Functional Programming in Python

2nd Edition

Martin McBride

Functional Programming in Python
by Martin McBride

Published by Axlesoft Ltd
info@axlesoft.com

Preface

Functional programming is one of the hidden gems of the Python language. Many developers are familiar with procedural and object-oriented development but tend to avoid functional programming. This is understandable as functional programming seems to require a complete paradigm change to a new programming language with a vast array of new terms and concepts. Added to that, traditional functional languages, such as Lisp, aren't exactly beginner-friendly.

Python is the polar opposite, known as a language that is very easy to learn and quick to code in. But if you take a slightly closer look, it has a rich assortment of functional programming tools that can be mixed with procedural or object-oriented code in a very natural and intuitive way.

I started using FP techniques in my work a few years ago, and in 2019 I gathered together some of the techniques I had learned, to create the first edition of this book.

Five years on, I have decided to create a second edition of the book. The overall structure of the book is unchanged, but I have added more examples and details throughout. I have also switched to using LaTeX rather than MS Word, which I think has improved the layout.

Technical details

This book was written in LaTeX, using TeXstudio[1], an open-source LaTeX authoring system.

All the diagrams in the book were created in Python, mainly using the generativepy[2],

[1] https://www.texstudio.org/

[2] https://github.com/martinmcbride/generativepy, https://pypi.org/project/generativepy/

an open-source maths visualisation library.

About the author

Martin McBride is a software engineer with forty years of experience developing software for many applications including medical imaging, maths visualisation, image processing, computer graphics, data compression, real-time data acquisition, and machine control systems. Much of his work has been rooted in mathematics.

Martin has a BA in Physics from Oxford University. He has written many articles on maths and software engineering (on medium.com and other websites) as well as several other books, including *Computer Graphics in Python* and *NumPy Recipes*.

Contact

If you would like to be updated when I publish other books and articles, please join my Substack newsletter. I regularly post free articles on there, as well as news about other projects.

My YouTube channel contains lots of animated videos covering various maths topics, including calculus.

Substack:	**graphicmaths.substack.com**
YouTube:	**www.youtube.com/@graphicmaths7677**
LinkedIn:	**www.linkedin.com/in/martin-mcbride-0014b5257**
Books:	**www.amazon.co.uk/stores/Martin-McBride/author/B07XSF9NFZ**
	leanpub.com/u/martinmcbride
Articles:	**medium.com/@mcbride-martin**
	graphicmaths.com

Finally, if you enjoyed this book, please leave a comment or review on Amazon, Leanpub, or wherever you purchased it. It helps to make the book more visible to others who might also find it useful.

Contents

Chapter 1

Introduction

Python supports several programming *paradigms* – procedural programming, object-oriented programming (OOP), and functional programming (FP). Of these, FP is probably the least understood and the least used. However, 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 in Python 3.12, although most will work with earlier or later 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 what 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 lines 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, have well-defined behaviour, and can be tested independently, it is much easier to write complex systems using OOP.

If you have used other OOP languages such as Java or C++, you will be familiar with objects such as strings and lists. Python has these objects too, but in Python, everything is an object, even things you might not expect to be. For example, numbers are objects, and so are functions.

Functional programming (FP) uses functions as the main building blocks. Unlike procedural programming, the functional paradigm treats functions as first-class citizens that can be passed into other functions as parameters, allowing new functions to be built dynamically as the program executes. Python allows this because, as noted earlier, functions are objects.

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 us 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, and returned from as a result other functions. 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. As mentioned above, 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:

- 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 in 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 absolutely repeatable – every time the function is called with a particular set of inputs, it must always produce exactly 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 an alternative version of the previous characteristic – prefers pure functions – 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 we pass a list into a function, the function can alter it. If we pass a tuple into a function, that is impossible because tuples are immutable.

FP prefers iterators over lists. An iterator is an object that provides access to a collection of data. An iterator can only read data one element at a time, it cannot 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 we use `map` to apply a function to a data stream, the meaning is clear and unambiguous. If we define a procedural function that loops over the data and applies the function, anyone working on the code in the future will need to read and understand the code to check exactly what it is doing.

There are often fewer bugs. FP uses standard functions that are well-tested, rather than ad hoc loops that might contain bugs. This means it is generally more reliable.

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

Multiprocessing can be applied easily. For example, if we are applying a pure function to a data stream, we 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, described in section 8.8, does this very effectively. If we 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 that 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 commonly used in Python, but we will cover this in chapter 12, *Functors and monads*.

