. Since objects are encapsulated, with well-defined behaviour, and capable of being tested independently, it is much easier to write complex systems using OOP.

Functional programming (FP) uses functions as the main building blocks. Unlike procedural programming, the functional paradigm treats functions as objects that can be passed as parameters, allowing new functions to be built dynamically as the program executes.

Functional programming tends to be more *declarative* rather than *imperative* – your code defines what you want to happen, rather than stating exactly how the code should do it. Some FP languages don't even contain constructs such as loops or if statements. However, Python is more general purpose and allows you to mix programming styles very easily.

1.2 What is functional programming?

Since functional programming is a paradigm, there are no absolute rules about what it is or is not. If you had to summarise it in one sentence it might be that *functional programming use functions as the fundamental building block for constructing software*.

You might also see it said that *functional programming treats functions as first class objects*. This means that functions are objects, just like lists or strings, that can be stored in variables, passed into other functions as parameters, returned from other functions as a result. This leads to the idea of higher order functions – that is, functions that operate on functions. Anything you can do with objects, you can do with functions.

An important cornerstone of functional programming is the idea of pure functions – functions that simply calculate a result without any other side effects.

1.3 Characteristics of functional programming

Rather than trying to precisely define functional programming, it is more useful to look at some of its characteristics – the sort of techniques functional programmers typically use.

FP prefers pure functions. A pure function is a function that calculates a result without any side effects, or any possibility of an unexpected result. For example, these are all pure functions:

- Adding two values.
- Calculating the square root of a number.
- Finding the length of a string.
- Returning a sorted copy of a list of items.

Functions that either change or rely upon external state are not pure. For example, functions that do any of these things are not pure functions:

- Sets a global variable
- Writes to a file or database.
- Modifies the value of a parameter that has been passed in.

Pure functions are only allowed to return a value, they are not allowed to alter the state of the system on any other way. Clearly the actions above change the state of the system in various ways.

In addition, a pure function must return a value that depends only on its input parameters. It must be totally repeatable, for given inputs it must always produce the same output. A function that reads from a global variable, file or database, or accepts user input, for example, is not repeatable and so not pure.

FP avoids side effects. This is really an alternative version of the previous characteristic that you will often see stated.

Functions are first class objects. As mentioned above, in FP a function is an object that can be stored in a variable, and passed as an argument to a function, or returned as the result of a function.

FP prefers immutable objects. Immutable objects, such as strings and tuples in Python, are objects that cannot be modified after they have been created. Immutability helps to prevent side effects in functions. For example, if you pass a list into a function, it is possible for the function to alter it. If you pass a tuple into a function, that is impossible.

FP prefers iterators over lists. An iterator is an object that provided access to a collection of data. An iterator can only read data one element at a time, it has no ability to change the data. This helps to prevent side effects and often avoids needing to store intermediate results at all via *lazy evaluation*. We often talk of the output of an iterator as being a stream of data.

FP favours lazy evaluation. A traditional procedural function that processes a list of data will typically process the entire list in one call. An iterator will often choose to calculate new values only as they are needed – this is called lazy evaluation. It often reduces the amount of memory used and allows the program to start creating output with less initial delay.

FP avoids loops and if statements. Rather than using a loop to process a list of data, FP tends to use higher order functions such as `map` that apply a function to an iterable data stream, converting it into a new data stream. Similarly, it uses functions such as `filter` to conditionally remove items from a stream of data.

FP often uses recursion to avoid loops. Recursion is a useful alternative to looping for certain algorithms.

FP uses higher order functions to define new functions. Procedural programming often defines new functions that call other functions to perform a task. In functional programming we tend to use higher order functions that modify or combine existing functions to create new functions.

1.4 Advantages of functional programming

Here are the main advantages of functional programming:

FP often creates less code. This is because it tends to work at a slightly higher level than the other paradigms, so achieves more with each line of code.

Intent of the code is clearer. For example, if you use `map` to apply a function to a data stream, the meaning is clear and unambiguous. If you define a procedural function that loops over the data and applies the function, you need to read and understand the code to check exactly what it is doing.

There are often fewer bugs. Using standard functions that are well tested, instead of ad hoc loops that might contain bugs is generally more reliable.

Code is potentially mathematically provable. If you program consists entirely of predefined functions that are known to be correct, and you combine those functions using higher order functions that are also known to be correct, and if you have eliminated all side effects, then it is possible, at least in principle, to prove that your code will be correct in all cases.

Multiprocessing can be applied easily. For example, if you are applying a pure function to a data stream, you can safely split that data stream into several blocks and process each block in a different thread, or even on a different computer, and in any order. The map-reduce pattern does this very effectively. If you have a procedural program that works on lists of data, multiprocessing can often be more difficult and error prone.

1.5 Disadvantages of functional programming

Functional programming has a few disadvantages, and situations where it cannot be used.

Not all functions can be pure. Most programs need to read and write files, communicate over a network, interact with users and other such things. The functions that do those tasks are not pure functions with totally predictable results.

A common way of handling this is to split the code into those parts than can be developed using a functional approach (commonly any complex algorithms or heavy data processing) and those parts that require a procedural approach. There should be a clear interface between the two. The non-pure parts of the system can be developed, for example, using an OOP paradigm.

Pure functional languages, such as Haskell, use monads and similar constructs to deal with impure functions. This is less commonkly used in Python, but we will cover th in a later chapter.

FP has a learning curve. It is probably true to say that there are far fewer programmers who are experienced in functional programming than some other paradigms. It is a conceptual leap to move from the idea of writing a function to do x to the idea of writing a function that creates a function to do x. FP has its own jargon, largely drawn from fairly obscure branches of mathematics, so you will need to learn terms such as lambda expression, closure, partial function, currying, comprehension, monad and functor. But none of it is as complicated as it sounds!

FP can be inefficient. In particular, immutable objects and recursion are very useful concepts, and in many cases they can be used without problem, but they can be inefficient in extreme cases. As well as thinking about functional programming in abstract terms, it is necessary to keep in mind what you are asking the poor computer to actually do. It is worth doing a sanity check for very large problems. See the example later of the recursive implementation of Fibonacci series.

1.5.1 About this book

In the remainder of this book we will introduce the various aspects of Python that are either directly to indirectly relevant to functional programming, with examples of their application:

- Objects, variables, and functions as objects.
- Immutable objects.
- Recursion.
- Closures.
- Iterators.
- Transforming and reducing iterables.
- Comprehensions.
- Generators.
- Partial application and currying.
- Functors and monads.
- itertools, functools and other useful libraries.

2 Functions as objects

As noted in the introduction, Python functions are first class objects. This means that functions are objects that can be stored in variables, referenced in lists or other data structures, and passed in and out of functions as parameters and return values. We will explore this in a bit more detail in this chapter.

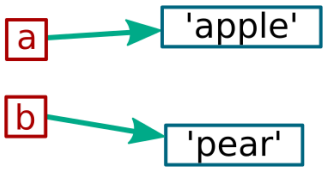
2.1 Objects and variables in Python

Before we talk about function objects, it is worth quickly recapping how objects and variables work in Python in general.

Consider the following simple line of Python:

```
a = 'apple'
b = 'pear'
```

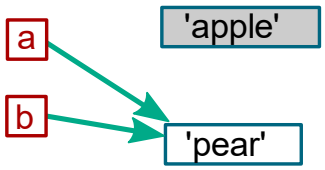
Now you might loosely say that the string `'apple'` is stored in the variable `a`, and the string `'pear'` is stored in the variable `b`. But that isn't quite correct. In fact, the strings are both objects that Python stores in memory somewhere. The variables `a` and `b` simply hold references to those objects – they point to those objects in memory. This diagram illustrates this:



This is quite important, because of what happens when we do this:

```
a = b
```

Now `a` and `b` both reference a string with the value `'pear'`, but the important this to realise is that `a` and `b` actually both reference the same object in memory:



The previous `'apple'` string is still in memory, but you can no longer access it in any way. Python will eventually free up the memory it occupies, so it can be used for something else.

We will come back to this in a later chapter on immutable data types.

2.2 Storing functions

When you look at the way a variable is initialised and used in Python, and compare it to the way a function is declared and used, you might easily assume that variables and functions are completely different things:

```
a = 10

def square(x):
    return x*x

b = square(a)
print(b)          # 100
```

Looking at the code, variables `a` and `b` are initialised by assigning a value to them, and used by directly referencing them, as in `print(b)`. Whereas the function `square` is created by the `def` keyword and invoked using round brackets `()`.

In fact, `a`, `b` and `square` are all just variables. The `def` block is just special syntax for defining a function object and assigning it to a variable (`square` in this case). The round brackets are a syntax that can be used with any *callable object* (which includes functions) to call it with parameters.

To further illustrate this, the following code treats square as an object and prints out its type, id and string representation, just like you can do with any other object:

```
print(type(square))
print(id(square))
print(str(square))
```

You will see that the `type` of the object is `<class 'function'>`. The `id` is just some number that is unique to that object. And its `str` representation is `<function square at XXX>`, where `XXX` is its address in memory. In other words, `square` behaves much like any other object.

2.3 Aliases

Aliasing is when two different variables reference the same object in Python. For example, consider this code:

```
t = (10, 20, 30, 40, 50)
u = t
print(t[2])   # 30
print(u[2])   # 30
```

We assign a tuple value to variable `t`. This means that `t` holds a *reference* to the tuple object. The tuple itself is stored in memory somewhere.

When we set `u = t`, we are actually copying the reference into the variable `u`. We don't create a copy of the actual tuple itself. There is only one tuple, but both `t` and `u` point to it, so we call then aliases – different names for the same data. When we then print `t[2]` and `u[2]`, they both refer to element index 2 in the original tuple.

In the earlier example, we saw that `square` is just a variable that holds a reference to a function object – a function that calculates the square of `x`. We can create an alias for that, too:

```
def square(x):
    return x*x

sq = square

a = 3
print(sq(a))    # 9
```

In this case, `sq` can be used in place of `square`, doing exactly the same thing, because they both point to the same underlying object – a function object.

This also works with built in functions. For example, you could create an alias of print, like this:

```
pr = print
pr('This is an alias')
```

Just because you can, doesn't mean you should, of course! This might seem like a great way of shortening your code if you use a lot of print statements, but it is likely to be quite confusing to anyone reading it.

In fact, you are quite unlikely to use aliases directly in your code. But you will use them indirectly quite often. In the previous example with `square`, we pass the variable `a` into `square`, but within the function it is aliased as `x`. In the next section we will look at passing *functions* into other functions as parameters, and they will be aliased in a similar way. This is the essential feature of Python that makes functional programming possible at all.

2.3.1 Redefining a function

Since functions are essentially variables that happen to hold function objects, you can reassign them at any time:

```
def a():
    print(1)

def a():
    print(2)
```

Python has no problem with this. But it has consequences, and generally is best avoided. Here is a simple example of what can happen:

```
def a():
    print(1)

def b():
    a()

b()    # 1

def a():
    print(2)
b()    # 2
```

We have defined a function a that prints 1. We then define function b that calls function a that prints 1. When we call b for the first time, it prints 1 as expected.

Next, we redefine a to print 2 instead. What happens when we call b again?

Well, as far as function b is concerned, a is just a global variable. It looks up the value of a, which is a function object. In fact, of course, it is now the function that prints 2. b calls that function, and 2 is printed.

The pitfall here is that you have changed the behaviour of function b without it being particularly obvious what has happened, which is a recipe for bugs. It is rarely a good thing to do.

2.4 Functions as parameters

Consider this function that converts inches to centimetres and prints the result. One inch is 2.54 cm, so the conversion is a simple multiplication:

```
def inch2cm(x):
    return x*2.54

def convert(x):
    y = inch2cm(x)
    print(x, '=>', y)

convert(3)    # 3 => 7.62
```

Suppose we wanted to generalise this function so that it could convert between different units. There are various ways to do this, but one way would be to remove the explicit call to inch2cm from the convert function. Instead, we could pass the function as a parameter, like this

```
def convert(f, x):
    y = f(x)
    print(x, '=>', y)

convert(inch2cm, 3)    # 3 => 7.62
```

Notice that the function is passed in as a normal parameter, `f`. When we need to call `f` to do the conversion, we just use `f(x)` exactly like any other function.

When we call `convert`, we need to pass `inch2cm` in as the first parameter. We use the syntax `inch2cm` to pass the function object, rather than `inch2cm()` which would try to *call* the function (which isn't what we want at all).

Now supposed we wanted to convert a temperature from Celsius to Fahrenheit. We can write a `c2f` function that does this:

```
def c2f(x):
    return x*1.8 + 32
```

To use this conversion, we just need to pass `c2f` into the `convert` function:

```
convert(c2f, 18)    # 18 => 64.4
```

Just as a final illustration, we will add a conversion from integers to text – 1 becomes "one", 2 becomes "two" etc. Here is our i2text function, which for brevity only works for values up to 0 to 3. It uses a list to convert integers to text:

```
def i2text(x):
    text = ['zero', 'one', 'two', 'three']
    return text[x]

convert(i2text, 2)    # 2 => two
```

The interesting thing here is that `i2text` doesn't use the same types as the previous functions. It accepts and integer and returns a string, whereas the `inch2cm` and `c2f` accept and return numerical values. The convert function doesn't mind this at all – it just passes the value to supplied function and returns whatever comes back.

This was a very simple example, now we will look at a more realistic example.

2.4.1 The sorted function

You may be familiar with the Python built in `sorted` function. It can be used to return a sorted copy of a list, like this:

```
p = [3, 7, 2, 6, 1]
q = sorted(p)
print(q)        # [1, 2, 3, 6, 7]
```

The `sorted` function uses standard Python comparisons to order the list, so in this case it sorts the numbers in increasing order. If the list contains strings, they will be sorted in alphabetical order instead:

```
p = ['red', 'green', 'blue', 'yellow', 'cyan']
q = sorted(p)
print(q)     # ['blue', 'cyan', 'green', 'red', 'yellow']
```

What if we wanted to sort the strings in a different way – for example, if we wanted to sort the keys in ascending length? Fortunately, the `sorted` function takes an optional parameter `key` that allows for this.

The `key` parameter accepts a function object as a value. The function is applied to each element is the list, and the list is sorted based on the return value.

If we want to sort a list of string by increasing length, we need to use a function that accepts a string and return the length of the string. Fortunately, we already have such a function – the built in `len` function. Here is a new version of the code, where we pass in the `len` function as the value of the `key` parameter:

```
p = ['red', 'green', 'blue', 'yellow', 'cyan']
q = sorted(p, key=len)
print(q)    # ['red', 'blue', 'cyan', 'green', 'yellow']
```

This works exactly as we had hoped. 'red' is first in the list because its length is 3, 'blue' and 'cyan' are next with length 4, 'green' with length 5 and finally 'yellow'.

Of course, we don't always have a convenient built in function that does exactly what we need. Sometimes we have to define our own. In the example below we have a list of rectangles, defined by a pair of values `(width, height)`. For example (3, 2) defines a rectangle that is 3 units wide by 2 units high. We wish to sort then by increasing area. To do this, we need a key function that multiplies the width by the height, such as the `area` function below:

```
def area(x):
    return x[0]*x[1]

p = [(3, 3), (4, 2), (2, 2), (5, 2), (1, 7)]
q = sorted(p, key=area)
print(q)    # [(2, 2), (1, 7), (4, 2), (3, 3), (5, 2)]
```

Each tuple will be passed into the `area` function. This function multiplies elements 0 and 1 of the tuple (the width and height) to give the area. The area is then used as the sort criterion. As you can see from the result, this sorts the rectangles in order of area.

We will cover `sorted` in more detail in the chapter *Transforming iterables*.

2.5 Lambda functions

Lambda functions sound like they are going to be something complicated, but in fact they really are very simple.

In the example above, we needed to create a function called `area`. This is a very small function, that will probably only be used in one place. There has to be a better way, surely?

Well there is. You can use lambda syntax to create a simple function in a Python expression. Here is how we could replace our `area` function:

```
lambda x: x[0]*x[1]
```

The `lambda` keyword identifies the lambda expression. `x` is the parameter (in this case there is only one parameter). The colon ends the parameter list and introduces the body of the function.

To use this expression, simply place it wherever you might normally use a function object. For example:

```
q = sorted(p, key=lambda x: x[0]*x[1])
```

This code creates a temporary, anonymous function object and passes it into the sorted function. The sorted function uses it to perform the sort. And then it's gone, just like any other temporary object.

The unnamed function you create with a lambda expression is exactly the same as a function created with `def`, it just doesn't have a name. If you really wanted to you could assign it to a variable, like this:

```
area = lambda x: x[0]*x[1]
```

This creates a function called `area`. It is more or less the same as creating an `area` function with `def`. There isn't really any point doing it this way, however, it will just confuse anyone reading it.

A lambda expression can have any number of arguments (including none), for example:

```
lambda: 1  # No arguments
lambda x, y: x + y
lambda a, b, c, d: a*b + c*d
```

You will probably find yourself using lambda expressions quite often when using functional programming. Like many aspects of Python, they can be expressive and make code shorter and more readable – or they can make for impossibly cryptic code. It is all a matter of balance. Here are some guidelines:

- Lambdas can only contain a single Python expression. If your function cannot be expressed in one line, you can't use a lambda.
- Generally, it is best to use them only for short and simple code, where the behaviour of the function is obvious by looking at it. If the behaviour is complicated, it is usually best to define a normal function so you can give it a meaningful name and add comments.
- Since a lambda expression will usually be used as part of a longer line of code, make sure that overall the code is still readable. If a function call uses several lambda expressions, it might be difficult to see what is going on.
- If the same function is used in several places, it is usually better to define a normal function, rather than repeating the lambda.

Although these criteria might seem restrictive, you will find there are many situations where a lambda is the perfect fit for what you need to do.

By the way, since a lambda is a function object, you can call it in-place like this:

```
a = (lambda x: x + 1)(3)
```

The lambda expression creates a function object that adds 1 to its argument. The `(3)` calls the function object with value 3, so `a` is set to 4. This isn't a particularly useful feature, because you could just write:

```
a = 3 + 1
```

This does exactly the same thing, so it isn't really of any practical use. But it illustrates that a lambda expression can replace a normal function is all situations.

2.6 Functions as return values

You can return a function as a value. Here is a simple example:

```
def add1():
    return lambda x: x + 1

f = add1()
print(f(2))
```

Here, `add1` returns a function that accepts a single argument and adds 1 to it. This isn't particularly useful, of course, we could just use the lambda. Things get a lot more interesting in the chapter *Closures*.

2.7 Function versions of standard operators

The standard `operator` module contains a set of functions that are equivalent to Python operators. For example:

```
x = operator.add(a, b)      # Equivalent to x = a + b
x = operator.truediv(a, b)  # Equivalent to x = a / b
x = operator.floordiv(a, b) # Equivalent to x = a // b
```

These are very useful functions that can often be used to replace lambda expressions. For instance, the earlier example:

```
lambda x, y: x + y
```

Could simple be replaced with `add` – a function that takes two values and adds them together (exactly what the lambda in doing). Using a standard function is shorter and more declarative.

