# Object-Oriented Programming in Python Documentation

***Release 1***

**University of Cape Town and individual contributors**

February 20, 2017

**1 Introduction** **3**
- 1.1 What is a computer? 3
- 1.2 History of computers 6
- 1.3 Programming a computer 7
- 1.4 Programming languages 11

**2 Python basics** **15**
- 2.1 Introduction 15
- 2.2 Getting started with Python 15
- 2.3 Essentials of a Python program 17
- 2.4 Integers 23
- 2.5 Floating-point numbers 25
- 2.6 Strings 26
- 2.7 Answers to exercises 29

**3 Variables and scope** **33**
- 3.1 Variables 33
- 3.2 Modifying values 38
- 3.3 Type conversion 41
- 3.4 Answers to exercises 44

**4 Selection control statements** **47**
- 4.1 Introduction 47
- 4.2 Selection: `if` statement 47
- 4.3 More on the `if` statement 51
- 4.4 Boolean values, operators and expressions 53
- 4.5 The None value 58
- 4.6 Answers to exercises 60

**5 Collections** **63**
- 5.1 Lists 63
- 5.2 Tuples 67
- 5.3 Sets 68
- 5.4 Ranges 70
- 5.5 Dictionaries 70
- 5.6 Converting between collection types 72
- 5.7 Two-dimensional sequences 75
- 5.8 Answers to exercises 76

**6 Loop control statements** **79**
6.1 Introduction . . . 79
6.2 The `while` statement . . . 80
6.3 The `for` statement . . . 82
6.4 Nested loops . . . 83
6.5 Iterables, iterators and generators . . . 84
6.6 Comprehensions . . . 86
6.7 The `break` and `continue` statements . . . 88
6.8 Answers to exercises . . . 90

**7 Errors and exceptions** **95**
7.1 Errors . . . 95
7.2 Handling exceptions . . . 97
7.3 Debugging programs . . . 103
7.4 Logging . . . 105
7.5 Answers to exercises . . . 106

**8 Functions** **109**
8.1 Introduction . . . 109
8.2 Input parameters . . . 111
8.3 Return values . . . 112
8.4 The stack . . . 113
8.5 Default parameters . . . 114
8.6 `*args` and `**kwargs` . . . 117
8.7 Decorators . . . 118
8.8 Lambdas . . . 120
8.9 Generator functions and `yield` . . . 121
8.10 Answers to exercises . . . 122

**9 Classes** **127**
9.1 Defining and using a class . . . 127
9.2 Instance attributes . . . 129
9.3 Class attributes . . . 131
9.4 Class decorators . . . 133
9.5 Inspecting an object . . . 136
9.6 Overriding magic methods . . . 137
9.7 Answers to exercises . . . 139

**10 Object-oriented programming** **143**
10.1 Introduction . . . 143
10.2 Composition . . . 144
10.3 Inheritance . . . 147
10.4 More about inheritance . . . 149
10.5 Avoiding inheritance . . . 151
10.6 Answers to exercises . . . 152

**11 Packaging and testing** **159**
11.1 Modules . . . 159
11.2 Packages . . . 160
11.3 Documentation . . . 161
11.4 Testing . . . 163
11.5 Answers to exercises . . . 168

**12 Useful modules in the Standard Library** **173**
12.1 Date and time: `datetime` . . . 173

12.2 Mathematical functions: `math` . . . 174
12.3 Pseudo-random numbers: `random` . . . 174
12.4 Matching string patterns: `re` . . . 176
12.5 Parsing CSV files: `csv` . . . 180
12.6 Writing scripts: `sys` and `argparse` . . . 181
12.7 Answers to exercises . . . 183

**13 Introduction to GUI programming with `tkinter`** **187**
13.1 Event-driven programming . . . 187
13.2 `tkinter` basics . . . 187
13.3 Layout options . . . 189
13.4 Custom events . . . 190
13.5 Putting it all together . . . 191
13.6 Answers to exercises . . . 194

**14 Sorting, searching and algorithm analysis** **197**
14.1 Introduction . . . 197
14.2 Sorting algorithms . . . 197
14.3 Searching algorithms . . . 202
14.4 Algorithm complexity and Big O notation . . . 204
14.5 Answers to exercises . . . 206

**15 Indices and tables** **209**

Contents:

CHAPTER 1

# Introduction

The usefulness of computers is partly a result of their versatility in solving various problems and performing tasks. To be able to take advantage of the speed and power of computers, one needs to know how to program. This module is about computer programming and how it can be used to solve problems or perform useful tasks.

Our language of choice is Python – a recent language which has been found to be powerful, relatively easy to learn, and able to provide a platform to advanced programming. In this module you will learn how to analyse a problem and develop an effective solution for it using the Python programming language.

## What is a computer?

A computer is a general-purpose device which behaves according to the sets of instructions and data with which it is provided. Computers execute instructions to process data. Each computer has at its core a central processing unit (CPU) – modern CPUs are built as a single microprocessor chip.

### Computer instructions

A computer accepts a series of instructions as input, processes them one by one, and usually displays some kind of output to show what it has done. This is similar to the way people follow instructions or recipes. However, while people can understand complex instructions in natural languages (like English), computers can only understand very simple instructions which can be expressed in computer languages – restricted, formally specified languages which are much more limited than natural languages.

While some languages (like Python) can be used to express more high-level instructions than others (like assembly), there are still considerable limits. A computer can easily interpret an instruction like "add these two numbers together", but not a more complicated request like "balance my chequebook". Such a request would have to be broken down into many smaller step-by-step instructions which are simpler. Lists of instructions which tell the computer how to perform complex tasks are known as *programs*.

Here are some examples of simple computer instructions:

- *arithmetic*: adding, subtracting, multiplying or dividing numbers.
- *comparison*: comparing two numbers to see which is greater, or whether they are equal. These are often called *logical operations*.
- *branching*: jumping to another instruction in the program, and continuing from there.

Modern computers can execute many millions of these instructions in a second.

## Components of a computer

A computer contains four major types of components:

- *input*: anything that allows a computer to *receive* information from a user. This includes keyboards, mice, scanners and microphones.
- *processing*: the components of the computer which *process* information. The main processing component of a computer is the *central processing unit*, or CPU, but in a modern computer there are likely to be other processing units too. For example, many graphics cards come with *graphics processing units*, or GPUs, which were once only used to process graphics but today can also be used for general-purpose programs.
- *memory*: components where information is *stored*. This includes both *primary memory* (what we colloquially know as "memory") and *secondary memory* (what we know as *storage devices*, e.g. hard drives, CDs or flash disks).
- *output*: anything that the computer uses to *display* information to the user. This includes monitors, speakers and printers.

To understand how these components fit together, consider how a vending machine works. A vending machine is not, strictly speaking, a computer, but it can be broken down into very similar components:

- *input*: to use a vending machine, you put money into it and make a selection. The coin slots and selection buttons are the vending machine's input devices.
- *processing*: when you make your selection, the vending machine goes through several steps: verifying that it has received enough money, computing change, and making sure the selected item is in stock. The part of the machine that performs all these steps can be thought of as the processor.
- *output*: the vending machine deposits your purchase and your change in compartments from which you can retrieve them. It also has a simple electronic display which can show letters and numbers, and uses this to give you all kinds of feedback.
- *memory*: to perform the processing steps, the vending machine needs to keep track of information such as what items are in stock, their prices and the available change. This information must be stored somewhere.

## The CPU

The CPU is the most important part of a computer because of its instruction-processing functionality. The other parts of the computer follow the commands of the CPU. Two important characteristics of a CPU are:

- the *clock speed*: the CPU contains a clock which produces a regular signal. All the low-level operations (switches) that the CPU performs in order to process instructions are synchronised to this signal. The faster the clock, the faster the CPU can (in theory) operate – but there are other limitations. Today's typical clock speed is in excess of 1GHz or 1 000 000 000 switches per second.
- the *instruction set*: this is the set of instructions (more accurately, the machine language instructions) that the CPU understands. Different kinds of CPUs understand different sets of instructions: for example, Intel IA-32 and x86-64, IBM PowerPC or ARM.

A CPU has several important subcomponents:

- the *arithmetic/logic unit* (ALU) performs arithmetic and comparison operations.
- the *control unit* determines which instruction to execute next.
- *registers* form a high-speed storage area for temporary results.

## Memory

A computer stores information in its memory for later reference. There are two types of memory: primary and secondary.

*Primary memory* is connected directly to the CPU (or other processing units) and is usually referred to as RAM (random-access memory). Most primary memory loses its contents when the computer is switched off (i.e. it is *volatile*).

We can imagine primary memory as a long sequence of memory cells: each cell can be addressed by its memory address. These addresses start at zero for the first cell and each subsequent cell's address is one more than the one preceding it. Each cell can hold only a single number, but the CPU can replace the content with a new number at any time. The content can be retrieved without being erased from the cell.

*Secondary memory* is cheaper than primary memory, and can thus be made available in much larger sizes. Although it is much slower, it is non-volatile – that is, its contents are preserved even after the computer is switched off. Examples of this type of memory include hard disks and flash disks.

A computer's operating system provides high-level interfaces to secondary memory. These interfaces allow us to refer to clusters of related information called *files* which are arranged in a hierarchy of directories. Both the interfaces and the hierarchies are often referred to as *filesystems*.

We can think of directories as boxes (which may contain other boxes). Although it's easy to visualise the contents of a hard drive or flash disk using this metaphor, it is important to note that it is only a metaphor – at a lower level, a hard drive has a series of memory addresses just like RAM, and all the data is ultimately stored in this simple structure. Parts of the same file are not necessarily stored in adjacent memory addresses.

## Types of computers

Historically, computers have been categorised into specialised subtypes. The distinction is not always so clear-cut with modern computers, which can perform a greater variety of functions and often occupy multiple roles:

- *single-user personal computers*: these computers are designed for home use by a single person at a time. They are small enough to fit on a desk – something which was novel when they were first introduced. Modern personal computers are powerful enough to be used for many functions which were previously performed by more specialised computers.
- *batch computer systems*: most computers are *interactive* – when the user issues some kind of instruction, something happens in response right away. Some computers were designed to process large batches of instructions non-interactively – that is, large amounts of work was scheduled to be done without the possibility of further input from the user while it was being done. This allowed the computers to use their resources more efficiently.