FP has a learning curve. It is probably true to say that fewer programmers are experienced in functional programming than in some other paradigms. We usually write a function to do a particular task – it is a conceptual leap to move to the idea of writing a function that creates a function to do a particular task. 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 do. It is worth doing a sanity check for very large problems. See the example later of the recursive implementation of the 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.

Chapter 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 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 Python statements:

```
a = "apple"
b = "pear"
```

Now we often say, loosely, that the string **"apple"** is stored in the variable **a**, and the string **"pear"** is stored in the variable **b**. But that isn't a completely accurate explanation. In reality, 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 is shown in figure 2.1.

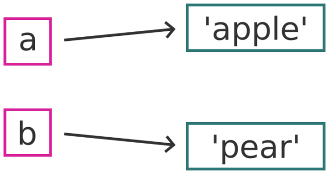

Figure 2.1: Objects are stored in memory, variables point to objects

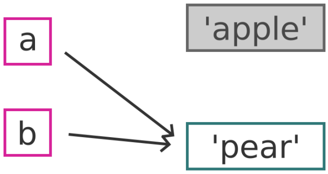

Figure 2.2: Old objects remain in memory until garbage collection reclaims them

To understand why this matters, consider what happens if we assign **b** to **a**, like this:

```
a = b
```

Now we would expect **a** and **b** to both have the same value, **"pear"**. But they don't both *contain* the string **"pear"** – how could two variables both contain the same object? No, of course, they both simply *reference* (or *point to*) the string **pear**. There is only one string, stored in memory somewhere. **a** and **b** both reference that same object in memory. This is illustrated in figure 2.2.

The previous **apple** string is still in memory, but we can no longer access it in any way. Python knows that this data is no longer needed, and will eventually free up the memory it occupies, so it can be used for something else. This is called *garbage collection* – periodically Python checks for any objects that are stored in memory but are no longer referenced from anywhere. We say these objects are *unreachable* because there is no longer any possibility of our code accessing them.

2.2 Storing functions

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

```
a = 10
print(a)        # 10

def square(x):
    return x*x

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

Looking at the code, variable **a** is initialised by assigning a value to it and accessed by directly referencing it:

```
a = 10       # Initialising a
print(a)     # Accessing a
```

Whereas the function **square** is created by the **def** keyword and accessed (ie called) using round brackets **()**:

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

b = square(a)    # Accessing square
```

In fact, **a**, **b** and **square** are all just variables. The **def** block is just a 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.

2.3 Inspecting objects

To further illustrate this, let's print out some details of **a** and **square**. We will display the **type**, **id** and string representation of each object:

```
print(type(a))   # <class 'int'>
print(id(a))     # 94773689623752
print(str(a))    # 10
```

The **type** function returns the type of the object, which is **<class 'int'>**.

The **id** function returns the id of the object. This is just an integer that is unique to that particular object. The value remains the same for the entire lifetime of the object. However, if we run the program a second time we would most likely get a completely different value for **id(a)**.

The string representation, for an **int**, is just a string containing the current value of the integer, which is 10 in our example code.

Since **square** is also an object we can print out its **type**, **id** and string representation, just like we did with **a**:

```
print(type(square))   # <class 'function'>
print(id(square))     # 139809144997024
print(str(square))    # <function square at 0x7f27da6c84a0>
```

This time the **type** of the object is **<class 'function'>** because it is a function. The **id** is, again, a unique number. And its **str** representation is a string that tells us that the object is a function, called square, that exists at a particular memory location (which again will probably be different every time we run the function). In other words, **square** behaves much like any other object.

2.4 Aliases

Earlier we used this example:

```
a = "apple"
b = a
print(a[0])       # Prints "a"
print(b[0])       # Prints "a"
```

As we saw earlier, this code creates one string, and both **a** and **b** reference it. We call them aliases – different names for the same data. When we then print **a[0]** and **b[0]**, they both refer to the first character in the string.

We can have any number of aliases for any object, which are all equally valid. For example **a** doesn't have any special status because it was created before **b**.

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, we could create an alias of **print**, like this:

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

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

It is quite rare to use aliases directly. It sometimes happens with certain specific libraries, for example, the NumPy library is often shortened to **np**, using **import as**:

```
import numpy as np
```

This import statement means that, in our code, we must always use **np** to access the **numpy** library. Almost everyone who uses the NumPy library uses this method, even the NumPy official documentation does it. But that is an exception, based on the nature of that library and how it is used. In general, using aliases in that way is rarely a good idea.