You can also use *partial application* to create new functions based on existing operators. For example:

```
from functools import partial
f = partial(add, 3)
x = f(4)                # Equivalent to x = 3 + 7
```

In this case, `partial` creates an anonymous function that takes one variable. It behaves like `add`, but as if the first parameter had been pre-set to 3. In other words, it is equivalent to the following lambda:

```
f = lambda x: 3 + x
```

We will cover partial application in more detail in a later chapter.

The operator module doesn't just include arithmetic operators. Here are a few more examples but refer to the documentation on python.org for a full list. Essentially, for anything you can do with an operator there will be a function that does the same thing:

```
operator.lt(a, b)           # a < b
```

```
operator.eq(a, b)          # a == b
operator.not(a)            # not a
operator.neg(a, b)         # -a
operator.getitem(s, i)     # s[i]
operator.setitem(s, i, x)  # s[i] = x
operator.delitem(s, i)     # del s[i]
```

operator also defines a few useful functions that return functions. For example, `itemgetter` returns a function that works like this:

```
k = [2, 4, 6, 8]
f = operator.itemgetter(2)
x = f(k)   # x = 6
```

Here, `itemgetter(2)` returns a function that will get element number 2 from a list. When we apply this function to list `k`, it gets he second element, value 6. There are similar functions to get a named attribute (`attrgetter`) and call a named method (`methodcaller`). These are particularly useful for use as the `key` argument for the `sorted` function. They will be described in more detail in the chapter *Transforming iterables*.

2.8 Summary – sources of function objects

To summarise, here are the various ways you can obtain function objects to use in your code. Some of these we have just met:

- Built in functions, such as `len`, `min`, `abs` etc. Remember that, for example, `len(s)` calls the `len` function to find the length of `s`, but `len` on its own gives the actual function object.
- The `operator` module contains function versions of most Python operators, for example add is the function equivalent of +.
- Lambda expressions can be used to create simple, unnamed functions.
- We can, of course, create new functions the standard way, using `def`.

Here are some more possibilities that we will explore in later chapters:

- Composition can be used to create a new function by combining two or more existing functions that call each other, for example `f(g(x))`.
- Partial application can be used to create a new function based on an existing function with some of its parameters already applied.
- Currying is an alternative way to achieve similar results to partial application.
- Closures can be used as general function factories if no other method provides quite what you need.
- Objects that implement `__call__` can be used as function objects.

3 Mutability

We say an object is *mutable* if its value can be changed after it has been created. An object is *immutable* if its value is fixed when it is created and can never be changed. Immutable values are, in effect, read-only values. In this chapter we will look mutability in Python, and the pros and cons of both types of objects.

3.1 Mutability in Python

In Python, it is usually an object's type that tells you whether it is mutable or not. For example, lists are mutable. If you create a list, you can change it in various ways:

```
k = [10, 20, 30]
k[1] = 7          # k is now [10, 7, 30]
k.append(5)       # k is now [10, 7, 30, 5]
del k[2]          # k is now [10, 7, 5]
```

There are many other ways to alter a list, we won't go through them here, it is fairly standard Python.

Tuples are similar to lists in almost every way, except that they are immutable. Once you have created a tuple, you cannot change it in any way – you can't add or remove elements, and you can't replace one of the elements with a different value. Here is what happens if you try the previous code with a tuple:

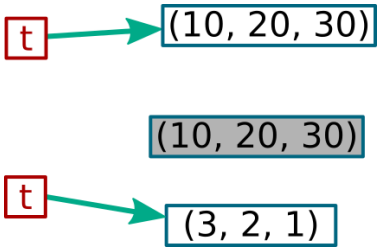
```
t = (10, 20, 30)
t[1] = 7        # TypeError tuple doesn't support assignment
t.append(5)     # AttributeError tuple has no append method
del t[2]        # TypeError tuple doesn't support deletion
```

You simply can't do those operations on a tuple. All of the operations you can use to modify a list simply don't work in tuples, so it is impossible to ever change a tuple.

One thing to remember, of course, is that the variable is not the object – variables only hold a reference to an object. So, it is perfectly ok to do something like this:

```
t = (10, 20, 30)
t = (3, 2, 1)
```

After the first line, t holds a tuple (10, 20, 30). After the second line, t holds a tuple (3, 2, 1).



But we haven't changed the value of the tuple. We have created a brand-new tuple with a different value, and simply changed the variable to reference the new tuple. The original tuple is

still there, unchanged, but since nothing is using it any more it will eventually be garbage collected.

There are several other mutable data types, including dictionaries and sets. There are also several immutable data types including strings, frozensets (the immutable version of a set), and number types.

Mutable data types	Immutable data types
list, set	tuple, string, frozenset, int, float, complex

3.2 Numbers are immutable

As an aside, you might be initially surprised to see that numbers are immutable. That is because numbers are objects in Python, and if they were mutable some very strange things would happen. Imagine this code:

```
a = 3
b = a
a = 4
```

Here, the first line creates an int object, with value 3, and assigns it to variable `a` (variable `a` references the int object). The second line assigns `a` to `b`, so now variable `b` also references the same int object.

Now what happens when we set `a` to 4 in the third line? If this operation changed the value of the int object, not only would `a` be equal to 4, `b` would be equal to 4 as well! That would be pretty disastrous. Fortunately, numbers are immutable, so what the third line actually does is create a new int object, with value 4, and assign it to `a`. Variable `b` still points to the original int object, which still has the value 3.

3.3 The problem with mutable objects

The basic problem with mutable objects is this: if you pass a mutable object into a function, you have no way of guaranteeing that the function won't change the object. For example:

```
def evil_func(x):
    x[0] = 0

k = [1, 2, 3]
evil_func(k)
print(k)     # [0, 2, 3]
```

When you pass `k` into `evil_func`, the local variable `x` is given a reference to the same list that is stored in `k`. if the function does something to the list in `x`, it is actually doing it to the list in `k`. When you pass a list into `evil_func`, you have no control over what happens to that list.

So, in the example above, even though you might think you are just passing `k` into the function, `k` actually gets changed by the call!

Sometimes, of course, you might actually want a function to alter the list. You might pass the list into the function expecting it to do something useful to it, for example sorting it in place – though in fact by the end of this chapter you might prefer to design your function to return a sorted *copy* of the input data.

But let's assume that `evil_func` has no obvious reason to alter the list you pass it, so you are *hoping* it won't change it. That is ok if the function is very simple, and part of the same module. At least, until you or someone else edits the function in the future and doesn't realise that `evil_func` isn't supposed to change `x`, or creates a bug that means `x` is changed accidentally.

But what if `evil_func` is part of a third-party library, and you have no control of the source code. Then you are essentially trusting the developers of that library to not change `x`. Worse still, the developers of that library might pass your list into another function, `evil_func2`, in someone else's library, that also isn't supposed to change the list. So, you are not just trusting the developers of one library, you are also trusting everyone that they trust.

3.3.1 Defensive copying

One way around this is defensive copying. Rather than passing your list into a function, you pass a copy of your list into the function:

```
def evil_func(x):
    x[0] = 0

k = [1, 2, 3]
evil_func(list(k))
print(k)     # [1, 2, 3]
```

The key change here is that the call to `evil_func` passes `list(k)`. The `list` function creates a copy of the original list, so now `x` is a copy of `k`. So, no matter `evil_func` does to its list, nothing is going to happen to your list `k`.

This solution isn't completely terrible, but it has its downsides:

- You have to remember to do it.
- If the list is very big, you are creating an extra copy which may be a waste of time if the function is well written and doesn't in fact corrupt the list.
- It can get out of hand...

On the last point, if function1 calls function2 calls function3, and each function makes a defensive copy, you can end up with the same data being copied many times.

Worse still, some authors make a defensive copy of data passed into the function, so that if they accidentally alter the data, the caller is protected:

```
def evil_func(x):
    xcopy = list(x)
    xcopy[0] = 0
```

So now the data is copied twice every time a function is called!

3.4 Immutability is the answer

The basic solution to this problem is, wherever possible, to use immutable data objects. The first thing we need to do is change the definition of `evil_func`. We should specify that `x` is immutable (or more precisely, that `x` is allowed to be immutable). So, you are allowed to pass in a tuple instead of a list and the function should still work.

Here is the new code, based on the assumption that **`x`** can be a tuple:

```
def evil_func(x):
    x[0] = 0

t = (1, 2, 3)
evil_func(t)
print(t)     # (1, 2, 3)
```

This time, rather than corrupting your tuple (which would be impossible anyway because tuples are immutable), `evil_func` will throw an exception. Exactly as it should because it is doing something illegal by trying to alter an object that is allowed to be immutable.

3.5 Changing immutable objects

We do sometimes need to "change" immutable objects. Of course, you can't actually do that, but what you can do is create a copy of the original object, modified in some way. There are various ways of doing this, which we will explore here.

Let's start with a simple example. The `tail` function takes a list and returns a list that is identical except that the first term is removed. So `[1, 2, 3]` becomes `[2, 3]`. Here is how we might do this:

```
def tail(x):
    if x:          # If x is already empty do nothing
        del x[0]

k = [1, 2, 3]
tail(k)
print(k)     # [2, 3]
```

This only works for lists. The list is passed into the function and modified in place. But for reasons we discussed previously, this will fail if we pass a tuple as the parameter.

What if we wanted to make this function work with tuples as well as lists? In that case, we can't modify the supplied argument, so instead we make our function return a modified tuple:

```
def tail(x):
    return x[1:]

t = (1, 2, 3)
t = tail(t)
print(t)     # (2, 3)
```

This does the same job as the list case above, but in a cleaner way. The way we call the function has changed, we now assign the return value back to `t`.

Notice how the function now operates. It uses slice notation to create a new tuple, containing only the elements from 1 to the end of the original tuple. Slice notation is very neat, but don't let it disguise the fact that we are creating a copy of the tuple. If the tuple is very long, and if we do this operation many times, there would be a performance hit both in terms of execution times and memory usage. But unless the tuple really is extremely large, that shouldn't be anything to worry about.

A bonus of this change is that the new function not only works with tuples and lists. it works with strings too! We use the term *sequence* to refer to lists, tuples, strings and similar data structures.

There are several ways to process immutable data, which we will look at now.

3.5.1 Using slices

Slices provide a very versatile way to chop sequences up into parts. Those parts can then be reassembled using the + operator. Here are a couple of examples. To add an element 3 into the middle of a tuple at position `n`, you can do something like this:

```
u = v[:n] + (3) + v[n:]
```

Alternatively, to remove the element at position `n` from a tuple you can do this:

```
u = v[:n] + v[n+1:]
```

The only thing to bear in mind here is that you are creating several copies of the tuple. You will be creating a temporary copy of `v[:n]`, a temporary copy of `v[n+1:]`, and of course the final tuple `u`. This shouldn't be an issue unless the tuples are very large.

This technique will also work with string values. To add a letter 'a' into the middle of a string, you can do this:

3.5.2 Using list comprehensions

Sometimes you need to perform a simple operation on each element of a sequence. For example, suppose you want to add 1 to each element in a tuple. So `(1, 5, 7)` becomes `(2, 6, 8)`. A list comprehension is a great way to do this:

```
u = [x + 1 for x in v]
t = tuple(u)
```

A list comprehension can take any type of sequence as input, but always creates a list. The example converts the list back into a tuple.

3.5.3 Using a loop

Suppose you wanted to duplicate all the zeros in a tuple, so `(1, 0, 2, 0, 5)` becomes `(1, 0, 0, 2, 0, 0, 5)`. You can't easily do that with a list comprehension, so a simple loop can be used instead:

```
u = []
for x in v:
    u.append(x)
    if x == 0:
        u.append(x)
t = tuple(u)
```

3.5.4 Converting the data to a list

If you need to do some particularly complicated processing of a tuple, you can always convert it to a list, do what you need to do, then convert it back to a tuple.

3.6 The problem with immutable objects

So, although immutable objects solve a lot of problems with accidental modification of data as it is passed around in a program, that comes at a cost:

- You may need to jump through a few hoops if you need to process the data.
- You may end up making several copies of the data.

In many cases it is worth using immutable data wherever possible for the sake of robustness. The main exception is if you are processing very large data structures. In that case, making a full copy of the data every time you make any change is just not practical, and you are better off using mutable data, and simply taking extra care about when and where it is modified.

In the rest of this chapter, we will look at some other considerations surrounding immutable objects.

3.7 Immutability is shallow

If you are dealing with more complex data structures, it is important to understand exactly what we mean by immutability.

Consider the case of a tuple that contains several lists:

```
t = ([1, 2], [4, 6], [5, 9])
```

You can access this data in various ways:

```
print(t[1])    # [4, 6]
print(t[2][1]) # 9
```

The first print statement access `t[1]`, the second element of `t`, which is of course the list `[4, 6]`. The next print statement accesses `t[2][1]`. Of course, `t[2]` is the third element if `t`, the list `[5, 9]`. This means that `t[2][1]` is the second element if that array, which is 9.

But what happens if we try to update these values:

```
t[1] = 0      # Error
t[2][1] = 0   # t becomes ([1, 2], [4, 6], [5, 0])
```

You can't change `t[1]` because that would be altering the tuple. But you can change `t[2][1]` because that is just changing a list that happens to be inside a tuple.

If you have not used tuples of arrays before, this might seem a little odd, because it seems like we are altering the tuple. But in fact, we are not really altering the tuple at all. Think of it like this:

- Initially, our tuple contains 3 references, to 3 list objects.
- We change the value of one elements of the list object.
- The tuple still contains 3 references to the same 3 list objects. One of those lists now contains different values, but it is still the same list object. The tuple has not changed.

The way it works is quite logical, it can just catch you out at first if you were thinking that placing a list inside a tuple protects the list from being changed – it doesn't!

3.8 Summary

In this chapter we have covered the topic of mutability and looked at mutable and immutable objects in python. We have seen the potential problem with mutable objects being changed unexpectedly and the costs of defensive copying, and how immutable objects can help ensure that functions have no side effects.

We have also looked at the limitations and performance costs of immutable objects.

4 Recursion

Recursion is a common technique that is often associated with functional programming. The basic idea is this – given a difficult problem, try to find procedure that turns the original problem into a simpler version of the same problem. Apply the same procedure repeatedly to make the problem simpler and simpler, until you have a problem that is so simple you can just solve it in one go.

As a Python programmer you may well look at some examples of recursion and think that it would obviously be easier to write a loop instead. Some other languages don't have loops, so you have to use recursion, but in those cases the interpreter often creates a loop behind the scenes.

But there are plenty of problems that are inherently recursive in nature and would be very difficult to solve in any other way, so recursion is definitely something to have in your toolbox.

4.1 Factorials

This example is a slight cliché, but it is still a good illustration of both the beauty and pitfalls of recursion.

The factorial of an integer n is the product of all the integers between 1 and n. For example, 6 factorial (usually written 6!) is:

```
6*5*4*3*2*1 = 720
```

Now as we said in the introduction, the obvious way to do this is with a loop. But there is an alternative, "cleverer" way, using recursion.

We can make the simple observation that `6!` is actually `6*5!`. And `5!` is `5*4!`, and so on. So, we could calculate `n!` without ever explicitly calculating a factorial at all. We just keep relying on smaller and smaller factorials, without ever calculating them.

Of course, you must stop somewhere – we know that `1!` is 1.

Here is the Python code for calculating the factorial of `n`. Like we said, we just return n times the factorial of n – 1, unless n is 1 when we just return 1:

```
def factorial(n):
    if n>1:
        x = n*factorial(n-1)
    else:
        x = 1
    return x

print(factorial(6))
```

Amazingly enough, this works. We can investigate this further by adding some debug print statements:

```
def factorial(n):
    print('Enter', n)
    if n>1:
        x = n*factorial(n-1)
    else:
        x = 1
    print('Exit', n)
    return x
```

Here is what it prints

```
Enter 6
Enter 5
Enter 4
Enter 3
Enter 2
Enter 1
Exit 1
Exit 2
Exit 3
Exit 4
Exit 5
Exit 6
```

As you can see, we have called a function within a function within a function ... that's recursion, of course.

4.2 Recursion limits

Recursion is relatively inefficient compared to looping. This is because each step in a recursion results in a function call, whereas each step in a loop merely requires a "jump" to a different place in the code.

Calling a function involves considerably more work than a simple jump, and in any system it is going to take more time and use extra memory (memory is required to store the current state on the function – the values of its local variables – each time the function calls itself recursively).

However, Python has a rather more immediate problem. Recursive calls are limited to a depth of 1000. The code above cannot be used to calculate the factorial of any number greater than 1000.

This doesn't mean that recursion isn't a useful tool in Python. If you are processing a binary tree, for example, a depth of 1000 allows you to process a tree containing around 2^{1000} elements, which is a vast number. But if the problem can be solved with a simple loop, that is probably the best solution.

4.3 Tail recursion

The form of recursion exhibited by `factorial` is called *tail recursion*. Tail recursion is when the recursive call is right at the end of the function (usually with a condition beforehand to terminate the function before making the recursive call).

When a function is tail recursive, you can generally replace the recursive call with a loop. In Python, you usually *should* do that!

Some languages automatically spot tail recursion and replace it with a looping operation. This is often called TCO (Tail Call Optimisation). Python **does not** do this. It tends to happen in pure functional languages, where in some cases loops don't even exist. Such languages are often far more declarative than Python, which makes it easier to detect tail recursion.

There are some hacks that allow you to implement tail recursion in Python, but they are not covered here.

4.4 Inefficient recursion – Fibonacci numbers.

Here is another classic example of recursion – calculating the nth Fibonacci number. It turns out that this is hopelessly inefficient using pure recursion, but we will also look at a useful technique to alleviate the problem.

If you are not familiar with the Fibonacci series, it is an infinite series of numbers defined as follows:

```
F0 = 0
F1 = 1
F2 = F1 + F0 = 1
F3 = F2 + F1 = 2
...
F(n) = F(n-1) + F(n-2)
```

In other words, each element is the sum of the two previous elements. Here are the first few values of the series:

```
0, 1, 1, 2, 3, 5, 8, 13, 21...
```

This can obviously be calculated recursively, like this:

```
def fibonacci(n):
    if n==0:
        x = 0
    elif n==1:
        x = 1
    else:
        x = fibonacci(n-1) + fibonacci(n-2)
    return x

print(fibonacci(8)) # 21
```

Notice that we need to supply two initial cases. You can't calculate `F0` or `F1`, they are defined. The series is numbered from 0, so element 8 is 21.

If we now look at how this function actually works, by analysing adding `Enter` and `Exit` print statements as before. It turns out to be a bit of a nightmare!

Calculating `F8` requires us to calculate `F7` and `F6`. That is where the inefficiencies start, because of course calculating `F7` also requires us to calculate `F6`. Since these calculations are done in separate branches of the recursion, `F6` will be calculated twice.