  Some large companies may still use old-fashioned computer systems like this to perform highly repetitive tasks like payroll or accounting. Most interactive computer systems can be used to perform non-interactive batch operations. These operations are often scheduled during off-hours, when they are unlikely to compete with users for resources.
- *time-share computer systems*: these computer systems were an improvement over batch processing systems which allowed multiple users to access the same central computer remotely at the same time. The central computer was typically located in an air-conditioned room which was physically far away from the users. The users connected to the central computer through *terminals* which had little processing power of their own – they usually had only a mouse and a keyboard.

  Unlike a batch-processing computer, a time-share computer could switch between different users' program state, polling different terminals to check whether there was any new input from a particular user. As computer speeds improved, this switching happened so rapidly that it appeared that all the users' work was being performed simultaneously.

Today multiple users can connect to a central computer using an ordinary computer network. The role of the central computer can be played by an ordinary personal computer (although often one with much better hardware) which performs a specialised role. Most modern computers have the ability to switch between multiple running programs quickly enough that they appear to be running simultaneously. The role of the terminal is usually performed by the user's normal personal computer.

There are also powerful *supercomputers* whose specialised hardware allows them to exceed greatly the computing power of any personal computer. Users are given access to such computers when they need to solve a problem that requires the use of a lot of computing resources.

- *computer networks*: these are multiple computers connected to each other with digital or analog cables or wirelessly, which are able to communicate with each other. Today almost all computers can be connected to a network. In most networks there are specialised computers called *servers* which provide services to other computers on the network (which are called *clients*). For example, a storage server is likely to have many fast, high-capacity disk drives so that it can provide storage and back-up services to the whole network. A print server might be optimised for managing print jobs. Using servers keeps costs down by allowing users to share resources efficiently, while keeping the maintenance in one area.

  The Internet is a very large international computer network. Many computers on the Internet are servers. When you use a web browser, you send requests to web servers which respond by sending you webpages.

## History of computers

Today's computers are electronic. Earlier computers were mostly mechanical and electro-mechanical. Over time, computers have advanced exponentially – from expensive machines which took up entire rooms to today's affordable and compact units.

The oldest mechanical calculating aid is the abacus. It was invented in Babylon over 3000 years ago and was also used by the Chinese. It can be used for addition, subtraction, multiplication and division. More recently, in 1642, Blaise Pascal invented the first mechanical calculator. Pascal's machine was able to do addition and subtraction. In 1671, Gottfried von Leibnitz extended Pascal's machine to handle multiplication, division and square roots.

In 1801, Joseph-Marie Jacquard invented a loom which read a tape of punched cards. It was able to weave cloth according to instructions on the cards. This was the first machine that could be reprogrammed.

Towards the middle of the 1800s, Charles Babbage designed the Difference Engine, which was supposed to compute and print mathematical tables. However, it was never completed because he became engrossed in the design of his Analytical Engine. It was designed to follow instructions in a program and thus able to handle any computation. Babbage's machine was also going to make use of punched cards, but unfortunately the English government stopped funding the project and the machine was never completed. It is doubtful that the machine could have worked, since it required metalworking skills beyond what was possible at the time.

Ada Lovelace assisted Babbage in some of his work. In 1942, she translated one of Babbage's papers on the Analytical Engine from French to English and in the margins she wrote examples of how to use the machine – in effect becoming the first programmer ever.

American Herman Hollerith invented a method of using punched cards for automated data processing. His machines were employed by the US government to tabulate the 1890 census. Hollerith's firm later merged with three others to form International Business Machines (IBM). The punched card system remained in use well into the 1960s.

In 1944, Howard Aiken and his team completed the Mark I computer - the world's first automatic computer. It operated with electro-mechanical switches and was able to multiply two numbers in six seconds. In 1946, John W. Mauchly and J. Presper Eckert designed the first general-purpose electronic computer called ENIAC (E)lectronic (N)umerical (I)ntegrator (A)nd (C)omputer. It was hundreds of times faster than any electro-mechanical computing devices and could be programmed by the plugging of wires into holes along its outside.

Since the 1950s, computer systems have been available commercially. They have since become known by generation numbers.

### First-generation computers (1950s)

Marking the first generation of computers, Sperry-Rand introduced a commercial electronic computer, the UNIVAC I. Other companies soon followed suit, but these computers were bulky and unreliable by today's standards. For electronic switchings, they used vacuum tubes which generated a lot of heat and often burnt out. Most programs were written in machine language and made use of punched cards for data storage.

### Second-generation computers (late 50s to mid-60s)

In the late 1950s, second generation computers were produced using transistors instead of vacuum tubes, which made them more reliable. Higher-level programming languages like FORTRAN, Algol and COBOL were developed at about this time, and many programmers began to use them instead of assembly or machine languages. This made programs more independent of specific computer systems. Manufacturers also provided larger amounts of primary memory and also introduced magnetic tapes for long-term data storage.

### Third-generation computers (mid-60s to early 70s)

In 1964, IBM introduced its System/360 line of computers – with every machine in the line able to run the same programs, but at different speeds. This generation of computers started to employ integrated circuits containing many transistors. Most ran in batch mode, with a few running in time-share mode.

### Fourth-generation computers (early 70s and onwards)

From the early 1970s, computer designers have been concentrating on making smaller and smaller computer parts. Today, computers are assembled using very large-scale integration (VLSI) integrated circuits, each containing millions of transistors on single chips. This process has resulted in cheaper and more reliable computers. In the late 1960s and early 1970s, medium-sized organisations including colleges and universities began to use computers. Small businesses followed suit by the early 1980s. In this period, most were time-shared systems. Today's computers are usually single-user or multiple-user personal computers, with many connected to larger networks such as the Internet.

## Programming a computer

### Algorithms

An algorithm is a series of steps which must be followed in order for some task to be completed or for some problem to be solved. You have probably used an algorithm before – for example, you may have assembled a model toy by following instructions or cooked using a recipe. The steps in a set of instructions or a recipe form a kind of algorithm. They tell you how to transform components or ingredients into toys or cookies. The ingredients act as an input to the algorithm. The algorithm transforms the ingredients into the final output.

#### Recipes as algorithms (the structured approach to programming)

In a typical recipe, there is a list of ingredients and a list of steps. The recipes do not usually say who is performing the steps, but implies that you (the cook) should perform them. Many algorithms are written in this way, implying that the CPU of the computer is the *actor* of these instructions. This approach is referred to as the structured approach to programming and it works reasonably well for simple programs. However, it can become more complex when you are trying to solve a real world problem where it is rare for one actor to be in control of everything. The object-oriented approach to programming is an attempt to simulate the real world by including several actors in the algorithm.

### Play scripts as algorithms (the object-oriented approach to programming)

A script of a play is a good analogy to the object-oriented (OO) approach. Actors and scenes are listed at the beginning (like the ingredients of a recipe). In a scene, the script lists the instructions for each actor to speak and act. Each actor is free to interpret these instructions (subject to the director's approval) in a way he or she sees appropriate. In the OO programming approach, multiple objects act together to accomplish a goal. The algorithm, like a play script, directs each object to perform particular steps in the correct order.

## Programming languages

To make use of an algorithm in a computer, we must first convert it to a program. We do this by using a programming language (a very formal language with strict rules about spelling and grammar) which the computer is able to convert unambiguously into computer instructions, or *machine language*. In this course we will be using the Python programming language, but there are many other languages which we will outline later in the chapter.

The reason that we do not write computer instructions directly is that they are difficult for humans to read and understand. For example, these are the computer instructions (in the Intel 8086 machine language, a subset of the Intel Pentium machine language) required to add 17 and 20:

```
1011 0000 0001 0001
0000 0100 0001 0100
1010 0010 0100 1000 0000 0000
```

The first line tells the computer to copy 17 into the AL register: the first four characters (1011) tell the computer to copy information into a register, the next four characters (0000) tell the computer to use register named AL, and the last eight digits (0001 0001, which is 17 in binary) specify the number to be copied.

As you can see, it is quite hard to write a program in machine language. In the 1940s, the programmers of the first computers had to do this because there were no other options! To simplify the programming process, *assembly language* was introduced.

Each assembly instruction corresponds to one machine language instruction, but it is more easily understood by humans, as can be seen in the equivalent addition program in the 8086 assembly language:

```
MOV AL, 17D
ADD AL, 20D
MOV [SUM], AL
```

Programs written in assembly language cannot be understood by the computer directly, so a translation step is needed. This is done using an assembler, whose job it is to translate from assembly language to machine language.

Although assembly language was a great improvement over machine language, it can still be quite cryptic, and it is so low-level that the simplest task requires many instructions. *High-level languages* were developed to make programming even easier.

In a high-level language, an instruction may correspond to several machine language instructions, which makes programs easier to read and write. This is the Python equivalent of the code above:

```
sum = 17 + 20
```

## Compilers, interpreters and the Python programming language

Programs written in high-level languages must also be translated into machine language before a computer can execute them. Some programming languages translate the whole program at once and store the result in another file which is then executed. Some languages translate and execute programs line-by-line. We call these languages *compiled* languages and *interpreted* languages, respectively. Python is an interpreted language.

A compiled language comes with a *compiler*, which is a program which *compiles* source files to executable binary files. An interpreted language comes with an *interpreter*, which *interprets* source files and executes them. Interpretation can be less efficient than compilation, so interpreted languages have a reputation for being slow.

Programs which need to use a lot of computer resources, and which therefore need to be as efficient as possible, are often written in a language like C. C is a compiled language which is in many ways lower-level than Python – for example, a C programmer needs to handle a lot of memory management explicitly; something a Python programmer seldom needs to worry about.