However, we will often use aliases indirectly. 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.

2.4.1 Redefining a function

Since functions are essentially variables that happen to hold function objects, we 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 unless we have a good reason to do it. Here is a simple example of what can happen:

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

def b():
    a()

b()                 # Prints 1

def a():
    print(2)

b()                 # Prints 2
```

We have defined a function **a** that prints 1. We then define a 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 we 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.

2.5 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)                  # Prints 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)         # Prints 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. Notice that we use the syntax **inch2cm**, without parentheses, to specify the **inch2cm** function object.

If we had used **inch2cm()**, with parentheses, that would *call* the function (which isn't what we want at all).

Now suppose 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)           # Prints 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)         # Prints 2 => two
```

The interesting thing here is that **i2text** doesn't use the same types as the previous functions. It accepts an 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 the supplied function and returns whatever comes back.

This was a very simple example, now we will look at a more realistic example of using a function object.

2.5.1 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 less than operator **<** 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 differently – 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 in the list, and the list is sorted based on the return value.

If we want to sort a list of strings by increasing length, we need to use a function that accepts a string and returns 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"]
```

Notice, as before, we use **len** to reference the function object, rather than **len()** which would call the function.

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. Suppose we wish to sort them 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 we can see from the result, this sorts the rectangles in order of area.

We will cover **sorted** in more detail in chapter 7 where we cover transforming iterables.

2.6 Lambda functions

In Python, if we need to pass a value into a function, we have two choices. We can either assign that value to a variable, or we can pass it in directly. For example:

```
# Assign values to a and b
a = 3
b = 5
print(a*b)

# Use values directly
print(2*4)
```

In the first case, we assigned the values 3 and 5 to variables **a** and **b** before using them. We could say that we *named* these values.

In the second case, we just used the values 2 and 4 directly, without assigning them to variables. We could say that these values are unnamed, or *anonymous*.

But what about functions? When we define a function using **def**, we always have to give the function a name. If we tried to define a function without a name, we would get a syntax error. For example, if we tried to define a simple adding function without naming it, the code wouldn't compile:

```
def (c, d):                #ERROR
    return c + d
```

This makes sense in most cases. Why would we want to create a function without a name? How would we call it?

But in functional programming, we sometimes create a function just to pass it into another function. For example, the **area** function we defined in the previous sorting example is defined and then immediately passed into the **sorted** function as a parameter. If that is all we ever do with the function, why should we need to name it?

This turns out to be quite a common requirement, so most function programming languages provide a way to create unnamed functions. They are often called *lambda functions*, for historical reasons[1]. A lambda function is simply an anonymous function.

[1] The name *lambda* comes from the mathematical term *lambda calculus*, which is part of the mathematical basis of functional programming

In Python, we create a lambda function using this syntax:

```
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 we 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 we create with a lambda expression is exactly the same as a function created with **def**, it just doesn't have a name.

It is possible to give a lambda function a name. We can 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**.

In general, if you want to create a named function it is better to just use **def** in the normal way. There isn't any advantage in using lambda to create a named function, except that it uses one less line of code. But we should also consider the disadvantage – it could be potentially confusing Someone reading our code might be left wondering why we used a lambda function. It is also more difficult to add documentation comments to a lambda function. This technique has its place, for example, if we are defining a trivial function that will used a few times in nearby code. However, if we need to name a function, lambdas are usually best avoided.

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

```
# No arguments returns current day of week 0-6
lambda: datetime.datetime.today().weekday()

# Adds x and y
lambda x, y: x + y

# Performs a calculation on a, b, c, d
lambda a, b, c, d: a*b + c*d
```

We will be 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 our function cannot be expressed in one line, we 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 we 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 often better to define a normal function, rather than repeating the lambda.

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

By the way, since a lambda is a function object, we 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 we could just write:

```
a = 3 + 1
```

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

2.7 Functions as return values

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

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

f = add1()
print(f(2))               # Prints 3
```

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 directly. This gets a lot more useful in chapter 5 when we introduce closures.

2.8 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
```

This could simply be replaced with **operator.add** – a function that takes two values and adds them together (exactly what the lambda is doing). Using a standard function is shorter and more declarative.

We 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 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 chapter 11.

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 we 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.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)                   # Returns k[2], ie 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 the 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 7.

2.9 Summary

To summarise, here are the various ways we can obtain function objects to use in our 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 function factories. A function factory allows us to create new functions based on particular criteria.
- Objects that implement the *magic method* **`__call__`** can be used as function objects.