Calculating `F6` twice then requires us to calculate `F5` twice, but we also need to calculate it again as part of the `F7` calculation, so we end up calculating `F5` three times.

Calculating `F6` twice and `F5` three times means we end up calculating `F4` five times. You might be noticing a pattern here – the number of times we have to calculate each successively lower level of recursion increases according to the Fibonacci series!

In short, this is a terribly inefficient method.

4.5 Memoization

The basic problem here is that we are calling `fibonacci` multiple times, with the same argument, but each time we are calculating the value all over again.

Now we know that `fibonacci` is a pure function. It has no side effects, and every time you call it with a particular value, you will always get the same result.

What we need is some way to remember all the times it has been called before, remember the result, and only calculate it if it is called with a value that has never been seen before. We can do this using a dictionary told all the previous calls. The dictionary key is the argument, the dictionary value is the result. Here is the code:

```
cache = dict()

def fibonacci(n):
    if n in cache:
        return cache[n]
    if n==0:
        x = 0
    elif n==1:
        x = 1
    else:
        x = fibonacci(n-1) + fibonacci(n-2)
    cache[n] = x
    return x

print(fibonacci(8))
```

Here we define an empty dictionary called `cache`. Every time we enter the `fibonacci` function, we check if the value if `n` already exists in the dictionary. If it does, we simply return the previous stored value for the function result, which is found in `cache[n]`.

If the value doesn't already exist, we calculate it in the normal way. Then before `fibonacci` returns we store the result in `cache`, so we never have to calculate it again.

4.5.1 functools lru_cache

This is all very well, but it is adding extra code to the `fibonacci` function. Extra code which in fact, has little to do with what the function is really doing, it has more to do with an efficiency improvement that you might wish to use with other function, not just `fibonacci`.

These so-called *cross cutting concerns* are exactly what decorators where invented for.

In addition, our cache implementation is quite crude and simplistic. It relies on having a global variable, `cache`, kicking around in the file, and hoping that nobody else uses it. It only works for functions that take exactly one argument. It also allows the cache to grow to any size, when it might sometimes be more sensible to set a maximum size.

Fortunately, there is an existing decorator, `lru_cache`, solves all those problems. It is in the `functools` module, and it only takes one line of code to set it up:

```
from functools import lru_cache

@lru_cache()
def fibonacci(n):
    print('Enter', n)
    if n==0:
        x = 0
    elif n==1:
        x = 1
    else:
        x = fibonacci(n-1) + fibonacci(n-2)
    print('Exit', n)
    return x

print(fibonacci(8))
```

That is it. Just import the decorator and add `@lru_cache` before the function definition, and it will only ever call `fibonacci` once for every value of `n`.

If you aren't familiar with decorators, they are explained in a later chapter.

4.6 Flattening lists

Consider a list like this:

```
[1, [2, 3], 4, [[5, 6], 7]]
```

This list contains a mixture if integers and lists. Those lists can also contain a mixture of integers and lists, and in fact the whole thing can be nested to any depth. You want to flatten this into a single list containing all the integers in the order they occur in the original unflattened list:

```
[1, 2, 3, 4, 5, 6, 7]
```

This is quite interesting because it is hard to come up with a solution that doesn't involve recursion. But there are different degrees to which you can use recursion. We will start with a fully recursive solution.

A simple fully recursive solution works like this. We take the original list and divide it into two parts. The first element of the list, which we will call the *head*, and the rest of the list, which we will call the *tail*.

The basic method is to flatten the head and flatten the tail, then join them together again. Since both parts of the list have been flattened, when we join them together we get a fully flattened list. Of course, we recursively call the flatten function to flatten the head and tail.

Of course, we need ours stopping conditions. If we are asked to flatten something that isn't a list (for example if it is an integer), we create a list from the value and return that. And if we are asked to flatten an empty list, we return an empty list. Here is the code:

```
def flatten(x):
    if not isinstance(x, list):
        return [x]
    if x == []:
        return x
    return flatten(x[0]) + flatten(x[1:])
```

Even without tracing through the code, it seems fairly plausible that this will work, for the following reasons:

- If you correctly flatten the head and the tail, and concatenate them, you will get a flattened list.
- Each iteration divides the list, so it will keep getting smaller.
- Every path therefore eventually results in a value that is either not a list (an integer value) or is an empty list. These two cases are handled correctly by returning a list representation that will be added to another list to create a solution.

Obviously, this falls well short of a mathematical proof of correctness, but it inspires confidence.

4.6.1 A less recursive solution

The solution above works. The main drawback is that it creates a least one level of recursion for every item in the list. That is because each level flattens the tail of the list and doesn't stop until the tail is empty. If the list is 100 elements long the recursion will be at least 100 deep. If the list is greater than 1000 long, even if the list is already flat, it will fail due to the Python recursion limit.

The basic problem is, in the quest for functional purity we have ended up with a solution that will break itself trying to flatten a list that is already flat. We can improve things be only flattening elements that are actually lists. Here is the solution:

```
def flatten(x):
    if not isinstance(x, list):
        return [x]
    if x == []:
        return x
    r = []
    for e in x:
        if isinstance(e, list):
            r += flatten(e)
        else:
            r.append(e)
    return r
```

This is still recursive, but it doesn't automatically recurse into the head and tail each time. It loops through the elements in x, and only flattens any lists it finds. If lists are nested, it will still recurse into those lists to flatten them, but the depth of recursion is limited by the depth of nesting of the lists, not the total number of elements in the original list. If the original list is flat, the function will make no recursive calls at all.

4.7 Summary

In this chapter we have looked at recursion as a more functional alternative to looping for certain algorithms. We have also seen the limitations of recursion in Python, in terms of the recursion depth limit and the lack of Tail Call Optimisation, and seen how memoization can help alleviate that in certain situations.

5 Closures

In functional programming, we sometimes write functions that create other functions. A simple and elegant way to do this is to use a closure. Closures provide a way to create functions dynamically, a little like lambdas but far more powerful.

5.1 Inner functions

Let's start by looking at *inner functions*. An inner function is a function that is defined inside another function. Like this:

```
def print3():

    def print_hello():
        print('hello')

    print_hello()
    print_hello()
    print_hello()

# Main code
print3()
print_hello()
```

`print_hello` is an inner function of `print3`. Specifically, `print3` first defines `print_hello`, then calls it 3 times.

The result when we run the main code is:

- Calling `print3` prints "hello" 3 times.
- Calling `print_hello` gives an error because it is only visible from inside `print3`.

5.1.1 Returning an inner function

A function can return an inner function, like this:

```
def make_print():

    def print_hello():
        print('hello')

    return print_hello

# Main code
fn = make_print()
fn()
```

Here the `make_print` function defines an inner function `print_hello`. But it doesn't call `print_hello`, it simply returns it as a function object.

When we call `make_print` in the main code, we assign the function pointer to the variable `fn`. This means that `fn` is now effectively and alias of the inner function `print_hello`. So, when we execute `fn`, it does what you would expect – it prints "hello".

This may be interesting, but it isn't particularly useful. We could, of course, have simply defined `print_hello` as a top-level function, and we wouldn't have needed any of the rest of the code.

5.1.2 A closure

This next step is where the magic happens, and we actually create our first closure:

```
def make_printx(x):

    def printx():
        print(x)

    return printx

# Main code
fn1 = make_printx(7)
fn2 = make_printx(100)
fn1()
fn2()
```

This time our outer function `make_printx` takes a parameter. And the inner function `printx`, uses that parameter. That is fine, of course, because an inner function can access the local variables and parameters of the enclosing function.

Now we call `make_printx` passing in a value of 7. This creates a function object for the function `printx`, but here is the important part – that function object is associated with the value `x = 7`. The combination of the function object together with the value of `x` is called a *closure*.

In the code above, `fn1` is a closure of `printx` with the `x` value 7, and `fn2` is a closure of `printx` with `x` value 100. Whenever we call `fn1` it will print 7, and whenever we call `fn2` it will print 100.

5.1.3 A more useful closure

Suppose we needed a function that prints a value, but automatically surrounds that value with brackets. We also want the ability to control what sort of brackets (such as [x] or {x} or <<x>>).

Here is how we could do that as a closure:

```
def make_printb(start, end):

    def printb(s):
        print(start + s + end)

    return printb

# Main code
sq = make_printb('[', ']')
dbl_ang = make_printb('<<', '>>')
sq('hello')
dbl_ang('world')
```

Here, `make_printb` accepts parameters for the `start` and `end` brackets. But in this case, the inner function `printb` accepts a parameter that represents the actual string to be printed inside the brackets. This means that the type of brackets if fixed when you create the closure, but you can set the content when you actually call the closure function.

So, when we create the closure `sq`, we set the brackets to be square brackets. Every time `sq` is called, it will use square brackets, but the content between the brackets can be whatever you like. Similarly, `dbl_ang` will always use double angle brackets.

What we have created with quite a simple closure is a factory for creating a whole family of functions that print bracketed text using different bracket styles.

5.2 What is a closure?

So, what is a closure? A closure normally requires three things:

- An outer function that contains an inner function.
- The outer function has parameters and/or local variables.
- The outer function returns the inner function as a function object.

In fact, strictly speaking any function that returns an inner function is a closure, even if it doesn't have any parameters. For example, our `make_print` function near the start of the chapter – but with no way to vary the behaviour of the closure, it isn't very useful.

5.3 Creating anonymous functions

In the next few sections we will look at various ways that closures can be used, starting with using closures to create anonymous functions.

5.3.1 A simple introduction to map

The `map` function is a Python built in function. In its simplest form it accepts a function object and a sequence (e.g. a list). It applies the function to each element of the list.

```
a = [2.2, 5.6, 1.9, 0.1]
b = map(round, a)
print(list(b))    # [2, 6, 2, 0]
```

In this example we apply the `round` function to every element in `a`. The `round` function rounds a value to the nearest integer. This gives the result shown. (Note that `map` uses lazy iteration, so we will use the `list` function to turn the result into a list that we can print).

5.3.2 Incrementing the elements in a list

Suppose we now wanted to add 1 to each element in the list. We need a function that accepts a single argument and adds 1 to it. One way to do this would be to use a lambda:

```
lambda x: x + 1
```

This creates an anonymous function that does exactly what we want. Let's try this with a map:

```
a = [1, 3, 0, 6]
b = map(lambda x: x + 1, a)
print(list(b))    # [2, 4, 1, 7]
```

5.3.3 Using a closure instead of a lambda

We can use a closure to create an anonymous function, instead of a lambda, like this:

```
def addn(n):
    def add(x):
        return x + n
    return add

a = [1, 3, 0, 6]
b = map(addn(1), a)
print(list(b))    # [2, 4, 1, 7]
```

Here, `addn(1)` creates an anonymous function that adds 1 to its argument - exactly like the lambda function we defined before. This involves more code than just using a lambda, because the closure has to be defined, but it has several advantages:

- If you need to use the function in more than one place, it might be better to define a closure.
- The closure allows you to create a family if related anonymous functions, for example `addn(2)` creates a function that adds 2 to its arguments.
- The function inside a closure can be as complex as you like, whereas a lambda is limited to a single expression. If you need a complex function, a closure is a good choice.

5.3.4 Other alternatives

There are other ways of creating anonymous functions like the ones here. You could use a callable object, as discussed earlier. Or you can use partial application, which is described in a later chapter.

5.4 Composing functions

Let's suppose you needed a function that could convert any character, for example 'a' into a hex string representing its ASCII character code (which would be '0x61').

There are two Python functions you can use to do this. `ord` converts a character to an int value representing its ASCII code (or more generally its Unicode value). `hex` converts an int value into a hex string. Using these two functions, we can define a function that does the task for us:

```
def codestr(c):
    return(hex(ord(c)))

h = codestr('a')
print(h)         # '0x61'
```

In this code we apply the `ord` function to the value `c`, and then apply the `hex` function to the result. Applying one function to the result of another is called *composing* the two functions.

Defining a function is a procedural way of doing things. A more functional way would be to *build* a function to do the job for us. Like this:

```
def compose(f, g):
    def fn(x):
        return f(g(x))
    return fn

codestr = compose(hex, ord)

h = codestr('a')
print(h)         # '0x61'
```

First, we define a compose function. `compose` accepts two functions, `f` and `g`. It returns a function that applies `g` to `x` and then applies `f` to the result. This is a completely generic that can used to compose any two functions you want. The only conditions are:

- `f` and `g` must each accept one parameter.
- The return value of `g` must be a valid input parameter for `f`.

The next step is use `compose` to create a function that applies `ord` and then `hex`. We will store this function object in `codestr`, then we can call it with value 'a' to test that it works.

5.4.1 The advantages of composing functions

Looking at the code, you might be thinking that the first version is simpler than the functional version. But remember that compose is a generic function that we might use many times. So, the original code looks like this:

```
def codestr(c):
    return(hex(ord(c)))
```

The equivalent functional code looks like this:

```
codestr = compose(hex, ord)
```

The code looks fairly similar, but the functional code demonstrates the intent much more clearly. The new function composes `hex` and `ord`, it says so right there! In the original code, that intent is expressed as a function that could be doing anything. You need to read the code to

be sure. It might seem like a minor difference, but with more complex code the cognitive burden can add up.

A related aspect is that, provided you trust `compose`, `hex`, and `ord`, then the functional solution *has to* work. How could it not, it is just three trusted functions doing what they do? With the original code, you have two trusted functions plus a brand-new function that, for all you know, could have a bug. Again, not very likely, but these things can add up in a more complex program.

Another advantage is that we can use `compose` to create anonymous functions. For example, we can use `map` to apply our composed function to a string and produce a list of hex values. This saves us an ugly lambda expression.

```
s = 'xyz'
b = map(compose(hex, ord), s)
print(list(b))         # ['0x78', '0x79', '0x7a']
```

If you are not too familiar with map, it will work with strings as well as arrays, applying the function to each character in the string and creating an iterator `b` that we then turn into a list.

Finally, we can use our `compose` function to create other functions. Here is how we would create a function calculate the square of the sine of x:

```
compose(lambda x: x*x, math.sin)
```

5.5 Using closures instead of classes

For the final example, we will look at a simple number formatter. We want a formatter that can convert a floating-point number to a string, with a fixed number of decimal places.

We could do this with a simple class, like this:

```
class Format():

    def __init__(self, precision):
        self.p = precision

    def format(self, x):
        return '{:.{prec}f}'.format(x, prec=self.p)
```

This class can be used to create a format object (as with the 3 digit case) or you can create and call the object in one statement (as with the 5 digit case):

```
format3 = Format(3)
print(format3.format(1.2345678))

print(Format(5).format(1.2345678))
```

Here is how you could use a closure as a factory to create format functions that do a similar job:

```
def formatn(precision):

    def format(x):
        return '{:.{prec}f}'.format(x, prec=precision)

    return format
```

And here are the two ways to call it:

```
format3 = formatn(3)
print(format3(1.2345678))

print(formatn(5)(1.2345678))
```

Both are valid, and there is nothing wrong with using a class in this case, but a closure offers quite an elegant solution. Generally, you can use a closure instead of a class if:

- The class would only have one method.
- The parameters are set in the `__init__` method and never changed.

If these conditions are not met, you will often be better using a class.

5.6 Using classes instead of closures

While we are looking at classes, it is worth mentioning that you can create classes that can be "called" like functions. All we need to do is define a method `__call__` in the class. This is useful to know, but probably not something you will use often.

`__call__` is one of the special methods that Python provides to allow user defined objects to support Python operators. All the methods are two underscores before and after their names to distinguish them from normal methods. For that reason, they are sometimes called "dunder" methods (double underscore), or alternatively magic methods. The method `__call__` supports function calling.

Here is some example code. We modify the class called `Format`, to include a `__call__` method instead of the previous `format` method.

```
class Format():

    def __init__(self, precision):
        self.p = precision

    def __call__(self, x):
        return '{:.{prec}f}'.format(x, prec=self.p)
```

Now we create an object `format3` as before. But this time, in order to invoke it we just need to use the same syntax as we would use to call a function:

```
format3 = Format(3)
print(format3(1.2345678))

print(Format(5)(1.2345678))
```

Notice that `format3` is not a function object, it is a `Format` object. But because it supports `__call__`, we can use function notation to call it. And, of course, we can still create an object like `Format(5)` and call it directly.

Well now our object can be used in a very similar way to the closure. Is it worth doing? Not usually, because defining a simple class is more hassle than defining a simple closure.

One scenario where the class might be a better choice is if you have more complex initialisation requirements. For example, suppose we wanted to allow the format to specify the number of decimal places and the overall width of the string. And to allow for more features later, we use a fluent interface style. Here is our fluent formatter:

```
class Format():

    def __init__(self):
        self.p = 0
        self.w = 0

    def prec(self, n):
        self.p = n
        return self

    def width(self, n):
        self.w = n
        return self

    def __call__(self, x):
        return '{:{width}.{prec}f}'.format(x, width=self.w,
                                           prec=self.p)
```

The fluent interface allows us to initialise our formatter like this:

```
format3 = Format().width(10).prec(3)
print(format3(1.2345678))
```

This type of interface can be very useful if the formatter has lots of options, with many of them optional. It can only really be done using a class.

5.7 Closure inspection

You can look inside a closure to find the values of its variables. Here is an example closure:

```
def f(x, a, v):
    def g(e):
        print(x, a, v, e)
    return g

c = f(5, 6, 7)
```

Python provides methods to inspect an object more deeply. In the code above, the function `f` returns the function object `g`, which is assigned to variable `c`. Since the object returned is a closure, it also contains information about the variables `x`, `a`, `v` and their values.

These variables are called free variables – the variables that are passed into `f`. In other words, variables that are used by `f` but not defined within `f`.

Every object in Python has "hidden" attributes that store information about its internals. For closures the important attributes are `__code__` and `__closure__`. These attributes aren't really hidden, of course, you can get a list of all them all using:

```
dir(c)
```

which gives you a dictionary of the names of the items:

```
['__annotations__', '__call__', '__class__', '__closure__',
'__code__', '__defaults__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__get__', '__getattribute__', '__globals__', '__gt__',
'__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
'__module__', '__name__', '__ne__', '__new__', '__qualname__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
```

We can list the free variables of `f` like this:

```
print(c.__code__.co_freevars)
```

The result is a tuple off values `('a', 'v', 'x')` which include all of the free variables. The variables are ordered by hash values, so it is safest to treat the order as being essentially random.

You can get the variable values from the __closure__ attribute. This contains a tuple of cells, where each cell contains the value of one of the variables. They can be accessed like this:

```
print(c.__closure__[0].cell_contents)
print(c.__closure__[1].cell_contents)
print(c.__closure__[2].cell_contents)
```

This returns 6, 7, 5, the values of `a`, `v` and `x`. The values are stored in the same order as the variable names in the `freevars` tuple. Here is how to list the names and values together:

```
for i, name in enumerate(c.__code__.co_freevars):
    print(name, c.__closure__[i].cell_contents)
```

This can be useful in some circumstances to find out the details of a closure in your code. The values are read only – you can't modify the value of a, for example.