This fine-grained control allows for a lot of optimisation, but can be time-consuming and error-prone. For applications which are not resource-intensive, a language like Python allows programs to be developed more easily and rapidly, and the speed difference on a modern computer is usually negligible.

## Developing a Python program

Suppose that we want to develop a program to display an average of a series of numbers. The first step is to understand the problem, but first we need to know more about the program:

- How will it get the numbers? Let's assume that the user will type them in using the keyboard.
- How will it know how many numbers to get? Let's assume that the user will enter 0 to signify the end of the list.
- Where should it display results? It should print them on the screen.

The second step is to come up with the algorithm. You can do this by trying to solve the problem yourself and noting the steps that you took. Try this. You are likely to end up with an instruction list which looks something like this:

1. Set running total to 0.
2. Set running count to 0.
3. Repeat these steps:
    - Read a value.
    - Check if value is 0 (stop this loop if so).
    - Add value to running total.
    - Add 1 to running count.
4. Compute the average by dividing the running total by the count.
5. Display the average.

The next step of development is to test the algorithm. You can do this by trying the algorithm on different lists of numbers and see if you can get a correct result from it. For simple algorithms such as this, you can do the test on paper or using a calculator. Try this with a simple list of numbers, like 1, 2, 3, 4 and 5. Then try a more complex list. You might get a feeling that this algorithm works for all cases. However, there is a case that the converted program might not be able to handle. Try a list which only contains 0. The average of a list with a single 0 in it is 0, but what does the algorithm tell you the answer is? Can you modify the algorithm to take care of this?

We now look at the four steps in developing a program in more detail.

### Understanding the problem

Before you start writing a program, you should thoroughly understand the problem that you are trying to solve:

- When you start, think about the problem on your own without touching the keyboard. Figure out exactly what you need the algorithm to do.
- If the program is for someone else, it might be helpful to speak to those who will be using the program, to find out exactly what is needed. There might be missing information or constraints such as speed, robustness or ease of use.
- Think about what type of input and output the program will need to have, and how it will be entered and displayed.
- It can be useful to plan the program on paper by writing down lists of requirements or sketching a few diagrams. This can help you to gather your thoughts and spot any potential error or missing information.

### Coming up with an algorithm

We have previously described an algorithm as a series of steps, but there are a few more formal requirements:

- It must be a *finite* series of steps: a never-ending list of steps is not an algorithm. A computer has finite resources (like memory) and cannot handle infinite lists of steps.
- It must be *unambiguous*: each step must be an unambiguous instruction. For example, you cannot have a step which says "multiply x by either 1 or -1". If you have an instruction like this, you have to specify exactly how the choice should be made – for example, "if y is less than 0, multiply x by 1, otherwise multiply x by -1".
- It must be *effective*: The algorithm must do what it is supposed to do.
- It must *terminate*: The algorithm must not go on forever. Note that this is different to the requirement that it be finite. Consider the following finite list of instructions:

  1. Set x to 1
  2. Set y to 2
  3. Repeat the following steps: #. Add x and y to get z #. Display z on screen
  4. Display the word 'Done'

  If you try to follow these instructions, you will get stuck on step 3 forever – therefore, this list is not an algorithm.

### Writing the program

Once you have an algorithm, you can translate it into Python. Some parts of the algorithm may be straightforward to translate, but others may need to be expressed slightly differently in Python than they would be in a natural language. Sometimes you can use the language's features to rewrite a program more cleanly or efficiently. This will become easier the more experience you gain with Python.

You should take care to avoid errors by thinking through your program as you write it, section by section. This is called *desk checking*. In later chapters we will also discuss other tools for checking your code, for example programs which automatically check the code for certain kinds of errors.

### Testing the program

After thoroughly desk checking the program as you write, you should run it and test it with several different input values. Later in this module we will see how to write automated tests – programs that you write to test your programs. Automated tests make it easy to run the same tests repeatedly to make sure that your program still works.

## Programming languages

There are hundreds of different programming languages. Some are particularly suited to certain tasks, while others are considered to be general-purpose. Programs can be categorised into four major groups – *procedural*, *functional*, *logic* and *object-oriented* languages – but some languages contain features characteristic of more than one group.

### Procedural languages

Procedural (also known as imperative) languages form the largest group of languages. A program written in a procedural language consists of a list of statements, which the computer follows in order. Different parts of the program communicate with one another using variables. A variable is actually a named location in primary memory. The value stored in a variable can usually be changed throughout the program's execution. In some other programming language paradigms (such as logic languages), variables act more like variables used in mathematics and their values may not be changed.

BASIC is an example of a procedural programming language. It has a very simple syntax (originally only 14 statement types). Here is some BASIC code for adding 17 and 20:

```
10 REM THIS BASIC PROGRAM CALCULATES THE SUM OF
20 REM 17 AND 20, THEN DISPLAYS THE RESULT.
30 LET RESULT = 17 + 20
40 PRINT "The sum of 17 and 20 is ", RESULT
50 END
```

In the early 1990s, Microsoft's Visual Basic extended the BASIC language to a development system for Microsoft Windows.

COBOL (COmmon Business Oriented Language) has commonly been used in business. It was designed for ease of data movement. Here's the addition program written in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.      ADDING.
DATE-WRITTEN.    JUL 11,1997.
DATE-COMPILED.   JUL 12,1997.
*    THIS COBOL PROGRAM CALCULATE THE SUM
*    OF 17 AND 20. IT THEN DISPLAYS THE RESULT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-370.
OBJECT-COMPUTER. IBM-370.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  TOTAL           PICTURE 99.
PROCEDURE DIVISION.
  ADD 17, 20 GIVING TOTAL.
  DISPLAY 'THE SUM OF 17 AND 20 IS ', TOTAL UPON CONSOLE.
  STOP RUN.
END PROGRAM
```

FORTRAN (FORmula TRANslator) was popular with scientists and engineers, but many have switched to C and C++. It was the first high-level language. Here's the addition program written in FORTRAN:

```
      PROGRAM ADDNUMS
C     THIS FORTRAN PROGRAM FINDS THE TOTAL OF
C     17 AND 20, THEN DISPLAYS THE RESULT.

      INTEGER RESULT
      RESULT = 17 + 20
      WRITE (*, 10) RESULT
   10 FORMAT ('THE SUM OF 17 AND 20 IS ', I2)
      STOP
      END
```

Many programmers use C to write system-level code (operating systems, compilers). Here's the addition program code written in C:

```
/* This C program calculates the sum of 17 and 20.
It then displays the result. */

#include <stdio.h>

void main(void)
{
    int result;
    result = 17 + 20;
    printf("The sum of 17 and 20 is %d\n", result);
}
```

Pascal (named after Blaise Pascal) used to be a popular introductory programming language until the early 1990s, but many schools have switched to C++ or Java. Here is the addition program written in Pascal:

```
PROGRAM
AddNums (Input, Output);
{ This Pascal program calculate the sum of 17 and 20. It then displays the result }

BEGIN
   Result := 17 + 20;
   writeLn ('The sum of 17 and 20 is ', Result)
END
```

## Functional and logic languages

A functional language is based on mathematical functions. It usually consists of functions and function calls. In the language's pure form, variables do not exist: instead, parts of program communicate through the use of function parameters. One of the most popular functional languages is Common Lisp, which is widely used in the artificial intelligence field, especially in the US. Other functional languages include ML and Miranda. Here's the addition program written using a functional style, in Common Lisp:

```
;; This Lisp program calculates the
;; sum of 20 and 17. It then displays the result.
(format t "The sum of 20 and 17 is ~D~%" (+ 20 17))
```

A logic language is based on formal rules of logic and inference. An example of such a language is Prolog. Prolog's variables cannot be changed once they are set, which is typical of a logic language. Here's the addition program written in Prolog:

```
/* This Prolog program calculates the sum of 17 and 20. It then
/* displays the result. */

run :- Result is 17 + 20,
    write("The sum of 17 and 20 is ", Result),
    nl.
```

The above program does not show the deductive power of Prolog. Prolog programs can consist of a set of known facts, plus rules for inferring new facts from existing ones. In the example below, the first seven lines list facts about the kind of drink certain people like. The last line is a rule of inference which says that Matt likes a drink only when both Mike and Mary like that drink:

```
/* Shows Prolog's inference ability */
likes(dave, cola).
likes(dave, orange_juice).
likes(mary, lemonade).
likes(mary, orange_juice).
likes(mike, sparkling_water).
likes(mike, orange_juice).
likes(matt, Drink) :- likes(mary, Drink), likes(mike, Drink).
```

A Prolog program answers a query. For example, we might want to know what food Mary likes – we can query this:

```
likes(mary, Drink).
```

To which Prolog will output possible answers, like:

```
Drink = lemonade
Drink = orange_juice
```

To demonstrate the rule of inference, we can ask for the food that Matt likes. The system can find the solution by checking what Mary and Mike like:

```
likes(matt, Drink)

Drink = orange_juice
```

## Object-oriented languages

More recently, computer scientists have embraced a new programming approach – object-oriented (OO) programming. In this approach, programmers model each real-world entity as an object, with each object having its own set of values and behaviours. This makes an object an active entity, whereas a variable in a procedural language is passive.

Simula (Simulation Language), invented in 1967, was the first language to take the object-oriented approach. However, Smalltalk (early 1980s) was the first purely object-oriented language – everything in the language is an object. C++ is a hybrid OO language in that it has the procedural aspects of C. A C++ program can be completely procedural, completely OO or a hybrid. Most OO languages make use of variables in a similar fashion to procedural languages.

Here's the addition code written in C++; note the similarity to the earlier program written in C:

```
// This C++ program finds the result of adding 17 and 20.
// It then displays the result.
#include <iostream>

int main(void)
{
    int result = 17 + 20;
    std::cout << "The sum of 17 and 20 is " << result << std::endl;

    return 0;
}
```

Java was introduced in 1995 by Sun Microsystems, who were purchased by Oracle Corporation during 2009-2010. It is an OO language but not as pure as Smalltalk. For example, in Java primitive values (numbers and characters) are not objects – they are values. In Smalltalk, everything is an object. Initially it was designed for use in appliances, but the first released version was designed for use on the Internet. Java syntax is quite similar to C and C++, but functions cannot be defined outside of objects. Here is the addition code written in Java:

```
// This is a Java program to calculate the sum of
// 17 and 20. It then displays the result.
public class Add {
    public static void main(String[] args) {
        int result;
        result = 17 + 20;
        System.out.println("sum of 17 && 20 is " + result);
    }
}
```

Python is a general-purpose interpreted language which was originally created in the late 1980s, but only became widely used in the 2000s after the release of version 2.0. It is known for its clear, simple syntax and its dynamic typing – the same variables in Python can be reused to store values of different types; something which would not be allowed in a statically-typed language like C or Java. Everything in Python is an object, but Python can be used to write code in multiple styles – procedural, object-oriented or even functional. Here is the addition code written in Python:

```
# This Python program adds 17 and 20 and prints the result.
result = 17 + 20
print("The sum of 17 and 20 is %d." % result)
```

CHAPTER 2

# Python basics

## Introduction

In this chapter, we introduce the basics of the Python programming language. At this point you should already have set up a development environment for writing and running your Python code. It will be assumed in the text that this is the case. If you are having trouble setting up Python, contact a teaching assistant or post a message to the course forum. Throughout this course, you are strongly encouraged to try what you have learnt by writing an actual program. You can only learn how to program by actually doing it yourself.

Each chapter contains several exercises to help you practice. Solutions are found at the end of the chapter.

### Python 2 vs Python 3

Python recently underwent a major version change from 2 to 3. For consistency with other courses in the department, we will be using Python 3. Python 2 is still widely used, and although Python 3 is not fully backwards compatible the two versions are very similar – so should you ever encounter Python 2 code you should find it quite familiar. We will mention important differences between the two versions throughout this course. We will always refer to the *latest* version of Python 2, which at the time of writing was *2.7*.

## Getting started with Python

### Using the interactive interpreter

Entering `python` on the commandline without any parameters will launch the Python interpreter. This is a text console in which you can enter Python commands one by one – they will be interpreted on the fly.

**Note:** In these notes we will assume throughout that the `python` command launches Python 3, but if you have both Python 2 and Python 3 installed on your computer, you may need to specify that you want to use Python 3 by using the `python3` command instead. Whenever you launch the interpreter, you will see some information printed before the prompt, which includes the version number – make sure that it starts with 3! Take note of the command that you need to use.

Here is an example of an interpreter prompt:

```
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you type a number, string or any variable into the interpreter, its value will automatically be echoed to the console:

```
>>> "hello"
'hello'
>>> 3
3
```

That means that you don't have to use an explicit print command to display the value of a variable if you are using the interpreter – you can just enter the bare variable, like this:

```
>>> x = 2
>>> x
2
```

This won't work if you are running a program from a file – if you were to enter the two lines above into a file and run it, you wouldn't see any output at all. You would have to use the print function to output the value of `x`:

```
x = 2
print(x)
```

In most of the code examples in this module we have used explicit print statements, so that you will see the same output whether you use the examples in the interpreter or run them from files.

The interpreter can be very useful when you want to test out a small piece of code before adding it to a larger program. It's a quick and easy way to check how a function works or make sure that the syntax of a code fragment is correct.

There are some other interactive interpreters for Python which have more advanced features than the built-in interpreter, for example functionality for inspecting the contents of objects or querying the documentation for imported modules, classes and functions:

- IPython, which was originally developed within the scientific community
- bpython, a new project

## Running programs from files

The interpreter is useful for testing code snippets and exploring functions and modules, but to save a program permanently we need to write it into a file. Python files are commonly given the suffix `.py`. Once you have written a program and saved it, you can run it by using the `python` command with the file name as a parameter:

```
python myprogram.py
```

This will cause Python to execute the program.

Like any source code file, a Python file is just an ordinary text file. You can edit it with any text editor you like. It is a good idea to use a text editor which at least supports syntax highlighting – that is, it can display the words in your program in different colours, depending on the function they perform in your program. It is also useful to have indentation features such as the ability to indent or unindent blocks of code all at once, and automatic indentation (having the program guess the right level of indentation whenever you start typing a new line).

Some programmers prefer to use an *integrated development environment*, or IDE. An IDE is a program which combines a text editor with additional functionality like looking up documentation, inspecting objects, compiling the code (in the case of a compiled language) and running the code. Some IDEs support multiple languages, and some are designed for a specific language.

There are many IDEs, free and commercial, which can be used with Python. Python also comes with a simple built-in IDE called IDLE (you may need to install it from a separate package).

### Installing new packages

How you install new Python packages depends a little on your operating system. Linux distributions have their own package managers, and you may choose to install packages using these managers so that they are integrated with the other packages on your system. However, some obscure Python packages may not be available as system packages, and the packages which are available are often not the latest versions. It is thus sometimes necessary to install packages directly from PyPI.

The Python Package Index (PyPI) is a large repository of Python packages. You can install packages from this repository using a tool like easy_install or pip (which is intended to be a more modern replacement for easy_install). Both of these utilities are cross-platform. Here is how you install a package called `sqlobject` with pip:

```
pip install sqlobject
```

This command will search PyPI for a package called sqlobject, download it and install it on your system.

### Further reading

In this module we will see many examples of Python's built-in functions and types and modules in the standard library – but this document is only a summary, and not an exhaustive list of all the features of the language. As you work on the exercises in this module, you should use the official Python documentation as a reference.

For example, each module in the standard library has a section in the documentation which describes its *application programming interface*, or API – the functionality which is available to you when you use the module in your code. By looking up the API you will be able to see what functions the module provides, what input they require, what output they return, and so on. The documentation often includes helpful examples which show you how the module is meant to be used.

The documentation is available on the web, but you can also install it on your computer – you can either download a copy of the documentation files in HTML format so that you can browse them locally, or use a tool like `pydoc`, which prints out the documentation on the commandline:

```
pydoc re
```

## Essentials of a Python program

In most of today's written languages, words by themselves do not make sense unless they are in certain order and surrounded by correct punctuation symbols. This is also the case with the Python programming language. The Python interpreter is able to interpret and run correctly structured Python programs. For example, the following Python code is correctly structured and will run:

```python
print("Hello, world!")
```

Many other languages require a lot more structure in their simplest programs, but in Python this single line, which prints a short message, is sufficient. It is not, however, a very informative example of Python's syntax – so here is a slightly more complex program which does (almost) exactly the same thing:

```python
# Here is the main function.
def my_function():
    print("Hello, World!")
```

```
my_function()
```

This type of program is often referred to as a skeleton program, because one can build upon it to create a more complex program.

**Note:** The first line of the skeleton program is a comment. A hash (`#`) denotes the start of a comment. The interpreter will ignore everything that follows the hash until the end of the line. Comments will be discussed further in the later part of this unit.

## Keywords

In the code above, the words `def` and `if` are keywords or reserved words, i.e. they have been kept for specific purposes and may not be used for any other purposes in the program. The following are keywords in Python:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

## Identifier names

When we write a Python program, we will create many entities – variables which store values like numbers or strings, as well as functions and classes. These entities must given names by which they can be referred to uniquely – these names are known as identifiers. For example, in our skeleton code above, `my_function` is the name of the function. This particular name has no special significance – we could also have called the function `main` or `print_hello_world`. What is important is that we use the same name to refer to the function when we call it at the bottom of the program.

Python has some rules that you must follow when forming an identifier:

- it may only contain letters (uppercase or lowercase), numbers or the underscore character (_) (no spaces!).
- it may not start with a number.
- it may not be a keyword.

If we break any of these rules, our program will exit with a syntax error. However, not all identifiers which are syntactically correct are meaningful to human readers. There are a few guidelines that we should follow when naming our variables to make our code easier to understand (by other people, and by us!) – this is an important part of following a good coding style:

- be descriptive – a variable name should describe the contents of the variable; a function name should indicate what the function does; etc..
- don't use abbreviations unnecessarily – they may be ambiguous and more difficult to read.

Pick a naming convention, and stick to it. This is a commonly used naming convention in Python:

- names of classes should be in CamelCase (words capitalised and squashed together).
- names of variables which are intended to be constants should be in CAPITAL_LETTERS_WITH_UNDERSCORES.

- names of all other variables should be in lowercase_with_underscores. In some other languages, like Java, the standard is to use camelCase (with the initial letter lowercase), but this style is less popular in Python.
- names of class attributes and methods which are intended to be "private" and not accessed from outside the class should start with an underscore.

Of course there are always exceptions – for example, many common mathematical symbols have very short names which are nonetheless widely understood.

Here are a few examples of identifiers:

| Syntax error | Bad practice | Good practice |
|---|---|---|
| Person Record | PRcrd | PersonRecord |
| DEFAULT-HEIGHT | Default_Ht | DEFAULT_HEIGHT |
| class | Class | AlgebraCourse |
| 2totalweight | num2 | total_weight |

**Note:** Be careful not to redefine existing variables accidentally by reusing their names. This applies not only to your own variables, but to built-in Python functions like `len`, `max` or `sort`: these names are not keywords, and you will not get a syntax error if you reuse them, but you will encounter confusing results if you try to use the original functions later in your program. Redefining variables (accidentally and on purpose) will be discussed in greater detail in the section about scope.

### Exercise 1

Write down why each of the entries in the left column will raise a syntax error if it is used as an identifier.

### Flow of control

In Python, statements are written as a list, in the way that a person would write a list of things to do. The computer starts off by following the first instruction, then the next, in the order that they appear in the program. It only stops executing the program after the last instruction is completed. We refer to the order in which the computer executes instructions as the flow of control. When the computer is executing a particular instruction, we can say that control is at that instruction.