5.8 Summary

In this chapter we have learned about closures – inner functions returned by an enclosing function, that still retain access to variables within the scope of the enclosing function.

We have seen how closures can be used as function factories, providing a simple and elegant way to implement composition and partial functions.

We have also seen haw they can used as a more declarative alternative to lambda expressions, and a clean alternative to declaring a class in certain cases.

6 Iterators

You will be familiar with Python *sequences*. A sequence is an ordered collection if items, the most common ones being lists, strings and tuples. But sequences rely on a couple lower level types that are important in functional programming: iterators and iterables.

6.1 Iterators

In Python, an iterator is an object that can be used to iterate over a series of values, one after the other. Specifically, an iterator can be passed into the built-in function `next` to get the next value in the series.

In the earlier chapter on closures, we used `map` to apply the `round` function to all the elements in a list:

```
a = [2.2, 5.6, 1.9, 0.1]
b = map(round, a)
```

If we were to print the value of b, we would find that it isn't a list of values, but instead it is a map object:

```
<map object at 0x000002581A529860>
```

A map object can act as an iterable (that is, it can be passed to the `next` function), so we can do this to print out its values:

```
print(next(b))  # 2
print(next(b))  # 6
print(next(b))  # 2
print(next(b))  # 0
print(next(b))  # throws StopIteration
```

The first four calls will print the consecutive rounded values of `a`. When we make the fifth call to `next(b)`, the iterator has run out of values, so it will throw a StopIteration exception. This isn't an error it is a standard way for an iterator to indicate that is has no more values left.

This illustrates an important feature of iterators – you only get one go. Each time you call `next` you get the next value, but you can't go back to the beginning, it is a one-shot deal.

6.2 Iterables

An iterable is something that you can iterate over. For example, lists are iterables (so are strings and tuples).

An iterable can be passed to the built in `iter` function. This function returns an iterator that you can use to do the actual iterating. Here is an example:

```
a = [1, 3, 7]

b = iter(a)
print(a)
print(b)
```

Here we have created a list, `a`, and obtained its iterator, `b`. Here is what we get when we print them:

```
[1, 3, 7]
<list_iterator object at 0x000001A279DC5A90>
```

`a` is a list object, and `b` is a `list_iterator`, a type of iterator that is configured to iterate over the values in b. Here is what happens if we call `next(b)` several times:

```
print(next(b))  # 1
print(next(b))  # 3
print(next(b))  # 7
```

This is quite similar to the `map` example, but because a list is an iterable rather than an iterator, we needed to take the extra step of calling `iter` to get the iterator.

6.3 How for loops work

Now we can take a quick look at how a for loop works. Consider this:

```
a = [1, 3, 7]
for x in a:
    print(x)
```

The for loop requires a variable (`x` in this case), and something to loop over (`a` in this case). The for loop operates as follows:

- It obtains an iterator from the iterable `a` using the `iter` function.
- It fetches values from the iterator, one by one. For each value, it assigns the value to `x` and executes the body of the loop.
- When the iterable throws `StopIteration`, the loop terminates.

6.4 Iterators also support iter

The previous description of for loops leaves us with a potential problem. The map function returns and iterator, but we need an iterable to use for loop. So how can we loop over a map, like this:

```
a = [1, 3, 7]
for x in map(lambda x: x*x, a):
    print(x)
```

The answer is that all iterators also support the `iter` function – but in the case of an iterator, calling `iter` returns the object itself. So:

- If you try to loop over an iterable, Python will use the `iter` function to get its iterator.
- If you try to loop over an iterator, Python will again call the `iter` function, but it will return the iterator itself.

Either way, the loop will obtain an iterator to work with.

6.5 Iterators vs iterables

To summarise:

- An *iterator* is an object that can iterate through a sequence of values, by repeatedly passing it to the `next` function.
- An *iterable* is an object that can be iterated over. If you pass an iterable to the `iter` function, it will return an iterator that you can use to iterate over it.

[[Iterables can create a new iterator so you can loop over a list multiple times, but and iterator can only be used once]]

6.6 Iterators use lazy evaluation

The only way to get values from an iterator is to request the next item. This means that if you want to get the 100th element, you will have to keep asking for the next item, 100 times! There is no other option.

This means that an iterator doesn't have to create all its values in one go. In fact, many iterators calculate their values one at a time, as they are needed. Each time you request the next item, the iterator will calculate it there and then. This is called *lazy* evaluation – the iterator doesn't do any work until it absolutely has to. There are several advantages to this:

- When you request the first value, the iterator can return it straight away. Without lazy evaluation, the iterator would need to calculate all its values before it could even return the first value. This can make your program more responsive if the series is very long.
- You do not need to store the calculated values. A long series might use a lot of memory if you needed to store it.
- You don't waste time calculating values that you might not use.

To give an example, suppose you had an iterator, `myiter`, that created 1000 values and you wanted to find the first zero value. You could do it like this:

```
for x in myiter:
    if x==0:
        break
```

Without lazy evaluation, `myiter` would calculate all 1000 values before the loop even started. If it then turned out that the second value was a zero, then you would have calculated the remaining 998 values for nothing! With lazy evaluation, `myiter` would calculate the first value just before the first pass through the loop, then the second value just before the second pass through the loop ... and then the loop would end, so the remaining 998 values would never be calculated.

In some cases, an iterator might be potentially infinite. For example, if you created an iterator to generate the series of prime numbers, it has no end. You would have to set some arbitrary limit for the longest prime number you can handle, and then you would have to wait for a long time until all those numbers were generated.

With lazy evaluation, you can just create prime numbers, one by one, as they are needed, and carry on going more or less forever.

There are some cases where lazy generation is not the best approach. One example would be an iterator that reads bytes from a file. It is generally going to be quite inefficient to access one byte at a time, so the iterator might decide to read a largish block of data in one go.

6.7 Sequences

A sequence is an ordered collection of items that allows random access. Examples of sequences include list, strings and tuples. You should already be familiar with these, but here is how they relate to iterators

Ordered means that each item in the sequence has an index, starting at 0.

Random access means we can directly access item index `i` in the sequence using square bracket notation:

```
n = a[i]
a[i] = 3
del a[i]
```

Immutable sequences such as tuples and strings only allow items to be read, not modified or deleted.

All sequences are iterable – that is, they support the `iter` function to obtain an iterator. This also means that they will work with for loops, of course.

In addition, sequences generally have a specific number of items, and you can use the len function to find out how many items there are in a sequence.

It is interesting to note that range creates an immutable sequence. You can do this:

```
r = range(2, 8)
print(len(r))  # 6
print(r[3])    # 5
```

In this example, `r` is a `range object`. It is an iterable, of course, but it also supports the use `len` and random reading of elements. But don't make the mistake of thinking that a range actually stores a list of value, like a list. It creates the values lazily. The values for `len(r)` and `r[3]` are calculated from the range parameters.

6.8 Realising an iterator

It is sometimes useful to convert an iterator into a concrete sequence such as a list. This is sometimes called "realising" the iterator. There are several reasons you might want to realise an iterator:

- To find its length.
- To access the elements more than once (an iterator can only be read from once, then it is spent).
- To access the elements in a different order.
- To print it.

The process on realising an iterator involves evaluating every term in the iterator and making those terms available as a sequence. There are two main ways to do this – using a sequence constructor such as `list` or using the `*` operator.

6.8.1 Using sequence constructors

Here is a simple example where we have created an iterator with the `map` function, and we want to print the result:

```
a = [2.2, 5.6, 1.9, 0.1]
b = map(round, a)
print(b)         # <map object at 0x000002470E579828>
```

The problem here is that `b` is an iterator, so when you print it you will just see the details of the iterator object, not the values it contains.

A simple way to do this is to use the `list` function. This will convert almost anything into a list. When you do this:

```
print(list(b))  # [2, 6, 2, 0]
```

The `list` function will loop through the iterator, evaluating each item, and create a list from all the items.

The `tuple` function will do a similar job, creating a tuple instead of a list:

```
print(tuple(b))  # (2, 6, 2, 0)
```

The `set` function will create a set instead of a list. Remember that a set only allows one instance of each value, so the number 2 will only occur once. Also, sets have no natural order so you shouldn't really rely on items being listed in any specific order:

```
print(set(b))  # {0, 2, 6}
```

Strings are slightly different. This example uses `map` and the `chr` function to convert a list of numbers into characters based on their ASCII values:

```
a = [72, 101, 108, 108, 111]
b = map(chr, a)
print(str(b))    # <map object at 0x000001D23F1E9828>
```

Unfortunately, one of the quirks of Python us that `str` works rather differently to `list`. While `list` will take an object and attempt to get all its elements and create a list, `str` works on a different level. It attempts to find a string representation of the object itself.

If we pass an iterator to `list`, it will realise the iterator and form a list from its elements. But if we pass an iterator to `str`, it will simply describe the iterator itself – in this case it is a map object. It won't evaluate the iterator.

The solution is to use the string `join` function. This takes an iterable of string values and joins them:

```
print(''.join(b))
```

If you are not familiar with join, it is a method of the string type. It joins all the elements of b to create a single string. The `''` is a literal empty string that causes join to join the strings with no extra characters between them.

6.8.2 Unpacking an iterable to a parameter list

Here is a simple function that multiplies three numbers:

```
def mult3(a, b, c):
    return a*b*c

x = mult3(2, 3, 5)  # 30
```

Suppose the arguments you needed were already in in a list? You can use the unpacking operator, *, to "unpack" the values:

```
p = [2, 3, 5]
x = mult3(*p)   # equivalent to mult3(2, 3, 5)
```

In this case, `*p` is equivalent to taking the elements in `p` and passing them in as three separate arguments.

This doesn't just work with sequences like lists and tuples. It will work with any iterable. Python will realise the iterable and pass the resulting elements into the function as separate arguments. To use our rounding example again:

```
a = [2.2, 5.6, 1.9]
b = map(round, a)
x = mult3(*b)    # equivalent to mult3(2, 6, 2)
```

In fact, `map` returns an iterator, not an iterable, but as we saw previously an iterator serves as its own iterable.

This technique is useful in a number of situations, as we will see later. Wherever you have a set of values in an iterable that you want to pass into a function, you can use * to unpack it. However, the number of elements in the iterable must match the number of arguments needed by the function.

6.8.3 Unpacking an iterable into a sequence

You can use exactly the same unpacking notation to create a list, by enclosing the unpacked variable inside square brackets:

```
a = [2.2, 5.6, 1.9]
b = map(round, a)
k = [*b]      # k is [2, 6, 2]
```

This is an alternative to using the `list` function described above. You can use the same technique to create a tuple, but you need the trailing comma just as you would to create a tuple with one normal element:

```
t = (*b,)    # t is (2, 6, 2)
```

And, of course, you can create a set in a similar way:

```
s = {*b}    # s is {2, 6}
```

6.8.4 Extended unpacking

You can unpack more than one iterable, and you can even interleave other values with unpacked values, for example:

```
a = range(3)
b = range(4, 7)
k = [*a, 10, *b]  # [0, 1, 2, 10, 4, 5, 6]
```

Here we use two range iterables, one with values 0, 1, 2 and the other with values 4, 5, 6. We unpack these both into a list, with an extra element 10 between them.

This technique also works with unpacking into argument lists, as we did above. It also works with tuples and sets.

6.9 Creating your own iterator

Just out of interest, we will create a couple of iterators of our own. In fact, you will probably never need to do this as it is almost always better to use *generators* instead, so the rest of the chapter if pretty much optional, for background information only. This section assumes you know the basics of creating Python classes.

An iterator is just a class that implements a `__next__` method and an `__iter__` method. Notice that these functions have double underscores before and after their names, which indicates they are special class methods defined by Python. The built-in `next` function calls the object's `__next__` method (similar for `iter`).

As we noted earlier, the `__iter__` method just needs to return the iterator itself (it allows the iterator to act as an iterable if needed). So, we will mainly concentrate on the `__next__` method.

6.9.1 An alphabet iterator

We will start with an iterator that returns the first 5 characters of the alphabet. It could easily be extended to return all 26 characters but keeping to 5 makes the examples a little shorter. Here is our skeleton class:

```
class Alphabet():

    def __init__(self):
        # Number of characters read
        self.pos = 0

    def __iter__(self):
        return self

    def __next__(self):
        # Add code to return the next value
```

To implement our next method, we will use a string to define the characters in the alphabet:

```
chars = 'abcde'
```

This string will be defined within the class, as a static variable. We can then define our `__next__` method.

```
    def __next__(self):
        if self.pos < len(self.chars):
            c = self.chars[self.pos]
            self.pos += 1
            return c
        else:
            raise StopIteration
```

This function is fairly simple. For the first 5 characters, we simply return the correct character from the string. We use `self.pos` as the index. Once all the characters are used up, we raise a `StopIteration` exception to show that the iterator is finished.

Here is the complete code, including a test. We can use `Alphabet` in a for loop, it will loop 5 times, printing `'a'` to `'e'`.

```
class Alphabet():

    chars = 'abcde'

    def __init__(self):
        self.pos = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.pos < len(self.chars):
            c = self.chars[self.pos]
            self.pos += 1
            return c
        else:
            raise StopIteration

for c in Alphabet():
    print(c)
```

6.9.2 A Fibonacci iterator

As a second example, we will make an iterator that returns values from the Fibonacci series. If you are not familiar with this, the series starts with 0, then 1. Each subsequent value is the series is formed by adding the two previous values, like this:

```
0, 1, 1, 2, 3, 5, 8, 13, 21...
```

The interesting thing about this series is that it is infinite. Our iterator will keep on creating values forever. Here is the code:

```
class Fibonacci():

    def __init__(self):
        self.c = 0
        self.n = 1

    def __iter__(self):
        return self

    def __next__(self):
        ret = self.c
        self.c, self.n = self.n, self.c + self.n
        return ret

for i in Fibonacci():
    print(i)
    if i > 100:
        break
```

Notice that, since our series is infinitely long, we use a break in the for loop otherwise it would go on forever.

6.10Built in functions

Python includes a number of functional that operate on iterables. We can split these into four groups.

6.10.1Primitive functions

The functions `iter` and `next` perform primitive operations on iterables/iterators. We covered these earlier in this chapter.

6.10.2Creation/conversion functions

We have seen earlier that the `list` and `tuple` functions can be used to realise a lazy iterable into a concrete sequence. These functions along with `str`, are also used to create and convert other items.

The `list` function can be used to create a list from various values:

```
list()          # Creates an empty list
list(1, 2, 3)   # Creates a list [1, 2, 3]
list(alist)     # Creates a shallow copy of alist
list(atuple)    # Creates a shallow copy of a tuple
list('abc')     # Creates a list ['a', 'b', 'c']
```

The `tuple` function works in the same way as the `list` function but creates a tuple rather than a list.

As noted earlier, `str` doesn't work in the same way as `list` and `tuple`. If you pass an object into `str`, it will create a string description of the object, rather than attempting to convert the object's data into a string of characters.

As an aside, the following code will create a shallow copy of any sequence and also maintain its type:

```
a = b[:]
```

This takes a full slice of the object, whatever it might be, creating a copy. If `a` is a list, `b` will be a list. If `a` is a tuple, `b` will be a tuple. If `a` is a string, `b` will be a string.

6.10.3Transforming functions

Transforming functions include `map` (that applies a function to an iterable to create a new series), `filter` (that removes items from a series based on a filter function). These are covered in the chapter *Transforming iterables*.

6.10.4Reducing functions

Reducing functions reduce all the values in an iterable to single derived value . Examples include `sum` (that adds all the elements to create a total) and `min` (that finds the smallest element). These are covered in the chapter *Reducing iterables*.

6.11Summary

In this chapter we have learned about:

- Iterators, iterables and how they interact.
- How for loops work.
- Lazy iteration and its advantages.
- Sequences as random access iterables.
- Converting lazy iterators to sequences.
- Creating your own iterators.
- Built-in functions on iterables.

7 Transforming iterables

Functional programming prefers iterables over lists, because there is less risk of side effects. We often need to transform an iterable stream in some way, and Python provides a number of standard functions to do that.

7.1 enumerate

You may have seen the `enumerate` function used in a for loop like this:

```
a = ('red', 'green', 'blue')
for i, s in enumerate(a):
    print(i, s)
```

This is a common idiom that is used if you ever need to access the loop counter within the loop. In this case, the loop operates 3 times, with `i` set to 0, then 1, then 2. The code prints 3 lines:

```
0 red
1 green
2 blue
```

You might have used this without ever thinking about what is going on behind the scenes. If so, it is quite useful to unpick it a bit. We can use `enumerate` in a more conventional loop with just one loop variable:

```
a = ('red', 'green', 'blue')
for t in enumerate(a):
    print(t)
```

This time the output looks like this:

```
(0, 'red')
(1, 'green')
(2, 'blue')
```

What `enumerate` is actually doing is returning a series of tuples. In the original version of the loop, we are simply unpacking this tuple into `i, s` so that `i` takes the values 0, 1, 2 and `s` takes the string values red, green and blue.

In the context of functional programming, where we try to avoid loops, enumerate transforms a data stream. For example, the stream of 3 values:

```
'red', 'green', 'blue'
```

gets transformed into a stream of tuples:

```
(0, 'red'), (1, 'green'), (2, 'blue')
```

This can often be quite a useful transform. And while we are talking about enumerate, don't forget that it can also take an optional start value, if you don't want to start from 0:

```
a = ('red', 'green', 'blue')
for i, s in enumerate(a, 15):
    print(i, s)
```

This creates the following output:

```
15 red
16 green
17 blue
```

7.2 zip

Another function you may have seen used in a for loop is `zip`. It provides a way to loop over more than one sequence in the same loop:

```
first = ('John', 'Anne', 'Mary', 'Peter')
last = ('Brown', 'Smith', 'Jones', 'Cooper')
age = (25, 33, 41, 28)
for f, l, a in zip(first, last, age):
    print(f, l, a)
```

This prints the following:

```
John Brown 25
Anne Smith 33
Mary Jones 41
Peter Cooper 28
```

On the first pass through the loop, `f`, `l` and `a` are set to the first element of `first`, `last` and `age` respectively. On the second pass, `f`, `l` and `a` are set to the second element of `first`, `last` and `age`, and so on. As you might have guessed, `zip` is producing tuples that are getting unpacked into `f`, `l` and `a`.

7.2.1 How zip transforms iterables

zip accepts a set of iterables, and transforms them into an iterator of tuples, like this:

```
a = (10, 11, 12, 13)
b = (20, 21, 22, 23)
c = (30, 31, 32, 33)

z = zip(a, b, c)
print(list(z))
```

We have converted the iterator `z` into a list, which looks like this:

```
[(10, 20, 30), (11, 21, 31), (12, 22, 32), (13, 23, 33)]
```

This is reorganised so that each output tuple contains the nth element from each input iterable. Exactly as we saw in the names example above.

What happens when we loop over this zipped stream? For example:

```
for t in zip(a, b, c):
    print(t)
```

We would print each tuple in turn:

```
(10, 20, 30)
(11, 21, 31)
(12, 22, 32)
(13, 23, 33)
```

And, of course, if we unpack the tuple in the loop:

```
for x, y, x in zip(a, b, c):
    print(x, y, z)
```

We would effectively be processing the three original lists, `a`, `b` and `c`, at the same time. Just like the name example above.

7.2.2 Stream with different lengths

Incidentally, if the original streams have different lengths, zip will terminate when the shortest stream is exhausted:

```
a = (10, 11, 12)
b = (20, 21)
c = (30, 31, 32, 33)

z = zip(a, b, c)
print(list(z))
```

This prints the following (because the `b` list only has 2 elements):

```
[(10, 20, 30), (11, 21, 31)]
```

7.2.3 zip is self-reversing

A common question people ask when they first meet `zip` is, how do I do the opposite? How do I *unzip* some data? It might not be immediately obvious, but the `zip` function is self-reversing - well almost. Looking at the output from the previous example:

```
[(10, 20, 30), (11, 21, 31), (12, 22, 32), (13, 23, 33)]
```

As you can see, taking the first element of each tuple gives `(10, 11, 12, 13)` – which is exactly the same as we started with. zipping some previously zipped data restores it to its previous state.

Or does it? There is a minor problem here in that the output of the `zip` function, called `z` in the example, is an iterator that provides a set of tuples. But we can't just pass that iterator back into `zip` again. `zip` expects each of the tuples to be passed in as a separate argument.

Fortunately, in the chapter on iterators, we met the * operator that converts an iterator to a list of arguments. `*z` is roughly equivalent to converting `z` to list and then passing in `z[0]`, `z[1]`, `z[2]`, `z[3]`:

```
a = (10, 11, 12, 13)
b = (20, 21, 22, 23)
c = (30, 31, 32, 33)

z = zip(a, b, c)

restored = zip(*z)
print(list(restored))
```

This gives us back our original data:

```
[(10, 11, 12, 13), (20, 21, 22, 23), (30, 31, 32, 33)]
```

7.3 filter

The `filter` function can be used to remove items from an iterable based on a testing function. It returns an iterator to access the result. Here is an example:

```
a = [3, 2, 1, 6, 7, 0]
f = filter(lambda x: x > 2, a)
```

This code takes uses a lambda expression as the testing function. In this case, the function returns true if the value of `x` is greater than 2. This test is applied to each element in the iterable `a`. Only those elements that pass the test are included in the output iterable. If we print `list(f)` we get only the elements that are > 2:

```
[3, 6, 7]
```

We can use `filter` in a for loop, as you would expect. This loop uses filter to only print the non-empty strings:

```
strings = ('red', '', 'green', '', 'blue')
for s in filter(len, strings):
    print(s)
```

The strings list contains both empty and non-empty strings. we use filter to apply the built in `len` function. For those strings that are empty, `len` will return 0. Python treats 0 as `False`, so those strings will be filtered out. Here is the what the program prints:

```
red
green
blue
```

7.4 map

The `map` function applies a supplied function to a set of arguments. It returns an iterator to access the results.

7.4.1 map with one parameter

In this example, we will use `map` with a user defined function that takes one parameter. We will use the `square` function from previous examples:

```
def square(x):
    return x*x

a = [2, 5, 6]
m = map(square, a)
```

The map function applies `square` to each value element in `a`, returning the squared values via an iterator. If we convert `m` to a list and print it, we get:

```
[4, 25, 36]
```

7.4.2 Lazy evaluation

This is perhaps a good time to revisit the idea of lazy evaluation. All the functions described so far use lazy evaluation. We will illustrate this by adding some extra print statements to our example above.

```
def square(x):
    print('Evaluating square', x)
    return x*x

a = [2, 5, 6]
print('Calling map')
m = map(square, a)
print('Called map')

print('Entering loop')
for x in m:
    print('Start of loop body')
    print(x)
```

Here is what this code prints out:

```
Calling map
Called map
Entering loop
Evaluating square 2
Start of loop body
4
Evaluating square 5
Start of loop body
26
Evaluating square 6
Start of loop body
36
```

We have placed print statements before and after the call to the `map` function (**Calling map** and **Called map**). Notice that the `square` function doesn't get called at all when we call `map` – if it did, we would see **Evaluating square** messages. All map does is to return an iterator, `m`, that will perform the calculations when we ask for each value.

We then start the loop (**Start of loop body** message). Within the for loop, we ask `m` for the next value. At this point, the iterator calls `square` once only. `square` is called with a parameter equal to the first value in the input list `a`. That value is 2, so `square` prints the **Evaluating square 2** message. The loop then prints the result of 2 squared, 4.

We loop round again, and again we ask `m` for the next value. The iterator calls `square` once more. `square` is called with a parameter equal to the next value in the input list `a`. That value is 5, so `square` prints the **Evaluating square 5** message. The loop then prints the result of 2 squared, 4.

We then execute the third and final iteration o the for loop, with an input value of 6, printing the **Evaluating square 6** message and the result, 36.

7.4.3 map with more than one parameter

We can use map with functions that take more than one parameter. We must supply map with extra iterable parameters, one for each argument that the applied function takes.

For example, in the previous code, `square` takes one argument, so `map` requires two arguments (the function, and an iterable supplying a series of values for the function argument).

In the next example, `add` takes two argument, so `map` requires three arguments (the function, and two iterables supplying a series of values for the first and second arguments). Here is the sample code:

```
import operator

a = [20, 30, 40]
b = range(3)
m = map(operator.sub, a, b)
```

This time we will use the `operator.sub` function. This is just a function version of the - operator, it takes two arguments `x`, `y` and returns `x–y`. We need to import the `operator` module to use `sub`.

We need two iterables because `operator.sub` takes two arguments. `a` is a list, `b` is `range(3)`, which of course provides a sequence 0, 1, 2. So map will calculate:

```
sub(20, 0)
sub(30, 1)
sub(40, 2)
```

The result, if we print `list(m)` is, as expected:

```
[20, 29, 38]
```

7.5 reversed

`reversed` is a useful function that returns an iterator that reverses the order of the elements in the original sequence. For example:

```
a = [2, 4, 6, 8]
r = reversed(a)
print(list(r))
```

Here `r` is an iterator that accesses the elements of `a` in reverse order. When we create a list from `r`, it contains:

```
[8, 6, 4, 2]
```

Note that reversed doesn't work with all types of iterable. It only works on sequences (lists, tuples, strings etc). You can't do this:

```
a = [2, 5, 6]
m = map(square, a)
r = reversed(m)
```

This is because `m` is not a sequence. You can fix this by converting `m` to a list or tuple before passing it to reversed:

```
a = [2, 5, 6]
m = map(square, a)
r = reversed(list(m))
```

For more details on the sort of objects that support reversed, and how to make your own reversible objects, refer to the later chapter on functional programming with classes.

7.5.1 Reversing a range

You can use `reversed` with `range` it is quite useful for counting backwards. For example, to count down from 9 to 0 you would need to do this using just `range`:

```
for i in range(9, -1, -1):
    print(i)
```

This is a little bit non-intuitive. Alternatively. you can just reverse a `range` that counts from 0 to 9. The result is much clearer:

```
for i in reversed(range(10)):
    print(i)
```

7.5.2 reverse

Lists have a method `reverse` that does the same thing as reversed, but it operates in place on the list:

```
k = [1, 3, 7]
k.reverse()
print(k)    # [7, 3, 1]
```

This method doesn't return anything, it just reverses the list itself. Don't get `reversed` and `reverse` confused.

7.6 sorted

We met the `sorted` function briefly in the **functions as objects** chapter. Here it is again in a bit more detail.

`sorted` isn't quite like the other transforming functions. It will work on any iterable, but it doesn't produce an iterator as output, instead it always creates a list. This doesn't usually cause any problems, but it is worth knowing.

It is interesting to compare `sorted` and `reversed`. They are both restricted, but in different ways:

- `reversed` requires a sequence as input but creates a lazy iterator as output. This is because the first thing you need to output when you reverse a series is the *last* element. You can't reverse a series unless you have random access to its elements, so a sequence is required as input.
- `sorted` can accept a lazy iterator as input but creates a list as output. Python uses a sorting algorithm called Timsort, that is derived from a hybrid of merge sort and insertion sort. The algorithm can accept data element by element but requires random access to the output list to place element in the correct final position.

7.6.1 Example – complex sort by month then year

We covered the basic operation of sorted in an earlier chapter. We will give another, slightly more advanced, example here, a complex sort on dates. We want the dates to be sorted by month, but within each month group to be sorted by year. Here are our dates:

```
dates = ['2019/04/06',
         '2017/04/15',
         '2019/03/21',
         '2018/04/10',
         '2019/04/08',
         '2017/03/20',
         '2018/06/30',
         '2019/09/30',
         '2018/04/11',
         '2017/03/14']
```

If we simply sort this list, we will get the dates in ascending order (that is because we are using a year/month/day format):

```
sorted_dates = sorted(dates)
```

Giving:

```
2017/03/14
2017/03/20
2017/04/15
2018/04/10
2018/04/11
2018/06/30
2019/03/21
2019/04/06
2019/04/08
2019/09/30
```

Now what if we wanted to sort this sorted list again, but just using the month field? You may recall that `sorted` accepts a key parameter that is a function. The function converts an item value (a date in this case) to a key that can be used to sort the list. We want to sort by month, so we need to convert a date value `'2019/04/06'` into a month value `'04'`. This can be done using a slice. Here is the sorted call with its key function (we have used a lambda):

```
sorted_by_month = sorted(sorted_dates, key=lambda x: x[5:7])
```

Giving:

```
2017/03/20
2017/03/14
2019/03/21
2017/04/15
2018/04/10
2018/04/11
2019/04/06
2019/04/08
2018/06/30
2019/09/30
```

The important thing here is that `sorted` is *stable*. This means that when we sort by month, all the entries that have the same month retain their original order relative to each other. So, you will see that the dates are primarily grouped by month, but within each group of same month items they are sorted by year.

To produce a list that is primarily grouped by month, and then sorted by date within each group, we must sort first by date and them by month.

7.6.2 Some utility key functions

Suppose we have the following list of people's details, stored as a list of tuples:

```
people = [('John', 'Brown', 25),
          ('Anne', 'Smith', 33),
          ('Mary', 'Jones', 41),
          ('Peter', 'Cooper', 28)]
```

We would like to sort them by their second names. That isn't difficult, we can just use a lambda function at the key, to extract the second element, like this:

```
sorted_by_surname = sorted(people, key=lambda x: x[1])
```

There is nothing wrong with this, but if you read the code you need to take a look at the lambda function to understand it. A lambda could be doing anything, but in this case all it is doing is getting the second item from a tuple.

As it happens, the `operator` module has a function, `itemgetter`, to help with this. We have met this module before it includes function equivalents for the standard operators. For example, the `add` function can be used in place of the + operator. The `itemgetter` function can be used in place of list indexing (the `[]` operator).

It is used like this:

```
from operator import itemgetter

sorted_by_surname = sorted(people, key=itemgetter(1))
```

This is better than before because it is more declarative. Instead of defining a lambda function to get an item, you are using the standard `itemgetter` function to do it.

It is worth noting that `itemgetter` isn't quite as simple as it seems. The `key` parameter requires a function as its value.

`itemgetter(1)` doesn't get the second element from a sequence. It returns a function that gets the second element from any sequence you pass to it. It acts rather like a closure:

```
f = itemgetter(1)
t = ('Anne', 'Smith', 33)
s = f(t)  # 'Smith'
```

This is exactly what we need, of course, because `sorted` is going to apply this function multiple times to get the second element from every item in the list.

For objects with named attributes, the `attrgetter` function does a similar job, except that it takes a string (the attribute name) rather than an integer.

Another useful operator function is `methodcaller`. This returns a function that calls a particular method on any object you pass to it. Let's see how this works.

Here is an example of trying to sort some strings:

```
fruits = ['Banana', 'apple', 'Apricot', 'Clementine',
          'avocado']
sorted_names = sorted(fruits)
```

This doesn't do quite what you might want. The problem is, the default string sort is case sensitive. All uppercase Latin letters come before all lowercase ones. So, *Banana* would come before *apple*.

To fix this we need to use a lowercase version of the string as the key. We need to call each string's `lower` method to generate a sort key. We could do this with a lambda as before:

```
sorted_names = sorted(fruits, key=lambda x: x.lower()))
```

This fixes the problem, but a better method is this:

```
from operator import methodcaller

sorted_names = sorted(fruits, key=methodcaller('lower'))
```

Again, `methodcaller` creates a function. This new function calls the lower method of any object you pass to it:

```
f = methodcaller('lower')
s = f('Banana')  # 'banana' equivalent to 'Banana'.lower()
```

7.6.3 Reversing the sort order

You can reverse the sort order using the optional `reverse` parameter, which should be set to `True` to reverse the sort. This is particularly useful is you are relying on the natural Python sorting order. For example, form our earlier date sorting example, we could sort the dates in descending order like this:

```
sorted_dates = sorted(dates, reverse=True)
```

This would sort the dates from the most recent to the oldest.

7.6.4 sort

Lists have a method `sort` that does the same thing as `sorted`, but it operates in place on the list:

```
k = [1, 7, 2, 4, 1]
k.sort()
print(k)    # [1, 1, 2, 4, 7]
```

This method doesn't return anything, it just sorts the list itself.

`sort` has the same optional parameters, `key` and `reverse`, that `sorted` has. They work in exactly the same way.

7.7 Combining functions

It is often useful to combine these functions, often in a single expression. Here are some examples.

7.7.1 map and filter

map and filter work well together. Here is an example where we are using map to take the square root of a series of numbers. Since the square root function doesn't accept negative input, we use filter to remove any negative values first. Here is the code:

```
import math

k = [1, 4, -2, 16, -3, 36, -1]

f = filter(lambda x: x>=0, k)
m = map(math.sqrt, f)

print(list(m)) #[1.0, 2.0, 4.0, 6.0]
```

Of course, the output data has less elements than the input data because some negative values have been filtered out. We have shown the map and filter as two separate lines of code, but it would be quite normal to combine them like this:

```
m = map(math.sqrt, filter(lambda x: x>=0, k))
```

7.7.2 Pipelines

We looked at lazy evaluation earlier in this chapter. When we chain two or more functions that use lazy evaluation, we create a pipeline. In this section we will see how this works.

We are going to use map and filter again, but this time using a couple of user defined functions whose main job is to print something out so we can tell when each function gets executed. Here is the `same` function:

```
def same(s):
    print('Same', s)
    return s
```

The `same` function just prints a message and returns the same value it received. And here is the `not_empty` function:

```
def not_empty(s):
    if s:
        print('True', s)
        return True
    else:
        print('False')
        return False
```

This function returns `True` if the string is not empty, `False` otherwise. It also prints what it has done. Now here is the main loop:

```
k = ['a', '', 'b', '']
m = map(same, filter(not_empty, k))
print('Start')
for s in m:
    print('In loop', s)
```

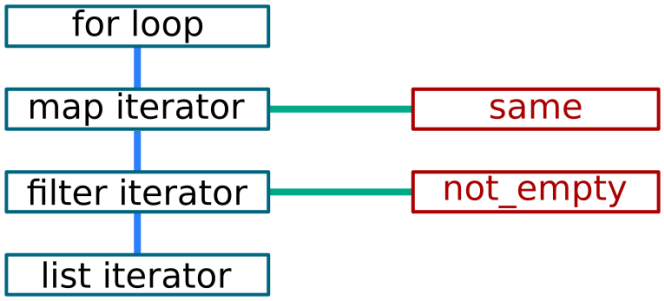
And here is what it prints:

```
Start
True a
Same a
In loop a
False
True b
Same b
In loop b
False
```

Let's look at this step by step. We first create our map expression:

```
m = map(same, filter(not_empty, k))
```

This line doesn't print anything – we know this, because `Start` is the first thing printed. It doesn't call `not_empty` or `same`. It just sets up a pipeline of iterators:



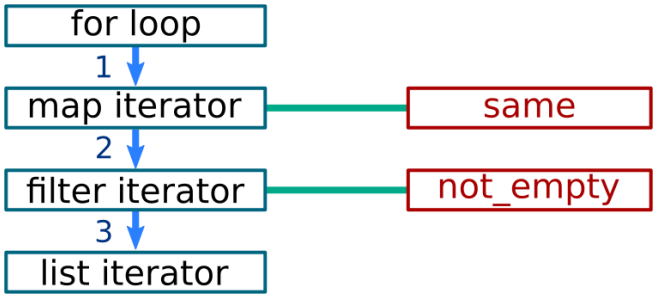
The **first iteration** of the loop prints this:

```
True a
Same a
In loop a
```

Here is how it works. First a set of requests go down the pipeline:

1. The loop requests a value from the map iterator.
2. The map iterator requests a value from the filter iterator.
3. The filter iterator requests a value from the list iterator.

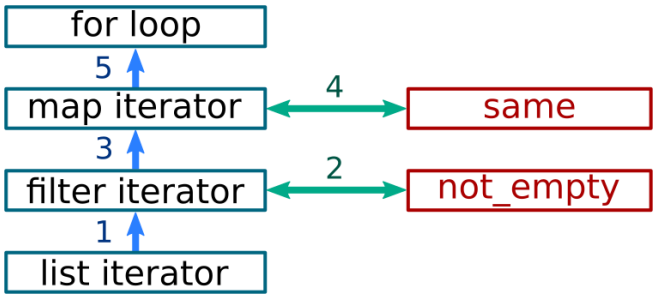
Here it is as a diagram:



Next, the responses get sent back up the pipeline. This is when our functions actually get called:

1. The list iterator passes the value `'a'` back to the filter iterator.
2. The filter iterator passes the value `'a'` to the `not_empty` function, which prints `'True a'` because the string a is not empty.
3. The filter iterator passes the value `'a'` is passed back to the map iterator.
4. The map iterator passes the value `'a'` to the `same` function, which prints `'Same a'`.
5. The map iterator passes the value `'a'` is passed back to the loop iterator.

At this point, the loop prints `'In loop a'`. Here is this as a diagram:



The **second iteration** of the loop prints this:

```
False
True b
Same b
In loop b
```

This is very similar to the first iteration, except that when the filter iterator requests a value from the list iterator, it gets an empty string (the second value in `k`). This means that the `not_empty` function prints `'False'` and returns `False`.

Now the whole point of the filter step is to filter out the cases when `not_empty` returns `False`. So, the filter doesn't pass this value back to the map iterator, instead it throws it away. Then it requests the next value from the list iterator, which is a `'b'` this time, so it gets passed back up the pipeline as before.