### Indentation and (lack of) semicolons

Many languages arrange code into blocks using curly braces (`{` and `}`) or `BEGIN` and `END` statements – these languages encourage us to indent blocks to make code easier to read, but indentation is not compulsory. Python uses indentation only to delimit blocks, so we *must* indent our code:

```python
# this function definition starts a new block
def add_numbers(a, b):
    # this instruction is inside the block, because it's indented
    c = a + b
    # so is this one
    return c

# this if statement starts a new block
if it_is_tuesday:
    # this is inside the block
    print("It's Tuesday!")
# this is outside the block!
print("Print this no matter what.")
```

In many languages we need to use a special character to mark the end of each instruction – usually a semicolon. Python uses ends of lines to determine where instructions end (except in some special cases when the last symbol on the line lets Python know that the instruction will span multiple lines). We may optionally use semicolons – this is something we might want to do if we want to put more than one instruction on a line (but that is usually bad style):

```
# These all individual instructions -- no semicolons required!
print("Hello!")
print("Here's a new instruction")
a = 2

# This instruction spans more than one line
b = [1, 2, 3,
    4, 5, 6]

# This is legal, but we shouldn't do it
c = 1; d = 5
```

## Exercise 2

Write down the two statements inside the block created by the `append_chickens` function:

```
no_chickens = "No chickens here ..."

def append_chickens(text):
    text = text + " Rawwwk!"
    return text

print(append_chickens(no_chickens))
```

## Exercise 3

The following Python program is not indented correctly. Re-write it so that it is correctly indented:

```
def happy_day(day):
if day == "monday":
return ":("
if day != "monday":
return ":D"

print(happy_day("sunday"))
print(happy_day("monday"))
```

## Letter case

Unlike some languages (such as Pascal), Python is case-sensitive. This means that the interpreter treats upper- and lowercase letters as different from one another. For example, `A` is different from `a` and `def main()` is different from `DEF MAIN()`. Also remember that all reserved words (except `True`, `False` and `None`) are in lowercase.

## More on Comments

Recall that comments start with `#` and continue until the end of the line, for example:

```
# This is a comment
print("Hello!")    # tells the computer to print "Hello!"
```

Comments are ignored by the interpreter and should be used by a programmer to:

- describe what the program does
- describe (in higher-level terms than the code) how the program works

It is not necessary to comment each line. We should comment in appropriate places where it might not be clear what is going on. We can also put a short comment describing what is taking place in the next few instructions following the comment.

Some languages also have support for comments that span multiple lines, but Python does not. If we want to type a very long comment in Python, we need to split it into multiple shorter lines and put a `#` at the start of each line.

**Note:** It is possible to insert a multi-line string literal into our code by enclosing it in triple quotes. This is not normally used for comments, except in the special case of docstrings: strings which are inserted at the top of structures like functions and classes, and which document them according to a standard format. It is good practice to annotate our code in this way because automated tools can then parse it to generate documentation automatically. We will discuss docstrings further in a future chapter.

**Note:** You can easily disable part of your program temporarily by commenting out some lines. Adding or removing many hashes by hand can be time-consuming – your editor should have a keyboard shortcut which allows you to comment or uncomment all the text you have selected.

## Reading and writing

Many programs display text on the screen either to give some information or to ask for some information. For example, we might just want to tell the user what our program does:

```
Welcome to John's Calculating Machine.
```

Perhaps we might want to ask the user for a number:

```
Enter the first number:
```

The easiest way to output information is to display a string literal using the built-in `print` function. A string literal is text enclosed in quotes. We can use either single quotes (`'`) or double quotes (`"`) – but the start quote and the end quote have to match!

These are examples of string literals:

```
"Welcome to John's Calculating Machine."
'Enter the first number:'
```

We can tell the computer to print "Hello!" on the console with the following instruction:

```
print("Hello!")
```

As you can see the `print` function takes in a string as an argument. It prints the string, and also prints a newline character at the end – this is why the console's cursor appears on a new line after we have printed something.

To query the user for information, we use the `input` function:

```
first_number = input('Enter the first number: ')
```

There are several things to note. First, unlike the `print` function, the `input` function does *not* print a newline automatically – the text will be entered directly after the prompt. That is why we have added a trailing space after the colon. Second, the function always returns a string – we will have to convert it to a number ourselves.

The string prompt is optional – we could just use the `input` function without a parameter:

```
second_number = input()
```

**Note:** in Python 2, there is a function called `raw_input` which does what `input` does in Python 3: that is, it reads input from the user, and returns it as a string. In Python 2, the function called `input` does something different: it reads input from the user and tries to evaluate it as a Python expression. There is no function like this in Python 3, but you can achieve the same result by using the `eval` function on the string returned by `input`. `eval` is almost always a bad idea, and you should avoid using it – especially on arbitrary user input that you haven't checked first. It can be very dangerous – the user could enter absolutely anything, including malicious code!

## Files

Although the `print` function prints to the console by default, we can also use it to write to a file. Here is a simple example:

```
with open('myfile.txt', 'w') as myfile:
    print("Hello!", file=myfile)
```

Quite a lot is happening in these two lines. In the `with` statement (which we will look at in more detail in the chapter on errors and exceptions) the file `myfile.txt` is opened for writing and assigned to the variable `myfile`. Inside the `with` block, `Hello!` followed by a newline is written to the file. The `w` character passed to `open` indicates that the file should be opened for writing.

As an alternative to `print`, we can use a file's `write` method as follows:

```
with open('myfile.txt', 'w') as myfile:
    myfile.write("Hello!")
```

A method is a function attached to an object – methods will be explained in more detail in the chapter about classes.

Unlike `print`, the `write` method does not add a newline to the string which is written.

We can read data from a file by opening it for reading and using the file's `read` method:

```
with open('myfile.txt', 'r') as myfile:
    data = myfile.read()
```

This reads the contents of the file into the variable `data`. Note that this time we have passed `r` to the `open` function. This indicates that the file should be opened for reading.

**Note:** Python will raise an error if you attempt to open a file that has not been created yet for reading. Opening a file for writing will create the file if it does not exist yet.

**Note:** The `with` statement automatically closes the file at the end of the block, even if an error occurs inside the block. In older versions of Python files had to be closed explicitly – this is no longer recommended. You should always use the `with` statement.

### Built-in types

There are many kinds of information that a computer can process, like numbers and characters. In Python (and other programming languages), the kinds of information the language is able to handle are known as types. Many common types are built into Python – for example integers, floating-point numbers and strings. Users can also define their own types using classes.

In many languages a distinction is made between built-in types (which are often called "primitive types" for this reason) and classes, but in Python they are indistinguishable. Everything in Python is an object (i.e. an instance of some class) – that even includes lists and functions.

A type consists of two parts: a domain of possible values and a set of possible operations that can be performed on these values. For example, the domain of the integer type (`int`) contains all integers, while common integer operations are addition, subtraction, multiplication and division.

Python is a dynamically (and not statically) typed language. That means that we don't have to specify a type for a variable when we create it – we can use the same variable to store values of different types. However, Python is also strongly (and not weakly) typed – at any given time, a variable has a definite type. If we try to perform operations on variables which have incompatible types (for example, if we try to add a number to a string), Python will exit with a type error instead of trying to guess what we mean.

The function `type` can be used to determine the type of an object. For example:

```
print(type(1))
print(type("a"))
```

## Integers

An integer (`int` type) is a whole number such as `1`, `5`, `1350` or `-34`. `1.5` is not an integer because it has a decimal point. Numbers with decimal points are floating-point numbers. Even `1.0` is a floating-point number and not an integer.

### Integer operations

Python can display an integer with the `print` function, but only if it is the only argument:

```
print(3)
# We can add two numbers together
print(1 + 2)
```

We can't combine a string and an integer directly, because Python is strongly typed:

```
>>> print("My number is " + 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

If we want to print a number and a string together, we will have to convert the number to a string somehow:

```
# str function converts things to strings.
# Then we can concatenate two strings with +.
print("My number is " + str(3))

# String formatting does the conversion for us.
print("My number is %d" % 3)
```

Other integer operations:

| Operation | Symbol | Example | Result |
|---|---|---|---|
| Addition | `+` | `28 + 10` | `38` |
| Subtraction | `-` | `28 - 10` | `18` |
| Multiplication | `*` | `28 * 10` | `280` |
| Division | `//` | `28 // 10` | `2` |
| Modulus (remainder) | `%` | `28 % 10` | `8` |
| Exponent (power) | `**` | `28**10` | `296196766695424` |

Note that all these operations are integer operations. That is why the answer to `28 // 10` is not `2.8`, but `2`. An integer operation results in an integer solution.

**Note:** In Python 2, the operator `/` performed integer division if both the dividend and the divisor were integers, and floating-point division if at least one of them was a float. In Python 3, `/` *always* performs floating-point division and `//` *always* performs integer division – even if the dividend and divisor are floats!

**Note:** Some other languages (e.g. C, Java) store each integer in a small fixed amount of memory. This limits the size of the integer that may be stored. Common limits are `2**8`, `2**16`, `2**32` and `2**64`. Python has no fixed limit can stored surprisingly large integers such as `2**1000000` as long as there is enough memory and processing power available on the machine where it is running.

## Operator precedence

Another important thing to keep in mind is operator precedence. For example, does `1 + 2 // 3` mean `(1 + 2) // 3` or `1 + (2 // 3)`? Python has a specific and predictable way to determine the order in which it performs operations. For integer operations, the system will first handle brackets `()`, then `**`, then `*`, `//` and `%`, and finally `+` and `-`.

If an expression contains multiple operations which are at the same level of precedence, like `*`, `//` and `%`, they will be performed in order, either from left to right (for left-associative operators) or from right to left (for right-associative operators). All these arithmetic operators are left-associative, except for `**`, which is right-associative:

```
# all arithmetic operators other than ** are left-associative, so
2 * 3 / 4
# is evaluated left to right:
(2 * 3) / 4