In the **final attempt** of the loop, filter gets the last item from the list iterator, which happens to be another empty string. It discards this, the loop iterator throws a `StopIteration` exception, and the for loop ends.

7.7.3 map and zip

Here is some code that uses map to format the names data from the previous zip example:

```
def format_person(first, last, age):
    return '{}, {} - age {}'.format(last, first, age)

first = ('John', 'Anne', 'Mary', 'Peter')
last = ('Brown', 'Smith', 'Jones', 'Cooper')
age = (25, 33, 41, 28)

m = map(format_person, first, last, age)

list(map(print, m)) # Prints the result
```

Here is the output you would get:

```
Brown, John - age 25
Smith, Anne - age 33
Jones, Mary - age 41
Cooper, Peter - age 28
```

Incidentally, we can use `map` to print the list, as shown in the example. It is neater than a loop but note that you need realise the map output (for example by converting it to a list) otherwise `print` will never get called.

But suppose the data wasn't in quite the correct format. Suppose you had a list of person records:

```
people = [('John', 'Brown', 25),
          ('Anne', 'Smith', 33),
          ('Mary', 'Jones', 41),
          ('Peter', 'Cooper', 28)]
```

In order to map these, we need to *unzip* them. As we saw, we can use zip to unzip data, it is self-reversing. So, we just need to change our map call to this:

```
m = map(format_person, *zip(*people))
```

Why do we need `*zip(*people)`? Well firstly, `people` is a list, but `zip` needs a set of separate arguments, so we must unpack `people` so `zip` can use it. And secondly, `zip` returns an iterator, but `map` needs a set of separate arguments, so we must unpack `zip` so `map` can use it.

*zip(*people)	zip(*people)	*people	people
('John', 'Anne', 'Mary', 'Peter') ('Brown', 'Smith', 'Jones', 'Cooper') (25, 33, 41, 28)	<zip object>	('John', 'Brown', 25) ('Anne', 'Smith', 33) ('Mary', 'Jones', 41) ('Peter', 'Cooper', 28)	[('John', 'Brown', 25), ('Anne', 'Smith', 33), ('Mary', 'Jones', 41), ('Peter', 'Cooper', 28)]

7.8 Summary

In this chapter we have learned about the built-in functions that Python provides to transform iterables.

We have seen how `enumerate`, `zip`, `filter` and `map` provide lazy evaluation.

We have also seen how to combine these functions in different ways, and how the resulting lazy pipeline doesn't access elements of the original iterable until they are needed.

There are some additional functions on iterables in the `itertools` library, discussed in the chapter *Useful libraries*.

8 Reducing iterables

In the earlier chapter *Iterators,* we divided the built-in iterator functions into groups – transforming, reducing, converting and primitives. In this chapter we will cover the reducing functions.

A reducing function takes all the values from an iterable and reduces them to a single representative value. For example, `sum` adds all the values in an iterable and returns the total value.

8.1 len

`len` should be very familiar. It simply returns the length of the item – the number of elements if it is a list or tuple, the number of characters if it is a string:

```
len([1, 2, 30])   # 3
len('uvwxyz')     # 6
```

`len` doesn't work with lazy iterables (such as the output of `map` or `filter`). You can convert a lazy iterable to as list and apply `len` to the result. Alternatively, you can use one of the methods described in the section on the map reduce pattern, later in this chapter.

8.2 sum

`sum` accepts an iterable and returns its sum – the result of combining all its elements using the + operator. For example:

```
a = [2, 5, 7, 1]
print(sum(a))  # 15
```

You can supply an optional start value to `sum`. It will just get added to the total:

```
a = [2, 5, 7, 1]
print(sum(a, -3))  # 12
```

`sum` will also work with sequences such as lists. However, the code below won't quite work:

```
a = [[2, 4], [0, 0], [5, 3]]
print(sum(a)) # ERROR
```

You will get a cryptic error message about unsupported operand types. The problem is that `sum` has a default start value of 0, so the code above will effectively be trying to calculate:

```
0 + [2, 4] + [0, 0] + [5, 3]
```

This is invalid because you can't add a list and an integer. The solution is to add an empty list as the start value, so sum calculates this instead:

```
[] + [2, 4] + [0, 0] + [5, 3]
```

This is now a valid calculation. Your code will look like this:

```
a = [[2, 4], [0, 0], [5, 3]]
print(sum(a, [])) # [2, 4, 0, 0, 5, 3]
```

An alternative way to join iterables is the `itertools.chain`, covered in the chapter on `itertools`. It is often more efficient.

Note thing to note is that `sum` does **not** work with strings. Python deliberately prevents this because it is terribly inefficient to add a set of strings together using `add`. It is far better to use `join` to concatenate strings, like this:

```
a = ['abc', 'pqr', 'xyz']
s = ''.join(a)
```

8.3 min

`min` accepts an iterable and returns the minimum value from the iterable. For example:

```
a = [2, 5, 7, 1]
print(min(a))  # 1
```

The iterable can contain items of any type, provided they can be compared with each other. For example, it can contain strings, or lists, which will be compared in the standard way. Here is an example with lists:

```
a = [[1, 2, 3], [1, 1, 5], [6, 7, 8], [1, 1, 5]]
print(min(a))  # [1, 1, 5]
```

Lists are compared element by element, so `[1, 1, 5]` is less than `[1, 2, 3]`. You might also notice that there are two lists with the value `[1, 1, 5]`. That is two different object that happen to have the same value. `min` will always return the first object in that case.

8.3.1 default argument

If you call `min` on an empty iterable, you will get a `ValueError` exception. You can avoid that using the `default` argument. This is a keyword only argument, that is used like this:

```
a = []
print(min(a, default=0))  # 0
```

Since the list is empty, min returns the default value 0. This value is only used in the iterable is empty, so for example:

```
a = [2, 5, 7, 1]
print(min(a, default=0))  # 1
```

Even though the default value is less than 1, `min` returns 1 as that is the smallest item in the list.

8.3.2 key argument

`min` has an optional argument `key` that can be used to modify the comparison order. It is a keyword only argument and works in the same way as the `sorted` function. Here is a simple example using a lambda function that returns the third element of the item:

```
a = a = [[1, 2, 3], [1, 1, 5], [6, 7, 8], [1, 1, 5]]
print(min(a, key=lambda x: x[2]))  # [1, 2, 3]
```

Since we are now comparing the third element, `x[2]`, the list `[1, 2, 3]` is the smallest. See the description of `sorted` for more details.

8.4 max

`max` works in a very similar way to `min`, except that it returns the maximum value form an iterable.

8.5 any

`any` accepts an iterable. It will return `True` if any of the elements have a true value. It will return `False` if none of the elements have a true value, or if the iterable is empty:

```
print(any([1, 0, 2]))      #1
print(any(['a', '', 'z'])) #2
print(any([0, '', False])) #3
print(any([]))             #4
```

1. `True` because values 1 and 2 count as true
2. `True` because `'a'` and `'z'` count as true
3. `False` because 0, `''` and `False` count as false
4. `False` because the iterable `[]` is empty

8.6 all

`all` accepts an iterable. It will return `True` if all of the elements have a true value. It will return `False` if any of the elements have a false value. Unlike `any`, `all` will return `True` if the iterable is empty:

```
print(all([1, 0, 2]))      #1
print(all(['a', '', 'z'])) #2
print(all([1, 'a', True])) #3
print(all([]))             #4
```

5. `False` because value 0 counts as false
6. `False` because `''` counts as false
7. `True` because 1, `'a'` and `True` count as true
8. `True` because the iterable `[]` is empty

8.7 functools reduce

If the reduce functions above don't meet your needs, you can create your own using the `reduce` function.

This function lets you define your own behaviour. For example, suppose we wanted to reduce a list by multiplying the elements. We can do this:

```
import functools, operator

a = [2, 3, 5, 2]
print(functools.reduce(operator.mul, a)) # 60
```

Remember that `operator.mul` is a function equivalent of the multiply operator `*`. This performs the equivalent of:

```
(((2 * 3) * 5) * 2)
```

`reduce` accepts a function and an iterable. The function you supply must take two parameters. reduce works like this:

1. Get the first and second values from the iterable and combine them using the function.
2. Get the next value from the iterable. Combine the previous result with the new value using the function.
3. Repeat 2 for all items in the iterable.

8.7.1 Initial value

`reduce` accepts and optional third argument, `initializer`. This provides an initial value when reduce first starts:

```
a = [2, 3, 5, 2]
print(functools.reduce(lambda x, y: x*y, a, 10)) # 600
```

In this case we have an initializer of 10, so the calculation is:

```
((((10 *2) * 3) * 5) * 2)
```

giving 600. Adding an `initializer` is similar to adding an extra value at the start of the input iterator.

8.7.2 Special cases

With no initialiser:

- If the iterable is empty, `reduce` will throw a `TypeError`.
- If the iterable only has one element, `reduce` will return that element.

With an `initialiser`:

- If the iterable is empty, `reduce` will return the value of the `initializer`.
- If the iterable only has one element, `reduce` will return the value of the `initializer` combined with the element.

8.8 The map-reduce pattern

The map-reduce pattern is a way of processing large data sets in a way that can be distributed amongst many computers.

The basic idea is to start by processing data elements individually, and finally combine them to give the required result.

To give a simple example, suppose we wanted to calculate the average word length of the words in a block of text:

The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code -- not in reams of trivial code that bores the reader to death.

We can break this task down into two steps:

- Count the number of letters in each word.
- Sum the total number letters in all the words.

Dividing the sum by the number of words will give us our result, the average word length.

Here is a list of our words, with punctuation removed:

```
strings = ['the', 'joy', 'of', 'coding', 'Python', 'should',
           'be', 'in', 'seeing', 'short', 'concise',
           'readable', 'classes', 'that', 'express', 'a',
           'lot', 'of', 'action', 'in', 'a', 'small', 'amount',
           'of', 'clear', 'code', 'not', 'in', 'reams', 'of',
           'trivial', 'code', 'that', 'bores', 'the', 'reader',
           'to', 'death']
```

Counting the number of letters in each word is fairly easy – we just map the `len` function onto the list of strings:

```
lengths = map(len, strings)
```

`lengths` is now an iterator that stream the lengths of the individual words. We could print this using:

```
print(list(lengths))
```

Not forgetting to convert the iterator to a list so we can print its values. This will give us:

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

Now we need to calculate the average length of each word – this is simply the sum of the lengths of all the words, divided by the number of words:

```
average = sum(lengths)/len(strings)
```

So, we can calculate the average word length of a list of strings using two fairly simple and obvious lines of code:

```
lengths = map(len, strings)
average = sum(lengths)/len(strings)
```

If we really wanted, we could even get this down to one line. How readable it is depends on how familiar you are with reading functional code, but the following probably isn't excessively complex:

```
average = sum(map(len, strings))/len(strings)
```

8.8.1 Ignoring short words

As a further example, let's try the same thing, but we will take the average of all the words excluding 'a' and 'the'. We can do this by filtering `strings` like this:

```
filter(lambda x : x not in ('a', 'the'), strings)
```

This will return an iterable of all the strings in `strings` that are not 'a' or 'the'. We can calculate the average of the list in a similar way to before using the filtered list:

```
s = filter(lambda x : x not in ('a', 'the'), strings)
average = sum(map(len, s))/len(s)   # ERROR
```

But there is a problem. We can't find the `len` of s because `s` is an iterator. Iterators don't support the `len` function. A quick fix, though not really part of the functional programming paradigm, is to convert the filtered words to a list. Here is the complete, working code:

```
s = list(filter(lambda x : x not in ('a', 'the'), strings))
average = sum(map(len, s))/len(s)
```

8.8.2 A more FP solution

The solution above has a minor problem in that it is necessary to store the entire list of values in memory before calculating the average. This is not usually a problem, in fact if we weren't concentrating of functional programming, we probably wouldn't give it a second thought.

But suppose we wanted to find the average word length of all the pages on Wikipedia? At the time of writing, a typical PC would struggle to hold the full contents in memory at one time.

How could we modify our code to work with any number of words, with limited memory?

The obvious solution would be to count the number of words as we sum them. We could create our own reducing function:

```
def sumcount(it):
    sum = 0;
    count = 0;
    for x in it:
        sum += x
        count += 1
    return sum, count
```

This function behaves a lot like the standard `sum` function, but it also counts the number of elements as it goes. At the end it returns a tuple of the sum of all the elements and the number of elements. We could use this to calculate the average like this:

```
s = filter(lambda x : x not in ('a', 'the'), strings)
total, count = sumcount(map(len, s))
average = total/count
```

This solution calculates the average without having to store a copy of all the words. If we were really trying to process the whole of Wikipedia, of course, we would not use the `strings` list to store all the words. We would create some kind of iterator that fetched the words from the web, one at a time.

This is a reasonable solution, the main code is pure functional code. The `sumcount` function uses a loop, which isn't ideal, but it is quite hidden away.

8.8.3 Using enumerate and reduce

We could improve things further by getting rid of that pesky loop. What we need to do is sum the values and count them at the same time. Maybe the `enumerate` function could help. We can enumerate the output of `map`:

```
s = filter(lambda x : x not in ('a', 'the'), strings)
m = map(len, s)
e = enumerate(m, 1)
```

The second parameter in `enumerate` function is the start value. It will start counting from 1 rather than 0. The iterator `e` gives the following values:

```
(1, 3), (2, 2), (3, 6), (4, 6), (5, 6)...
```

The first element of each tuple is the word count so far. The second element is the length of the current word. What we really need to do is reduce this sequence in such a way that the word count is maintained (we just keep the most recent version) but the lengths are summed.

We can use `functools.reduce` to do this. Here is how we would use `reduce` to simulate the `sum` function:

```
functools.reduce(operator.add, m)   # same as sum(m)
```

But we are going to accumulate a series of tuples, so we need an alternative to the `add` function. Here it is:

```
def opsumcount(a, b):
    return(b[0], a[1] + b[1])
```

This function accepts two tuples, `a` and `b`. It returns a tuple. The first element of the return tuple if the most recent word count `b[0]`. The second element is the accumulated sum of word lengths `a[1] + b[1]`. Here is the complete solution, without a loop or stored list in sight:

```
import functools

def opsumcount(a, b):
    return(b[0], a[1] + b[1])

s = filter(lambda x : x not in ('a', 'the'), strings)
m = map(len, s)
length, total = functools.reduce(opsumcount, enumerate(m, 1))
average = total/length
print(average)
```

8.9 Summary

In this chapter we have looked at the various standard Python functions that reduce iterables (converting and iterable to a single representative value).

We also looked at the general purpose functools reduce function, that can be used to create our own specialised reducing functions.

Finally, we looked at an example of using the map-reduce pattern to analyse text, using several functional programming techniques.

9 Comprehensions

It is often useful to create a list with particular content, perhaps based on another list or iterable. It is possible to do this using a loop, or perhaps a `map` function.

List comprehensions provide an alternative that is more declarative than a loop and often clearer than using a `map` function.

In addition to list comprehensions, there are similar techniques for generating lazy iterators (generator comprehensions), sets and dictionaries, which we will also cover in this chapter.

9.1 List comprehensions

To start with a simple example, suppose we wanted to create a list, length 100, filled with the strings '0' to '99'. There are several ways to do this. We could use a loop:

```
a = []
for i in range(100):
    a.append(str(i))
```

We could use `map`:

```
a = list(map(str, range(100)))
```

Now here is the list comprehension version:

```
a = [str(i) for i in range(100)]
```

In every case we get the same result, a list of 100 strings:

```
['0', '1', '2' ... '98', '99']
```

None of these solutions is terrible. The first solution is the most verbose and the least declarative. What do we mean by that? It is a loop that just happens to be building up a list according to a simple pattern. But since it is a loop, it could be doing anything. The code is in an imperative style, it doesn't just tell Python what sort of list you need, it tells Python exactly how to create the list.

Imperative code is more flexible, but that can be a disadvantage when you only need to do something simple and boring. You have to double check the code to make sure it isn't doing something more complicated than you think.

The map case is more declarative in style. It says that you want to `map` the `str` function onto the numbers 0 to 99, and make a `list` out of the result. These are all standard Python functions, applied in a standard way. In this particular case, there isn't much to fault this solution.

The final example uses a list comprehension. My personal opinion is that it is slightly preferable to the `map` example, but only slightly. List comprehensions exist for the specific purpose of creating a list from another iterable, so when you see that syntax you know exactly what the code is doing.

The best way to understand a list comprehension is probably to read it as an English sentence:

```
make a list of str(i) for all values of i in range(100)
```

Here is a different example. We want to take a list of numbers and create a new list where the numbers have been rounded to the nearest 5. Here is how we could do this in a loop. Notice that to round to the nearest 5, we divide by 5, round to the nearest whole number, then multiply by 5:

```
k = [12, 33, 49, 57]
a = []
for x in k:
    a.append(round(x/5)*5)
```

Using `map`, it is:

```
k = [12, 33, 49, 57]
a = list(map(lambda x: round(x/5)*5, k))
```

And as a list comprehension:

```
k = [12, 33, 49, 57]
a = [round(x/5)*5 for x in k]
```

In this case, the list comprehension benefits from not having to use a lambda function, which makes it marginally less complex.

9.2 Using conditions

Now imagine you wanted to find the square root of every value in a list, but you wanted to ignore any negative values, so that they don't even appear in the output list. Here is how you might do it with a loop:

```
k = [-1, 16, 9, -4, 0, 25]
a = []
for x in k:
    if x>=0:
        a.append(math.sqrt(x))
```

You will also need to import math at the start of all these examples, of course. The result will be:

```
[4.0, 3.0, 0.0, 5.0]
```

There are 6 values in the input list `k`, but only 4 in the output list `a`, as you would expect because the 2 negative values are excluded.

Here is the equivalent functionality using `map`. We use `filter` to remove the negative values before we take the square root:

```
a = list(map(math.sqrt, filter(lambda x: x>=0, k)))
```

Finally, here is the list comprehension version:

```
a = [math.sqrt(x) for x in k if x>=0]
```

It is a matter of preference to some extent, but this solution seems more readable than the `map` example. If you are having difficulty reading this list comprehension, try this:

```
make a list of sqrt(x) for all values of x in k,
but only if x>=0
```

9.3 Nested comprehensions

You can create nested list comprehensions. In fact, there are a couple of ways to do it.

9.3.1 Creating a 2D list

Possibly the easiest example of a nested list comprehension is to nest one comprehension inside another. For example:

```
[[x for x in range(3)] for y in range(4)]
```

We know that:

```
[x for x in range(3)]
```

Gives `[0, 1, 2]`. So, we are really calculating:

```
[[0, 1, 2] for y in range(4)]
```

Which, of course, evaluates to:

```
[[0, 1, 2],
 [0, 1, 2],
 [0, 1, 2],
 [0, 1, 2]]
```

We are simply creating an outer list of 4 elements where each of those elements is a list of 3 elements.

We could extend this by making the output element depend on `x` and `y`:

```
[[x + 10*y for x in range(3)] for y in range(4)]
```

In this case, the inner array is going to be different every time. It will be equal to:

```
[10*y, 1+10*y, 2+10*y]
```

Where y is the `row` number. This gives a final output of:

```
[[0, 1, 2],
 [10, 11, 12],
 [20, 21, 22],
 [30, 31, 32]]
```

It is worth looking at how this might be implemented as a for loop:

```
outer = []
for y in range(4):
    inner = []
    for x in range(3):
        inner.append(x+10*y)
    outer.append(inner)
```

The final result in this case is `outer`.

Another thing that you can see from the expanded code is that the range of `x` can depend on `y`. For example, we could do:

```
a =[[x + 10*y for x in range(y)] for y in range(4)]
```

Which gives us:

```
[[],
 [10],
 [20, 21],
 [30, 31, 32]]
```

9.3.2 Creating a flat list

This code does something different:

```
[x + 10*y for x in range(3) for y in range(4)]
```

Clearly, we are only creating one flat list here, rather than a list of lists (there is only one `[]` pair). But we are still looping over `y` from 0 to 3, and for each `y` we are looping over `x` from 0 to 2. Here is what we get:

```
[0, 10, 20, 30, 1, 11, 21, 31, 2, 12, 22, 32]
```

It is the same numbers as before, but all in one list. For comparison, here is the equivalent for loop code:

```
outer = []
for x in range(3):
    for y in range(4):
        outer.append(x+10*y)
```

Notice that the leftmost loop in the comprehension corresponds to the outer loop in the for loop version. This might seem to contradict the previous example, but in that example we had two separate list comprehensions, one inside the other.

9.4 Summary

List comprehensions provide a simple, idiomatic way to create a list based on an existing iterable, such as a `range`, another list, or any other iterable.

The list comprehension allows a conditional filter to be applied, then calculates an expression based on the original sequence.

Although a list comprehension is procedural in style, it has limitations in the processing that can be applied, which make it far more declarative than an equivalent for loop.

In a later chapter we will meet generator comprehensions, which are similar to list comprehensions, but create a lazy iterator rather than a list as output.

10 Generators

We saw in the chapter on iterators that it is possible to create your own iterator, by defining a class with a couple of specific methods. That method works fine, but it involves a fair bit of boilerplate code, and required the logic of the iterator to be written in an inside-out style.

Generators provide a simple method of implementing many types of iterator, using a simple syntax and often a more intuitive software flow.

10.1 Example – alphabet iterator

In the iterators chapter we developed an iterator that simply returns the letters of the alphabet, one by one, and then stopped. In fact, we only returned the letters *a* to *e*. Here is the equivalent as a generator:

```
def alphabet():
    for c in 'abcde':
        yield c

for x in alphabet():
    print(x)
```

`alphabet` looks like a normal function, but actually it is a generator. The way to tell is that it has a `yield` statement instead of a `return` statement.

Within a for loop, `alphabet` is used in a similar way to `range`. If you used `range(5)` it would loop through the values 0 to 4. `alphabet` does something similar, but it loops through the characters *a* to *e*.

This generator is only for illustration. The loop would work perfectly well if you just use the string `'abcde'` in place of the `alphabet` call.

10.2 How a generator works

To understand how the generator works, we will open out the for loop like this:

```
def alphabet():
    for c in 'abcde':
        yield c

g = alphabet()

x = next(g)
print(x)

x = next(g)
print(x)
```

First, we call `alphabet`. Unlike a regular function, a generator returns a `generator object` that gets stored in `g`. The key thing here is that `alphabet` contains a `yield` statement

rather than a `return`. That is the thing that differentiates it from a normal function. Python knows to create a generator instead,

The `generator` object acts like an iterator. When we call `next` on it, it responds with the first value in the sequence, `'a'`. If we call `next` again it responds with the next value, `'b'`. And so on.

What is actually happening here? The first thing to realise is that the code in the body of the generator is not executed when you call `alphabet()`. In true iterator style, a generator uses lazy evaluation. It does nothing until you actually request a value.

The first time you call `next`, Python starts to execute the code in `alphabet`. It starts the for loop, entering the loop for the first time with the initial vale of `c` equal to `'a'`. Then it encounters the **yield** statement.

This causes Python to stop executing the `alphabet` code and return the value of `c` to the main calling code. But here is the critical thing – `yield` stores the exact state of the alphabet code before it exits.

Now the main code runs `print(x)` to print the value `'a'`. It then calls `next` again. But this time, instead of starting back at the beginning of the `alphabet` code, it jumps back into the code straight after the previous `yield` statement. The state is restored to the exact state it was when the yield statement executed.

`alphabet` loops back round for the next iteration of the for loop. This time it picks up the second character from the string, a `'b'` character. This get returned on the next yield.

This continues each time the main loop calls `next`. Eventually, the loop in the `alphabet` code is exhausted. Instead of calling `yield`, the `alphabet` code reaches the end. The `generator` object will throw a `StopIteration` error at that point, to notify the calling code that the iteration is complete.

As you can see, execution passes backwards and forwards between the calling code and the `alphabet` code. Generators are sometimes called *co-routines* for that reason. This contrasts with a normal function, which is sometimes called a subroutine. In a subroutine, the calling code passes entire control to the subroutine until it is finished. With a co-routine, control passes to and fro.

Don't get this confused with multithreading. In multithreading, two different sets of code can run at the same time (either on different cores or by time sharing one core). With co-routines there is only one thread of execution that just swaps between two different code paths in a totally predictable way.

10.3 Example – Fibonacci iterator

We also implemented a Fibonacci series iterator in the iterators chapter, so not of interest we will re-implement it here as a generator:

```
def fibonacci():
    c = 0
    n = 1
    while True:
        yield c
        c, n = n, c + n

for i in fibonacci():
    print(i)
    if i > 100:
        break
```

Since this generator produces an infinite sequence, so the generator loop use a `while True` loop rather than a for loop.

10.4 Chaining iterators

Now we will do something more useful. We will chain two iterators together, so that you get a single iterator that returns al the values of one iterator followed by all the values another iterator.

First, we will look at an identity generator. This provides a way to iterate over an existing iterator. Not very useful in itself (you could just iterate over the original iterator directly) but it is a step on the way.

```
def identity(it):
    for x in it:
        yield x

for i in identity(range(4)):
    print(i)
```

All `identity` does is iterate over it, yielding each value. Since `range(4)` creates the series 0, 1, 2, 3, our `identity` generator creates exactly the same series.

Chaining two iterators simply involves performing the identity operation on the first iterator, then doing the same this with the second iterator. This is quite easy in a generator:

```
def chain2(it1, it2):
    for x in it1:
        yield x
    for x in it2:
        yield x

for i in chain2(range(4), reversed(range(3))):
    print(i)
```

Here we are chaining `range(4)` – 0, 1, 2, 3 – and the reverse of `range(3)` – 2, 1, 0. This creates a single iterator that produces the series 0, 1, 2, 3, 2, 1, 0.

This code would be quite a lot more complicated using an iterable object, but it is quite simple and obvious using generators. But also, be aware that there is an existing `chain` function in the `itertools` library, covered in a later chapter.

10.5Generator comprehensions

A generator comprehension is similar to a list comprehension, which we met in an earlier chapter. The difference is that a generator comprehension uses lazy evaluation, which often uses less memory, and allows infinite iterators to be processed.

Converting a list comprehension to a generator comprehension is a simple matter of replacing the surrounding square brackets with round brackets. For example (from the list comprehension chapter) this code creates a list of strings `'0'`, `'1'`, `'2'`etc:

```
a = [str(i) for i in range(100)]
```

If you only need an iterator, not an actual list, you can do this:

```
g = (str(i) for i in range(100))
```

`g` is a generator object than delivers the series of values `'0'`, `'1'`, `'2'`etc. Unlike the list comprehension, these 100 values are not created in memory, which can be important if you are using much longer series.

10.5.1map variants

An advantage of generator comprehensions is that they can directly replace standard functions like `map`, useful if you want a slight variant. Here is a comprehension equivalent of a simple one parameter map of function `fn` over iterable `it`:

```
map(fn, it)
(fn(x) for x in it)
```

In this instance, `map` is probably the better option. But if you wanted to map `fn(x)+1` it looks different:

```
map(lambda x: fn(x)+1, it)
(fn(x)+1 for x in it)
```

It is often a matter of personal taste, and, whichever you choose it is usually better to use a consistent approach.

10.5.2filter-map variants

In a similar way you can replace a `filter`, or `filter-map` combination, with a comprehension:

```
map(fn filter(cmp, it))
(fn(x) for x in it if cmp(x))
```

Once again, if the function or comparison are slightly more complex, the comprehension has the advantage that you can just use normal expressions rather than using lambda functions.

You can also sometimes do special filtering operations, for example, this filter selects every second element from the incoming iterator:

```
(x for i, x in enumerate(it) if i%2==0)
```

This uses `enumerate` to get a count, `i`, for each element and only returns elements where `i` is even (`i` modulo 2 is zero).

To summarise, if you find yourself using a list comprehension but you don't need an actual list, consider using a generator comprehension to save memory. But if you can use standard functions like `map` or `filter` instead, that will usually be the better option.

10.6Summary

Generators are essentially a simple and convenient way to create your own custom iterators. As such they share all the benefits of iterators – lazy evaluation, pipelined processing and avoiding excessive memory usage in many types of process.

For simple iterators, generator comprehensions provide the benefits of iteration in the same one-line format as a list comprehension. Generator comprehensions should usually be preferred over list comprehensions in most cases, unless you specifically require a list as the end result.

11 Partial application and currying

This is the first of two chapters that will cover some additional techniques used in functional programming.

In this chapter we will revisit closures and composition, and also look at partial application and currying. These are all ways to create a new function from an existing function.

In the next chapter we will look at functors and monads. In addition, we will look at some functions offered by the `PyMonad` library that can do a lot if the work for you.

11.1 Closures

We have looked at closures quite extensively, with a whole chapter devoted to them, so all we will do here is a quick recap.

A closure occurs when:

- A function includes an inner function
- The outer function returns the inner function
- The inner function references some variables within the scope of the outer function.

In the case the returned inner function is called a closure. It can still access the values defined within the scope of the outer function, even though the outer function itself is no longer active.

11.2 Partial application

Partial application is a way of creating a new function based on an existing function but with some of the parameters already filled in with a chosen value.

For example, here is a closure based that creates partial applications of the standard function max:

```
def maxn(n):
    def f(x):
        return max(n, x)
    return f
```

Now, for example, if we call `maxn(0)`, it will return a closure of `f(x)` with the values of `n` fixed at 0. In other words, it will return a function that calculates `max(0, x)`. Here it is in action:

```
max0 = maxn(0)

print(max0(3))  # Equivalent to max(0, 3) -> 3
print(max0(-1)) # Equivalent to max(0, -1) -> 0
```

We have created our new function `max0` and a couple of test cases to prove that it does indeed calculate `max(0, x)`. In effect it replaces any negative value with 0.

Here is an example of using our partial application with map. In this case we are using `maxn` to create an anonymous function that gets used in the map call:

```
m = map(maxn(3), [1, 2, 3, 4, 5])
print(list(m))
```

The result is `[3, 3, 3, 4, 5]` – all values less than 3 are clamped at 3. Having defined our `maxn` function, it can be used to provide partial applications of `max` in a compact and expressive way.

11.2.1Functions with more variables

Now we will look at a function with more variables. We will use a function that calculates the value of a quadratic function:

$$y = ax^2 + bx + c$$

For given values of `a`, `b`, `c` and `x`. Here is a function that does this calculation:

```
def quad(a, b, c, x):
    return a*x*x + b*x + c
```

Now you will often be in a situation where a, b and c are fixed, and all we want to do is vary x. You can do this very easily with a closure:

```
def quad_abc(a, b, c):
    def f(x):
        return quad(a, b, c, x)
    return f
```

This allows us to create a partial function to calculate, for example:

$$x^2 - 3x + 2$$

Here is the code to do create this specific quadratic as `myquad`, and check it calculates the correct values (we know the value should be 2 when x is 0, and it has roots when x is 1 or 2):

```
myquad = quad_abc(1, -3, 2)

print(myquad(0))  # 2
print(myquad(1))  # 0
print(myquad(2))  # 0
```

We can also quite easily create a different partial application, for example suppose we wanted to apply values for a and c, but leave b and x as variables to be set later:

```
def quad_ac(a, c):
    def f(b, x):
        return quad(a, b, c, x)
    return f

myquad = quad_ac(1, 2)

print(myquad(0, 1))
print(myquad(1, 2))
print(myquad(2, -1))
```

So myquad is now a function that has a set to 1 and c set to 2. When we call myquad we pass in two arguments, containing values for b and x.

Clearly, by creating different closures we can choose to set any combination of a, b, c and x that we want is the partial application and leave the rest to be set when we actually call the function.

11.2.2functools.partial function

The functools module provides a function partial that can be used create a partial application of a function. Here is how you could use it to create a max0 function:

```
from functools import partial

max0 = partial(max, 0)

print(max0(3))  # 3
print(max0(-1)) # 0
```

Here is how you could use it with map, similar to the previous example:

```
m = map(partial(max, 3), [1, 2, 3, 4, 5])
print(list(m))
```

The advantage here is that you don't have to define a separate closure maxn. The code is more declarative and relies only on standard library functions.

The disadvantage is that the map call is slightly longer and more complex.

11.2.3functools.partial with more variables

The functools.partial function can be used to create a partial function for functions with more than two variables, for example our previous quad function. Here is how we would create a partial quad function with a, b, c set to 1, -3, and 2:

```
pquad3 = partial(quad, 1, -3, 2) # a, b, c = 1, -3, 2
print(pquad3(0))                 # x = 0
```

To be clear, quad takes 4 arguments. In the partial call, we have supplied the first 3 arguments to create a partial function pquad3. When we call pquad3, we must supply the final argument, x.

We can supply fewer arguments if we wish. In the case below, we just supply 2 argument to the partial call. This creates a partial function pquad2 where the first argument a has been set to 4, and b has been set to 1. The call to myquad1 must supply values for c and x.

```
pquad2 = partial(quad, 4, 1) # a, b = 4, 1
print(pquad2(3, 1))          # c, x = 3, 1
```

In the final example below, we just supply 1 argument to the partial call. This creates a partial function myquad1 where the first argument a has been set to 4. The call to myquad1 must supply values for b, c and x.

```
pquad1 = partial(quad, 4)    # a = 4
print(pquad1(1, 3, 1))       # b, c, x = 1, 3, 1
```

There is one thing you can't do with partials. You can't do the equivalent of myquad_ac, where you fix the values of a and c in the partial function. The syntax of partial doesn't allow you to pick and choose which arguments will have values applied.

11.2.4Applying keyword arguments

A partial function can apply keyword arguments too. Here is a simple example:

```
def make_print(sep):
    def f(*args):
        return print(*args, sep=sep)
    return f

print_csv = make_print(', ')
print_colon = make_print(':')

print_csv(1, 2, 3)         # 1, 2, 3
print_colon(1, 2, 3)       # 1:2:3
```

The make_print function returns a partial application of the standard print function, with the sep keyword argument set to the supplied value. sep is a string that gets inserted between the arguments (this is standard print functionality).

Notice also that the inner function f in make_print accepts *args. This means that the partially applied print function will accept multiple arguments and print then all, separated by the sep string.

We use make_print to create two new functions, print_csv prints values separated by commas, and print_csv prints values separated by colons.

You can use keyword arguments with `partial` too. Here is an alternative way to create our two print functions:

```
from functools import partial

print_csv = partial(print, sep=', ')
print_colon = partial(print, sep=':')
```

11.2.5 Don't overlook the simpler solutions

Before you dive in with a partial application solution, always bear in mind that there may be alternatives. Let's look at some of the possible alternatives for creating a `max0` function – a functional of `x` that returns 0 or `x`, whichever is larger.

Here is the original solution:

```
def maxn(n):
    def f(x):
        return max(n, x)
    return f

max0 = maxn(0)
```

If you only ever intend to use `max0` (you will never need a `maxn` with a value of `n` other than zero), it would be possible to define `max0` directly (no need to create a `maxn` closure):

```
def max0(x):
    return max(0, x)
```

If you only need to use this function once, you can simply use a lambda:

```
map(lambda x: max(0, x), [1, 2, 3, 4, 5])
```

This doesn't mean you should never use a partial function is these situations, just that you should always be aware of the alternatives and choose whichever makes you code robust and readable.

11.3 Currying

Currying is similar to partial application but takes a slightly different approach. Before we start, it is probably worth noting that the term currying derives from the name of Haskell Curry, the mathematician who did a lot of work on the theory behind currying. The Haskell programming language is also named after him.

Whereas the Python term *pickling* (serializing an object for later retrieval) is vaguely analogous to the culinary process it is named after, the term currying has nothing whatsoever to do with food. Searching for parallels there will only lead to confusion.

11.3.1 Curried version of quad

Currying isn't part of standard Python, but there are several open source functional programming libraries around. We will use `PyMonad`, one of the better-known libraries. You will need to install it as described in the chapter *Useful libraries*.

We curry a function by applying the `@curry` decorator to its declaration. Here is how we would curry our `quad` function (we will call it `quadc` to distinguish it, but that isn't something you would normally need to do):

```
from pymonad import curry

@curry
def quadc(a, b, c, x):
    return a*x*x + b*x + c
```

Now we can still use `quadc` is the same way as `quad`:

```
y = quadc(1, -3, 2, 0)
```

But we can also call `quadc` with just 3 arguments. In that case it will return a callable object (an object that can be used exactly like a function object). This callable object, `f`, can be called with a single argument to obtain the result, `y`:

```
f = quadc(1, -3, 2)
y = f(0)
```

This is very similar to creating a partial function as we did with the original quad function:

```
pquad3 = partial(quad, 1, -3, 2)
y = pquad3(0)
```

We can also call `quadc` with 2 arguments. It will return a callable that needs 2 extra arguments:

```
f = quadc(1, -3)
y = f(2, 0)
```

Again, this is similar to creating a partial function with `a` and `b` set, and then passing `c` and `x` in later. And, of course, we can call the curried function with a single argument:

```
f = quadc(1)
y = f(-3, 2, 0)
```

In fact, currying is even more flexible. You can split the function call more than once, for example:

```
f = quadc(1)
g = f(-3, 2)
y = g(0)
```

In this case, `f` is a callable object that wants 3 arguments. If we give it just two arguments, we get another callable object `g` that wants 1 argument. If we call `g` with one argument we get the actual result, `y`.

11.3.2 When to use currying

Currying and partial application both do similar jobs. We will use them both in an example using `map`.

Here is the example using partial application (using the `functools partial` function):

```
c = [1, 2, 3, 4, 5]
x = [2, 4, 6, 8, 10]
m = map(partial(quad, 1, 2), c, x)
```

We have created a partial function of `quad`, setting `a` to 1 and `b` to 2. We then map this function onto two lists containing the `c` and `x` values.