# ** is right-associative, so
2 ** 3 ** 4
# is evaluated right to left:
2 ** (3 ** 4)
```

The following table shows some more examples of precedence:

| Expression | How Python evaluates | Result |
|---|---|---|
| 20 + 10 // 2 | 20 + (10 // 2) | 25 |
| 20 + 10 - 2 | (20 + 10) - 2 | 28 |
| 20 - 10 + 2 | (20 - 10) + 2 | 12 |
| 20 - 10 * 2 | 20 - (10 * 2) | 0 |
| 20 // 10 * 2 | (20 // 10) * 2 | 4 |
| 20 * 10 // 2 | (20 * 10) // 2 | 100 |
| 20 * 10 ** 2 | 20 * (10 ** 2) | 2000 |

Sometimes it's a good idea to add brackets to arithmetic expressions even if they're not compulsory, because it makes the code more understandable.

### Exercise 4

1. Which of the following numbers are valid Python integers? `110`, `1.0`, `17.5`, `-39`, `-2.3`
2. What are the results of the following operations and explain why: #. `15 + 20 * 3` #. `13 // 2 + 3` #. `31 + 10 // 3` #. `20 % 7 // 3` #. `2 ** 3 ** 2`
3. What happens when you evaluate `1 // 0` in the Python console? Why does this happen?

## Floating-point numbers

Floating-point numbers (`float` type) are numbers with a decimal point or an exponent (or both). Examples are `5.0`, `10.24`, `0.0`, `12.` and `.3`. We can use scientific notation to denote very large or very small floating-point numbers, e.g. 3.8 x $10^{15}$. The first part of the number, 3.8, is the mantissa and 15 is the exponent. We can think of the exponent as the number of times we have to move the decimal point to the right to get to the actual value of the number.

In Python, we can write the number 3.8 x $10^{15}$ as `3.8e15` or `3.8e+15`. We can also write it as `38e14` or `.038e17`. They are all the same value. A negative exponent indicates smaller numbers, e.g. `2.5e-3` is the same as `0.0025`. Negative exponents can be thought of as how many times we have to move the decimal point to the left. Negative mantissa indicates that the number itself is negative, e.g. `-2.5e3` equals `-2500` and `-2.5e-3` equals `-0.0025`.

The `print` function will display floating-point numbers in decimal notation if they are greater than or equal to `1e-4` and less than `1e16`, but for smaller and larger numbers it will use scientific notation:

```
# This will print 10000000000.0
print(1e10)

# This will print 1e+100
print(1e100)

# This will print 1e-10
print(0.0000000001)
```

When displaying floats, we will usually specify how we would like them to be displayed, using string formatting:

```
# This will print 12.35
print("%.2f" % 12.3456)

# This will print 1.234560e+01
print("%e" % 12.3456)
```

Note that any rounding only affects the display of the numbers. The precision of the number itself is not affected.

### Floating-point operations and precedence

Arithmetic operations for floating-point numbers are the same as those for integers: addition, subtraction, multiplication, division and modulus. They also use the same operators, except for division – the floating-point division operator is `/`. Floating-point operations always produce a floating-point solution. The order of precedence for these operators is the same as those for integer operators.

Often, we will have to decide which type of number to use in a program. Generally, we should use integers for counting and measuring discrete whole numbers. We should use floating-point numbers for measuring things that are continuous.

We can combine integers and floating-point numbers in arithmetic expressions without having to convert them – this is something that Python will do for us automatically. If we perform an arithmetic operation on an integer and a floating-point number, the result will always be a floating-point number.

We can use the integer division operator on floating-point numbers, and vice versa. The two division operators are at the same level in the order of precedence.

**Note:** Python floating-point numbers conform to a standardised format named `IEEE 754`. The standard represents each floating-point number using a small fixed amount of memory, so unlike Python's integers, Python's floating-point numbers have a limited range. The largest floating-point number that can be represented in Python is `2**1023`.

**Note:** Python includes three other types for dealing with numbers:

- `complex` (like floating point but for complex numbers; try `1+5j`)
- `Fraction` (for rational numbers; available in the `fractions` module)
- `Decimal` (for decimal floating-point arithmetic; available in the `decimal` module).

Using these is beyond the scope of this module, but it's worth knowing that they exist in case you have a use for them later.

### Exercise 5

1. Which of the following are Python floating-point numbers? `1`, `1.0`, `1.12e4`, `-3.141759`, `735`, `0.57721566`, `7.5e-3`
2. What is the difference between integer and floating-point division? What is the operator used for integer division? What is the operator used for floating-point division?
3. What are the results of the following operations? Explain why: #. `1.5 + 2` #. `1.5 // 2.0` #. `1.5 / 2.0` #. `1.5 ** 2` #. `1 / 2` #. `-3 // 2`
4. What happens when you evaluate `1 / 0` in the Python console?
5. What happens when you evaluate `1e1000`? What about `-1e1000`? And `type(1e1000)`?

## Strings

A string is a sequence of characters. You should already be familiar with string literals from working with them in the last section. In Python, strings (type `str`) are a special kind of type which is similar to sequence types. In many ways, strings behave in similar ways to lists (type `list`), which we will discuss in a later chapter, but they also have some functionality specific to text.

Many other languages have a different variable type for individual characters – but in Python single characters are just strings with a length of 1.

**Note:** In Python 2, the `str` type used the ASCII encoding. If we wanted to use strings containing Unicode (for example, characters from other alphabets or special punctuation) we had to use the `unicode` type. In Python 3, the `str` type uses Unicode.

## String formatting

We will often need to print a message which is not a fixed string – perhaps we want to include some numbers or other values which are stored in variables. The recommended way to include these variables in our message is to use string formatting syntax:

```
name = "Jane"
age = 23
print("Hello! My name is %s." % name)
print("Hello! My name is %s and I am %d years old." % (name, age))
```

The symbols in the string which start with percent signs (`%`) are placeholders, and the variables which are to be inserted into those positions are given after the string formatting operator, `%`, in the same order in which they appear in the string. If there is only one variable, it doesn't require any kind of wrapper, but if we have more than one we need to put them in a tuple (between round brackets). The placeholder symbols have different letters depending on the type of the variable – `name` is a string, but `age` is an integer. All the variables will be converted to strings before being combined with the rest of the message.

## Escape sequences

An escape sequence (of characters) can be used to denote a special character which cannot be typed easily on a keyboard or one which has been reserved for other purposes. For example, we may want to insert a newline into our string:

```
print('This is one line.\nThis is another line.')
```

If our string is enclosed in single quotes, we will have to escape apostrophes, and we need to do the same for double quotes in a string enclosed in double quotes. An escape sequence starts with a backslash (`\`):

```
print('"Hi! I\'m Jane," she said.')
print("\"Hi! I'm Jane,\" she said.")
```

If we did not escape one of these quotes, Python would treat it as the end quote of our string – and shortly afterwards it would fail to parse the rest of the statement and give us a syntax error:

```
>>> print('"Hi! I'm Jane," she said.')
  File "<stdin>", line 1
    print('"Hi! I'm Jane," she said.')
                 ^
SyntaxError: invalid syntax
```

Some common escape sequences:

| Sequence | Meaning |
|---|---|
| `\\` | literal backslash |
| `\'` | single quote |
| `\"` | double quote |
| `\n` | newline |
| `\t` | tab |

We can also use escape sequences to output unicode characters.

### Raw strings

Sometimes we may need to define string literals which contain many backslashes – escaping all of them can be tedious. We can avoid this by using Python's *raw string* notation. By adding an `r` before the opening quote of the string, we indicate that the contents of the string are exactly what we have written, and that backslashes have no special meaning. For example:

```
# This string ends in a newline
"Hello!\n"

# This string ends in a backslash followed by an 'n'
r"Hello!\n"
```

We most often use raw strings when we are passing strings to some other program which does its *own* processing of special sequences. We want to leave all such sequences untouched in Python, to allow the other program to handle them.

### Triple quotes

In cases where we need to define a long literal spanning multiple lines, or containing many quotes, it may be simplest and most legible to enclose it in triple quotes (either single or double quotes, but of course they must match). Inside the triple quotes, all whitespace is treated literally – if we type a newline it will be reflected in our string. We also don't have to escape any quotes. We must be careful not to include anything that we don't mean to – any indentation will also go inside our string!

These string literals will be identical:

```
string_one = '''"Hello," said Jane.
"Hi," said Bob.'''

string_two = '"Hello," said Jane.\n"Hi," said Bob.'
```

### String operations

We have already introduced a string operation - concatenation (+). It can be used to join two strings. There are many built-in functions which perform operations on strings. String objects also have many useful methods (i.e. functions which are attached to the objects, and accessed with the attribute reference operator, `.`):

```
name = "Jane Smith"

# Find the length of a string with the built-in len function
print(len(name))

# Print the string converted to lowercase
print(name.lower())
# Print the original string
print(name)
```

Why does the last print statement output the original value of `name`? It's because the `lower` method does not change the value of `name`. It returns a modified *copy* of the value. If we wanted to change the value of `name` permanently, we would have to assign the new value to the variable, like this:

```
# Convert the string to lowercase
name = name.lower()
print(name)
```

In Python, strings are *immutable* – that means that we can't modify a string once it has been created. However, we can assign a new string value to an existing variable name.

### Exercise 6

1. Given variables `x` and `y`, use string formatting to print out the values of `x` and `y` and their sum. For example, if `x = 5` and `y = 3` your statement should print `5 + 3 = 8`.
2. Re-write the following strings using single-quotes instead of double-quotes. Make use of escape sequences as needed: #. `"Hi!  I'm Eli."` #. `"The title of the book was \"Good Omens\"."` #. `"Hi! I\'m Sebastien."`
3. Use escape sequences to write a string which represents the letters `a`, `b` and `c` separated by tabs.
4. Use escape sequences to write a string containing the following haiku (with newlines) inside single double-or-single quotes. Then do the same using triple quotes instead of the escape sequences:

```
the first cold shower
even the monkey seems to want
a little coat of straw
```

5. Given a variable `name` containing a string, write a print statement that prints the name and the number of characters in it. For example, if `name = "John"`, your statement should print `John's name has 4 letters.`.
6. What does the following sequence of statements output:

```python
name = "John Smythe"
print(name.lower())
print(name)
```

   Why is the second line output not lowercase?

## Answers to exercises

### Answer to exercise 1

| Syntax error | Reason |
|---|---|
| Person Record | Identifier contains a space. |
| DEFAULT-HEIGHT | Identifier contains a dash. |
| class | Identifier is a keyword. |
| 2totalweight | Identifier starts with a number. |

### Answer to exercise 2

The two statements inside the block defined by the `append_chickens` function are:

```python
text = text + " Rawwwk!"
return text
```

### Answer to exercise 3

The correctly indented code is:

```python
def happy_day(day):
    if day == "monday":
        return ":("
    if day != "monday":
        return ":D"

print(happy_day("sunday"))
print(happy_day("monday"))
```

## Answer to exercise 4

1. The valid Python integers are: `110` and `-39`
2. (a) `15 + 20 * 3`: `75` – `*` has higher precedence than `+`.

   (b) `13 // 2 + 3`: `9` – `//` has higher precedence than `+`.

   (c) `31 + 10 // 3`: `34` – as above.

   (d) `20 % 7 // 3`: `2` – `//` and `%` have equal precedence but are left-associative (so the left-most operation is performed first).

   (e) `2 ** 3 ** 2`: `512` – `**` is right-associative so the right-most exponential is performed first.
3. A `ZeroDivisionError` is raised.

## Answer to exercise 5

#. Only `1` and `735` are not floating-point numbers (they are integers).

1. In integer division the fractional part (remainder) is discarded (although the result is always a float if one of the operands was a float). The Python operator for integer division is `//`. The operator for floating-point division is `/`.
2. (a) `1.5 + 2`: `3.5` – the integer `2` is converted to a floating-point number and then added to `1.5`.

   (b) `1.5 // 2.0`: `0.0` – integer division is performed on the two floating-point numbers and the result is returned (also as a floating-point number).

   (c) `1.5 / 2.0`: `0.75` – floating-point division is performed on the two numbers.

   (d) `1.5 ** 2`: `2.25`

   (e) `1 / 2`: `0.5` – floating-point division of two integers returns a floating-point number.

   (f) `-3 // 2`: `-2` – integer division rounds the result down even for negative numbers.
3. A `ZeroDivisionError` is raised. Note that the error message is slightly different to the one returned by `1 // 0`.
4. `1e1000` is too large to be represented as a floating-point number. Instead the special floating-point value `inf` is returned (`inf` is short for `infinity`). As you will have noticed by inspecting its type, `inf` is really a floating-point number in Python (and not the string `"inf"`). `-1e1000` gives a different special floating-point value, `-inf`, which is short for `minus infinity`). These special values are defined by the `IEEE 754` floating-point specification that Python follows.

### Answer to exercise 6

1. One possible print statement is:

```
print("%s + %s = %s" % (x, y, x + y))
```

2. The equivalent single-quoted strings are: #. `'Hi!  I\'m Eli.'` #. `'The title of the book was "Good Omens".'` #. `'Hi!  I\'m Sebastien.'`

3. `"a\tb\tc"`

4. Using single double-quotes:

```
"the first cold shower\neven the monkey seems to want\na little
coat of straw"
```

Using triple quotes:

```
"""the first cold shower
even the monkey seems to want
a little coat of straw"""
```

5. `print("%s's name has %s letters." % (name, len(name)))`

6. The output is:

```
john smythe
John Smythe
```

The second line is not lowercase because Python strings are immutable and `name.lower()` returns a new string containing the lowercased name.

CHAPTER 3

# Variables and scope

## Variables

Recall that a variable is a label for a location in memory. It can be used to hold a value. In statically typed languages, variables have predetermined types, and a variable can only be used to hold values of that type. In Python, we may reuse the same variable to store values of any type.

A variable is similar to the memory functionality found in most calculators, in that it holds one value which can be retrieved many times, and that storing a new value erases the old. A variable differs from a calculator's memory in that one can have many variables storing different values, and that each variable is referred to by name.

### Defining variables

To define a new variable in Python, we simply assign a value to a label. For example, this is how we create a variable called `count`, which contains an integer value of zero:

```
count = 0
```

This is exactly the same syntax as assigning a new value to an existing variable called `count`. Later in this chapter we will discuss under what circumstances this statement will cause a new variable to be created.

If we try to access the value of a variable which hasn't been defined anywhere yet, the interpreter will exit with a name error.

We can define several variables in one line, but this is usually considered bad style:

```
# Define three variables at once:
count, result, total = 0, 0, 0

# This is equivalent to:
count = 0
result = 0
total = 0
```

In keeping with good programming style, we should make use of meaningful names for variables.

### Variable scope and lifetime

Not all variables are accessible from all parts of our program, and not all variables exist for the same amount of time. Where a variable is accessible and how long it exists depend on how it is defined. We call the part of a program where a variable is accessible its *scope*, and the duration for which the variable exists its *lifetime*.

A variable which is defined in the main body of a file is called a *global* variable. It will be visible throughout the file, and also inside any file which imports that file. Global variables can have unintended consequences because of their wide-ranging effects – that is why we should almost never use them. Only objects which are intended to be used globally, like functions and classes, should be put in the global namespace.

A variable which is defined inside a function is *local* to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing. The parameter names in the function definition behave like local variables, but they contain the values that we pass into the function when we call it. When we use the assignment operator (=) inside a function, its default behaviour is to create a new local variable – unless a variable with the same name is already defined in the local scope.

Here is an example of variables in different scopes:

```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

**Note:** The inside of a class body is also a new local variable scope. Variables which are defined in the class body (but outside any class method) are called *class attributes*. They can be referenced by their bare names within the same scope, but they can also be accessed from outside this scope if we use the attribute access operator (`.`) on a class or an instance (an object which uses that class as its type). An attribute can also be set explicitly on an instance or class from inside a method. Attributes set on instances are called *instance attributes*. Class attributes are shared between all instances of a class, but each instance has its own separate instance attributes. We will look at this in greater detail in the chapter about classes.

## The assignment operator

As we saw in the previous sections, the assignment operator in Python is a single equals sign (=). This operator assigns the value on the right hand side to the variable on the left hand side, sometimes creating the variable first. If the right hand side is an expression (such as an arithmetic expression), it will be evaluated before the assignment occurs. Here are a few examples:

```
a_number = 5              # a_number becomes 5
a_number = total          # a_number becomes the value of total
a_number = total + 5       # a_number becomes the value of total + 5
a_number = a_number + 1    # a_number becomes the value of a_number + 1
```

The last statement might look a bit strange if we were to interpret = as a mathematical equals sign – clearly a number cannot be equal to the same number plus one! Remember that = is an assignment operator – this statement is assigning a *new* value to the variable `a_number` which is equal to the *old* value of `a_number` plus one.

Assigning an initial value to variable is called *initialising* the variable. In some languages defining a variable can be done in a separate step before the first value assignment. It is thus possible in those languages for a variable to be defined but not have a value – which could lead to errors or unexpected behaviour if we try to use the value before it has been assigned. In Python a variable is defined and assigned a value in a single step, so we will almost never encounter situations like this.

The left hand side of the assignment statement must be a valid target:

```
# this is fine:
a = 3

# these are all illegal:
3 = 4
3 = a
a + b = 3
```

An assignment statement may have multiple targets separated by equals signs. The expression on the right hand side of the last equals sign will be assigned to all the targets. All the targets must be valid:

```
# both a and b will be set to zero:
a = b = 0

# this is illegal, because we can't set 0 to b:
a = 0 = b
```

## Compound assignment operators

We have already seen that we can assign the result of an arithmetic expression to a variable:

```
total = a + b + c + 50
```

Counting is something that is done often in a program. For example, we might want to keep count of how many times a certain event occurs by using a variable called `count`. We would initialise this variable to zero and add one to it every time the event occurs. We would perform the addition with this statement:

```
count = count + 1
```

This is in fact a very common operation. Python has a shorthand operator, +=, which lets us express it more cleanly, without having to write the name of the variable twice:

```
# These statements mean exactly the same thing:
count = count + 1
count += 1

# We can increment a variable by any number we like.
count += 2
count += 7
count += a + b
```

There is a similar operator, `-=`, which lets us decrement numbers:

```python
# These statements mean exactly the same thing:
count = count - 3
count -= 3
```

Other common compound assignment operators are given in the table below:

| Operator | Example | Equivalent to |
|---|---|---|
| `+=` | `a += 5` | `a = a + 5` |
| `-=` | `a -= 5` | `a = a - 5` |
| `*=` | `a *= 5` | `a = a * 5` |
| `/=` | `a /= 5` | `a = a / 5` |
| `%=` | `a %= 5` | `a = a % 5` |

## More about scope: crossing boundaries

What if we want to access a global variable from inside a function? It is possible, but doing so comes with a few caveats:

```python
a = 0

def my_function():
    print(a)

my_function()
```

The print statement will output `0`, the value of the global variable `a`, as you probably expected. But what about this program?

```python
a = 0

def my_function():
    a = 3
    print(a)

my_function()

print(a)
```

When we call the function, the print statement inside outputs `3` – but why does the print statement at the end of the program output `0`?

By default, the assignment statement creates variables in the local scope. So the assignment inside the function does not modify the global variable `a` – it creates a new local variable called `a`, and assigns the value `3` to that variable. The first print statement outputs the value of the new local variable – because if a local variable has the same name as a global variable the local variable will always take precedence. The last print statement prints out the global variable, which has remained unchanged.

What if we really want to modify a global variable from inside a function? We can use the `global` keyword:

```python
a = 0

def my_function():
    global a
    a = 3
    print(a)

my_function()
```

```
print(a)
```

We may not refer to both a global variable and a local variable by the same name inside the same function. This program will give us an error:

```
a = 0

def my_function():
    print(a)
    a = 3
    print(a)

my_function()
```

Because we haven't declared `a` to be global, the assignment in the second line of the function will create a local variable `a`. This means that we can't refer to the global variable `a` elsewhere in the function, even before this line! The first print statement now refers to the local variable `a` – but this variable doesn't have a value in the first line, because we haven't assigned it yet!