How would we do this with currying? We would use our curried version of `quad`, which we called `quadc`. After that, the code is fairly similar:

```
c = [1, 2, 3, 4, 5]
x = [2, 4, 6, 8, 10]
m = map(quadc(1, 2), c, x)
```

Notice that this only works because we are using the `@curry` decorator. This allows us to use `quadc(1, 2)` as a function that takes two arguments.

Looking at these two implementations, you might think that the second one is clearly better because it doesn't involve explicitly calling the `partial` function.

There is are potential downsides, though. The `@curry` decorator does a lot of stuff behind the scenes. It creates a `quadc` function that can be called in many different ways, for example:

```
quadc(1, 2, 3, 4)
quadc(1,2)(3, 4)
quadc(1, 2)(3)(4)
quadc(1)(2, 3, 4)
```

One point here is that calling a curried function is less efficient than a normal function call. This is not usually an important consideration, because if there are any parts of your program that require extreme efficiency you probably shouldn't be writing them using FP, and maybe not even Python. But it is worth keeping in mind.

The more important consideration is that currying creates functions that behave quite oddly. Of course, there is nothing magic going on here, the `@curry` decorator is using standard Python. `quadc` and its inner functions just handle different numbers of arguments. But if you are not familiar with the decorator, or maybe you don't even know that `quadc` was created using the decorator, then the code above could be unexpected and confusing.

Compare this with the `partial` function. When you use `partial`, it is quite explicit. You can see that something is being done, and you can look up `partial` if you are not familiar with it.

As a rule of thumb, if you are writing a large amount of functional code, and using currying throughout, it is reasonable to expect anyone reading the code to be familiar with how it works. It is fine to use currying.

But if you are working mainly with partials, or maybe working with code that isn't predominantly functional, then throwing in the odd curried function here and there is probably going to cause more confusion than it is worth.

11.4Composition

It is quite common in programming to have one function operate on the result of another function. For example, to calculate the square of the sine of `x` we use:

```
square(math.sin(x))
```

To create an iterator that counts down from `n-1` to 0 we use:

```
reversed(range(n))
```

We refer to this as composing functions, or composition.

It is generally simpler to stick to the cases where each function accepts a single argument. If that is not the case, we can often use partial application to solve this. For example, here is a composition to calculate `2 + 3*x`:

```
from operator import add, mul

add(2, mul(3, x))
```

The example above composes 2 functions that each take two arguments, making it difficult to generalize. We can improve this using partial functions:

```
from functools import partial

f = partial(add, 2)
g = partial(mul, 3)

f(g(x))
```

Or, in one line (but a little difficult to read):

```
partial(add, 2)(partial(mul, 3)(x))
```

11.4.1Creating a compose function

The examples above are quite procedural – we are composing function by writing code that calls one function then calls another function with the result. We are writing code that describes how to do the composition, rather than simply declaring that we want composition to happen. In the earlier chapter *Closures,* we created a simple `compose` function that accepted 2 functions and returned a function that composed them:

```
def compose2(f, g):
    def fn(x):
        return f(g(x))
    return fn
```

In this section we have renamed this function to `compose2`, for reasons that will become clear. `compose2` is a function that composes exactly 2 functions.

Instead of using `reversed` and `range` to directly create an iterator that counts down, we can compose those 2 functions to create a new function, `countdown`, that creates an iterator that counts down:

```
countdown = compose2(reversed, range)
countdown(n)
```

This is even more useful when we start using partial functions. Instead of directly using the partials of `add` and `mul` as before, we can create a new function by composing them:

```
addmul = compose2(partial(add, 2), partial(mul, 3))
addmul(x)
```

It is much clearer in this case what is going on.

There are several open source implementations around, but before we look at those, we will extend our `compose2` function to accept more than 2 functions.

Let's say we want to compose functions `p`, `q`, `r`, `s`. We want to create a single function that does:

```
p(q(r(s(x))))
```

We could create this by repeated use of our existing `compose2` function, like this:

```
f = compose2(p, q)    # f calculates p(q(x))
g = compose2(f, r)    # g calculates p(q(r(x)))
h = compose2(g, s)    # h calculates p(q(r(s(x))))
```

To understand the next step, let's imagine replacing the values with numbers `a`, `b`, `c`, `d`, and the `compose2` function with `add`. The chain of operations looks like this:

```
x = add(a, b)
y = add(x, c)
z = add(y, d)
```

Or more succinctly:

```
((a + b) + c) + d
```

This is simply the sum of all the numbers. Or put another we, we have *reduced* the list of number using the `add` operation. See the section on `functools.reduce` in the chapter *Reducing iterables*.

The same thing is also true for composition. To compose a list of functions, we simply reduce the list of functions using the `compose2` operation:

```
def compose(*f):
    def compose2(f, g):
        def fn(x):
            return f(g(x))
        return fn
    return functools.reduce(compose2, f)
```

Here, we have included our original `compose2` closure as an inner function of `compose`. This makes it a private – we don't need `compose2` to be accessible anymore, because compose

can compose2 functions, or 3, or 4 etc. We only use compose2 internally to reduce the list of functions.

The return value of compose is result of reducing the list of input functions, f, with the compose2 function, resulting in a single composite function.

One final point, the reduce function will give an error if it is called with an empty list, unless we supply a third parameter as an initial value. If we want to avoid this, we need to supply a suitable value. But what should we use?

The initial value should be the identity value for the operation we are using. If we were using the add operation, we would use 0, because x plus 0 is x. If we were using the mul operation it would be 1, because x times 1 is x. In the case of composition, we want a value that acts as an identity value when composed with any function. This value is the function f(x) that returns x. To avoid the error, change the last line of compose to:

```
return functools.reduce(compose2, f, lambda x: x)
```

11.4.2Existing libraries supporting composition

There are quite a few open source libraries that provide functional programming support, including a compose function. funcy and fn.py are two libraries that can be found on github.com or installed with pip.

We will concentrate on PyMonad in this section. This provides composition of curried functions (see the PyMonad @curry decorator described earlier in this chapter).

Rather than using a compose function, PyMonad uses a compose operator, *. Here is how it works. First, we need to create curried versions of the built-in reversed and range functions:

```
from pymonad import curry

@curry
def reversedc(x):
    return reversed(x)

@curry
def rangec(n):
    return range(n)
```

Once these functions are defined, composing them is quite neat:

```
countdown = reversedc * rangec
```

Since all the functions are curried, it makes it very easy to compose partial functions. Here are curried versions of add and mul:

```
@curry
def addc(a, b):
    return a + b

@curry
def mulc(a, b):
    return a * b
```

And here is our `addmul` function from earlier:

```
addmul = addc(2) * mulc(3)
```

This is clearly a lot more readable than the previous version:

```
addmul = compose2(partial(add, 2), partial(mul, 3))
```

11.5Summary

In this chapter we have looked at some of the basic ways of building new functions from existing ones. These are some of the basic building blocks of functional programming.

- Closures – are function factories, capable of building new functions in many different ways.
- Partial application creates new functions based on existing functions with some of their original arguments already assigned.
- Currying declares functions in a way that makes partial application and composition much simpler.
- Composition is a declarative way to create new functions from chained calling of existing functions.

12 Functors and monads

Functors and monads are two important types of object in functional programming. They have their roots in some fairly abstract maths, but here we will focus on their practical benefits.

Essentially what they do is wrap a value. The wrapper then controls how functions are applied to the value. It allows us to extend the capabilities of ordinary functions, for example to make them work automatically with collections, or to allow them to cope with values that might not exist (called *optionals* in some languages).

In fact, there are three related types:

- Functors are the basic type. A functor has a `map` method that allows it to control how functions are applied.
- Applicative functors (often just called applicatives) are a subset of functors. Applicative functors can do everything functors can do, but also have extra capabilities. All applicatives are functors, but not all functors are applicative functors.
- Monads are a subset of applicable functors. They can do everything an applicative functor can do, end even more.

These objects are not part of standard Python, but there are several libraries you can use. This chapter will be using `oslash`, a library that has quite a nice implementation of functors. See the chapter *Useful libraries*.

Most of the general stuff here about functors applies to other libraries and even other languages. Most of the syntax is `oslash` specific, which is based largely on Haskell.

You might find that you don't uses functors and monad all that much. They are used in pure functional languages to handle situations that are difficult to handle without procedural code. In Python, you have procedural coding methods available, so you might often use those rather than monads. But it is useful to know that they exist.

12.1 Functors

We need to be a little careful as the term functor has different meanings in different branches of computing. For example, it is sometimes used simply to refer to a function object. That is not what we mean here

This section describes how we use the term throughout this book. If you are familiar with the Haskell programming language, `oslash` functors are based closely on those. If you are not familiar with Haskell, don't worry, this section will tell you all you need to know.

A functor is an object that wraps a value and provides a `map` method that can be used to apply a function to that value.

Most of the functors we discuss in this section are also monads, so they have all the functionality of functors, applicatives and monads. So, don't be confused when the functor types we discuss here are also used as examples of applicatives or monads later on.

12.1.1 The Just functor

The `Just` functor is a simple wrapper around a value. We can create a `Just` functor like this:

```
from oslash import Just

a = Just(3)
print(a)
```

This code prints `Just 3`, to indicate that it is a `Just` wrapper around the value 3. Now if we try to apply the `operator neg` function to this object, we will get an error because `neg` doesn't know how to deal with functors:

```
from operator import neg

neg(a)  # Error!
```

What we must do instead is use `map` to apply the function. `Just` has a `map` function, because it is a functor. Here goes:

```
b = a.map(neg)
print(b)
```

This prints `Just -3`. That is because `Just.map` knows how to apply a function to its wrapped value and return a wrapped result.

You can also apply a function to a functor using the % operator. This is an infix operator that accepts the function first, followed by the functor it is being applied to.

```
b = neg % a
```

12.1.2 The Nothing functor

The `Nothing` functor is very simple. It represents nothing, in a similar way to the `None` type in standard Python. Although we said earlier that a functor wraps a value, `Nothing` is the exception. It doesn't wrap a value; it is just nothing.

We can create `Nothing` like this:

```
from oslash import Nothing

a = Nothing()
print(a)
```

This prints `Nothing`, as you would expect. Here is what happens when we apply a function to a `Nothing` value:

```
b = neg % a
print(b)
```

The result again is `Nothing`. In fact, the result of applying *any* function to `Nothing` is always `Nothing`.

12.1.3The List functor

Our next functor is `List`. It wraps a list of values, like this:

```
from oslash import List

a = List([1, 2, 3])
print(a)
print(type(a))
```

This prints:

```
[1, 2, 3]
<class 'oslash.list.List'>
```

The list contents are printed in a similar way to a standard Python list, but the type shows that it is an `oslash` object.

Now let's try applying a function to this list. We will use neg from the operator module – this in the function equivalent of the negation operator –. Here is the code:

```
from oslash import List
from operator import neg

a = List([1, 2, 3])
b = neg % a
print(b)
print(type(b))
```

This produces:

```
[-1, -2, -3]
<class 'oslash.list.List'>
```

It has applied the `neg` function to every element in the list. That is what the `List` functor does, you can use any function and `List.map` (called by the % operator) will apply it to every element in the list.

12.2Applicative functors

An applicative functor *wraps a function*. It can apply its function to another functor, for example:

```
from oslash import Just
from operator import neg

a = Just(3)
f = Just(neg)
b = f.apply(a)
print(b)
```

The first thing to know is that this code only works because `Just` isn't only a functor, it is also an applicative functor. An ordinary functor doesn't have an `apply` method. So, having

wrapped our `neg` function in the applicative functor `f`, we can apply it to `a`. The result is the same as before, a functor `Just -3`.

You can use the * operator as an infix version of `apply` (just like % is an infix version of `map`), so we could write:

```
b = f * a
```

12.2.1 Functions with more than one argument

Up until now we have only dealt with functions that take exactly one argument. What if we wanted to use a function like `operator add`?

```
a = Just(3)
b = add % a
print(b)
```

This prints:

```
Just functools.partial(<built-in function add>, 3)
```

This is very promising. A partial of `add`, with 3 applied, wrapped in a `Just` applicative! We know how to apply the second argument, using the * operator:

```
c = b * Just(6)
print(c)
```

Which gives us `Just 9`.

Let's see how that works with a function of 4 arguments, such as the `quad` function from earlier chapters (don't worry, it doesn't really matter what the function actually does):

```
a = quad % Just(1) * Just(3) * Just(2) * Just(0)
```

You can think of the % as being like the opening bracket and the * being like a comma between arguments. There is no closing bracket, analogies aren't always perfect. Or you could just wrap `quad` in a `Just` functor and use * all the way through:

```
a = Just(quad) * Just(1) * Just(3) * Just(2) * Just(0)
```

It is important to understand what is happening here. Let's break it down into stages and show the value of a at each stage:

```
a = Just(quad) # Just <function quad >
a = a*Just(1) # Just functools.partial(<function quad, 1)
a = a*Just(3) # Just functools.partial(<function quad, 1, 2)
a = a*Just(2) # Just functools.partial(<function quad, 1, 2, 3)
a = a*Just(0) # Just 2
```

Each step creates a new partial function, wrapped in a Just functor, with the parameters supplied so far set in the partial function. This works a little like currying.

12.3Monads

A monad wraps a value in a similar way to a functor. However, a monad has an additional function called `bind` that:

- Accepts a single parameter.
- Returns a value wrapped in a monad.

Unlike a `map`, the `bind` function itself is responsible for wrapping the return value. This means the `bind` function can decide what sort of monad to wrap the result in.

Here is an example, the function `oneover` returns `1/x`, wrapped in a `Just` monad:

```
from oslash import Just, Nothing

def oneover(x):
    ret = 1/x
    return Just(ret)

a = Just(2).bind(oneover)
print(a)

a = Just(0).bind(oneover)
print(a)
```

The first call, binding `oneover` to the value `Just 2` correctly returns `Just 0.5`. The `Just` monad unwraps the value 2, and passes it to `oneover`, which returns `Just 0.5`.

The second call, however, passes 0 to `oneover`, which results in a divide by zero exception – not exactly what you want from a pure function.

There is a solution. `oneover` has the choice of what sort of monad it returns. So, we could catch the exception and return a `Nothing` monad:

```
def oneover(x):
    try:
        ret = 1/x
    except:
        return Nothing()
    return Just(ret)
```

This works much more sensibly. The first call returns `Just 0.5`, the second returns `Nothing`.

12.4Summary

Functors, applicable functors and monads are mainly used by pure functional programming languages to handle situation such as errors, exceptions or order dependent operations in a clean way. This is less of a problem for Python, since procedural code can be used for the same purpose.

This chapter gave an overview of the use of these features with the `oslash` library. If you are interested in gaining a greater understanding of the context of these features, it would be a good idea to learn more about a pure functional language such as Haskell.

13 Useful libraries

We have mentioned several libraries in other sections of this book. This chapter tells you where to find them and lists further areas to explore.

13.1itertools

`itertools` is a standard Python library – it is part of Python and doesn't need any extra installation.

It contains a number of useful extra functions for iteration. Here are some of the highlights, refer to the documentation on python.org for a full list.

13.1.1Infinite iterators

These are iterators that go on forever.

`count(start, [step])` produces an infinite series of incrementing values, with an initial value `start`. For example:

```
count(0)   # Creates 0, 1, 2 ...
```

It behaves like range, but with the end value set to infinity. There is an optional step value:

```
count(5, 2)   # Creates 5, 7, 9 ...
```

`repeat(x, [n])` produces and infinite series of the value `x`, repeated over and over. The optional value `n` will cause the sequence to stop after `n` iterations.

`cycle(it)` creates an infinite series by repeating the values in `it` (an iterable) indefinitely. For example:

```
cycle([1, 2, 3])    # Create 1, 2, 3, 1, 2, 3, 1, 2, 3...
```

13.1.2Other iterators

itertools contains a set of other useful iterators. Some of these are useful variants on built-in functions.

`zip_longest(p, q ..., fillvalue=None)` zips a set of iterables. it works in a very similar way to the normal `zip` function. The difference is that `zip` stops when the shortest iterable runs out of data, whereas `zip_longest` continue until the longest iterable runs out of data. Missing values are set to `fillvalue`, which defaults to `None`:

```
a = [1, 2, 3]
b = [10, 20]
zip(a, b)         # Creates (1, 10), (2, 20)
zip_longest(a, b) # Creates (1, 10), (2, 20), (3, None)
```

`starmap(fn, it)` is very similar to `map`. The difference is, if `fn` takes more than one argument, `map` expects a set of iterables, one per argument. `starmap` expects a single iterable

containing tuples of `n` values. This is useful is the data happens to already be in that format – rather than having to use `zip` to reformat the data, you can just use `starmap` directly.

If your data is available as a set of iterables, use `map`:

```
map(add, [1, 2, 3], [4, 5, 6])
```

If your data is available as a single iterable of tuples, use `starmap`:

```
starmap(add, [(1, 4), (2, 5), (3, 6)])
```

`filterfalse(fn, it)` is exactly the same as `filter`, the sense of the test is reversed. `filter` keeps elements where `fn` is true, and discards elements where `fn` is false. `filterfalse` keeps elements where `fn` is false, and discards elements where `fn` is true.

There are some additional functions that are worth a look too:

- `accumulate` is similar to `sum`, but it keeps a running total of values.
- `chain` joins two or more iterables into a single iterator.
- `tee` splits one iterable into two or more separate iterators.
- `takewhile` is similar to `filter`, it returns all the values from an iterable until its function evaluates to false. Unlike `filter`, `takewhile` stops completely after the first false value.
- `dropwhile` is the opposite of `takewhile`. It ignores all values until the function evaluates to false, and then returns all values after that.

13.1.3Combinations

`itertools` also provides several functions for creating all permutations and combinations of the elements of am iterable.

13.2functools

`functools` is another standard Python library than requires no additional installation. We have used several functions from this library in earlier chapters:

- `lru_cache` for memoization.
- `reduce` for the map-reduce pattern.
- `partial` for creating partial functions.

13.3PyMonad

`PyMonad` is an open source library that supports functional programming techniques that are not supported by the standard Python libraries.

We have used it to illustrate currying and composition. As its name suggest, it also provides functor and monad implementations, although in this book we have opted for the `oslash` library to illustrate those concepts.

You can install `PyMonad` using:

```
pip install pymonad
```

Documentation can be found on the `PyMonad` pages on *github.org* and *pypi.org*. There are also various articles on `PyMonad` that can be found around the internet.

13.4oslash

`oslash` is another open source library that supports functional programming. Again, it is not a standard library and will need to be installed separately.

We have used it to illustrate functors and monad because it has quite a clean and simple implementation.

You can install `oslash` using:

```
pip install oslash
```

Documentation can be found on the `oslash` pages on *github.org* and *pypi.org*. It is a slightly less well-known library, so there is less supporting material available from other sources, but it is reasonably well documented on its official pages.