Note that it is usually very bad practice to access global variables from inside functions, and even worse practice to modify them. This makes it difficult to arrange our program into logically encapsulated parts which do not affect each other in unexpected ways. If a function needs to access some external value, we should pass the value into the function as a parameter. If the function is a method of an object, it is sometimes appropriate to make the value an attribute of the same object – we will discuss this in the chapter about object orientation.

**Note:** There is also a `nonlocal` keyword in Python – when we nest a function inside another function, it allows us to modify a variable in the outer function from inside the inner function (or, if the function is nested multiple times, a variable in one of the outer functions). If we use the `global` keyword, the assignment statement will create the variable in the global scope if it does not exist already. If we use the `nonlocal` keyword, however, the variable must be defined, because it is impossible for Python to determine in which scope it should be created.

## Exercise 1

1. Describe the scope of the variables `a`, `b`, `c` and `d` in this example:

```
def my_function(a):
    b = a - 2
    return b

c = 3

if c > 2:
    d = my_function(5)
    print(d)
```

2. What is the lifetime of these variables? When will they be created and destroyed?
3. Can you guess what would happen if we were to assign `c` a value of `1` instead?
4. Why would this be a problem? Can you think of a way to avoid it?

## Modifying values

### Constants

In some languages, it is possible to define special variables which can be assigned a value only once – once their values have been set, they cannot be changed. We call these kinds of variables *constants*. Python does not allow us to set such a restriction on variables, but there is a widely used convention for marking certain variables to indicate that their values are not meant to change: we write their names in all caps, with underscores separating words:

```python
# These variables are "constants" by convention:
NUMBER_OF_DAYS_IN_A_WEEK = 7
NUMBER_OF_MONTHS_IN_A_YEAR = 12

# Nothing is actually stopping us from redefining them...
NUMBER_OF_DAYS_IN_A_WEEK = 8

# ...but it's probably not a good idea.
```

Why do we bother defining variables that we don't intend to change? Consider this example:

```python
MAXIMUM_MARK = 80

tom_mark = 58
print(("Tom's mark is %.2f%%" % (tom_mark / MAXIMUM_MARK * 100)))
# %% is how we escape a literal % inside a string
```

There are several good reasons to define `MAXIMUM_MARK` instead of just writing `80` inside the print statement. First, this gives the number a descriptive label which explains what it is – this makes the code more understandable. Second, we may eventually need to refer to this number in our program more than once. If we ever need to update our code with a new value for the maximum mark, we will only have to change it in one place, instead of finding every place where it is used – such replacements are often error-prone.

Literal numbers scattered throughout a program are known as "magic numbers" – using them is considered poor coding style. This does not apply to small numbers which are considered self-explanatory – it's easy to understand why a total is initialised to zero or incremented by one.

Sometimes we want to use a variable to distinguish between several discrete options. It is useful to refer to the option values using constants instead of using them directly if the values themselves have no intrinsic meaning:

```python
# We define some options
LOWER, UPPER, CAPITAL = 1, 2, 3

name = "jane"
# We use our constants when assigning these values...
print_style = UPPER

# ...and when checking them:
if print_style == LOWER:
    print(name.lower())
elif print_style == UPPER:
    print(name.upper())
elif print_style == CAPITAL:
    print(name.capitalize())
else:
    # Nothing prevents us from accidentally setting print_style to 4, 90 or
    # "spoon", so we put in this fallback just in case:
    print("Unknown style option!")
```

In the above example, the values `1`, `2` and `3` are not important – they are completely meaningless. We could equally well use `4`, `5` and `6` or the strings `'lower'`, `'upper'` and `'capital'`. The only important thing is that the three values must be different. If we used the numbers directly instead of the constants the program would be much more confusing to read. Using meaningful strings would make the code more readable, but we could accidentally make a spelling mistake while setting one of the values and not notice – if we mistype the name of one of the constants we are more likely to get an error straight away.

Some Python libraries define common constants for our convenience, for example:

```
# we need to import these libraries before we use them
import string
import math
import re

# All the lowercase ASCII letters: 'abcdefghijklmnopqrstuvwxyz'
print(string.ascii_lowercase)

# The mathematical constants pi and e, both floating-point numbers
print(math.pi) # ratio of circumference of a circle to its diameter
print(math.e) # natural base of logarithms

# This integer is an option which we can pass to functions in the re
# (regular expression) library.
print(re.IGNORECASE)
```

Note that many built-in constants don't follow the all-caps naming convention.

## Mutable and immutable types

Some *values* in python can be modified, and some cannot. This does not ever mean that we can't change the value of a variable – but if a variable contains a value of an *immutable type*, we can only assign it a *new value*. We cannot *alter the existing value* in any way.

Integers, floating-point numbers and strings are all immutable types – in all the previous examples, when we changed the values of existing variables we used the assignment operator to assign them new values:

```
a = 3
a = 2

b = "jane"
b = "bob"
```

Even this operator doesn't modify the value of `total` in-place – it also assigns a new value:

```
total += 4
```

We haven't encountered any mutable types yet, but we will use them extensively in later chapters. Lists and dictionaries are mutable, and so are most objects that we are likely to write ourselves:

```
# this is a list of numbers
my_list = [1, 2, 3]
my_list[0] = 5 # we can change just the first element of the list
print(my_list)

class MyClass(object):
    pass # this is a very silly class

# Now we make a very simple object using our class as a type
my_object = MyClass()
```

```
# We can change the values of attributes on the object
my_object.some_property = 42
```

## More about input

In the earlier sections of this unit we learned how to make a program display a message using the `print` function or read a string value from the user using the `input` function. What if we want the user to input numbers or other types of variables? We still use the `input` function, but we must convert the string values returned by `input` to the types that we want. Here is a simple example:

```
height = int(input("Enter height of rectangle: "))
width = int(input("Enter width of rectangle: "))

print("The area of the rectangle is %d" % (width * height))
```

`int` is a function which converts values of various types to ints. We will discuss type conversion in greater detail in the next section, but for now it is important to know that `int` will not be able to convert a string to an integer if it contains anything except digits. The program above will exit with an error if the user enters `"aaa"`, `"zzz10"` or even `"7.5"`. When we write a program which relies on user input, which can be incorrect, we need to add some safeguards so that we can recover if the user makes a mistake. For example, we can detect if the user entered bad input and exit with a nicer error message:

```
try:
    height = int(input("Enter height of rectangle: "))
    width = int(input("Enter width of rectangle: "))
except ValueError as e: # if a value error occurs, we will skip to this point
    print("Error reading height and width: %s" % e)
```

This program will still only attempt to read in the input once, and exit if it is incorrect. If we want to keep asking the user for input until it is correct, we can do something like this:

```
correct_input = False # this is a boolean value -- it can be either true or false.

while not correct_input: # this is a while loop
    try:
        height = int(input("Enter height of rectangle: "))
        width = int(input("Enter width of rectangle: "))
    except ValueError:
        print("Please enter valid integers for the height and width.")
    else: # this will be executed if there is no value error
        correct_input = True
```

We will learn more about boolean values, loops and exceptions later.

## Example: calculating petrol consumption of a car

In this example, we will write a simple program which asks the user for the distance travelled by a car, and the monetary value of the petrol that was used to cover that distance. From this information, together with the price per litre of petrol, the program will calculate the efficiency of the car, both in litres per 100 kilometres and and kilometres per litre.

First we will define the petrol price as a constant at the top. This will make it easy for us to update the price when it changes on the first Wednesday of every month:

```
PETROL_PRICE_PER_LITRE = 4.50
```

When the program starts,we want to print out a welcome message:

```
print("*** Welcome to the fuel efficiency calculator! ***\n")
# we add an extra blank line after the message with \n
```

Ask the user for his or her name:

```
name = input("Enter your name: ")
```

Ask the user for the distance travelled:

```
# float is a function which converts values to floating-point numbers.
distance_travelled = float(input("Enter distance travelled in km: "))
```

Then ask the user for the amount paid:

```
amount_paid = float(input("Enter monetary value of fuel bought for the trip: R"))
```

Now we will do the calculations:

```
fuel_consumed = amount_paid / PETROL_PRICE_PER_LITRE

efficiency_l_per_100_km = fuel_consumed / distance_travelled * 100
efficiency_km_per_l = distance_travelled / fuel_consumed
```

Finally, we output the results:

```
print("Hi, %s!" % name)
print("Your car's efficiency is %.2f litres per 100 km." % efficiency_l_per_100_km)
print("This means that you can travel %.2f km on a litre of petrol." % efficiency_km_per_l)

# we add an extra blank line before the message with \n
print("\nThanks for using the program.")
```

### Exercise 2

1. Write a Python program to convert a temperature given in degrees Fahrenheit to its equivalent in degrees Celsius. You can assume that **T_c = (5/9) x (T_f - 32)**, where **T_c** is the temperature in °C and **T_f** is the temperature in °F. Your program should ask the user for an input value, and print the output. The input and output values should be floating-point numbers.
2. What could make this program crash? What would we need to do to handle this situation more gracefully?

## Type conversion

As we write more programs, we will often find that we need to convert data from one type to another, for example from a string to an integer or from an integer to a floating-point number. There are two kinds of type conversions in Python: implicit and explicit conversions.

### Implicit conversion

Recall from the section about floating-point operators that we can arbitrarily combine integers and floating-point numbers in an arithmetic expression – and that the result of any such expression will always be a floating-point number. This is because Python will convert the integers to floating-point numbers before evaluating the expression. This is an *implicit conversion* – we don't have to convert anything ourselves. There is usually no loss of precision when an integer is converted to a floating-point number.

For example, the integer `2` will automatically be converted to a floating-point number in the following example:

```
result = 8.5 * 2
```

`8.5` is a `float` while `2` is an `int`. Python will automatically convert operands so that they are of the same type. In this case this is achieved if the integer `2` is converted to the floating-point equivalent `2.0`. Then the two floating-point numbers can be multiplied.

Let's have a look at a more complex example:

```
result = 8.5 + 7 // 3 - 2.5
```

Python performs operations according to the order of precedence, and decides whether a conversion is needed on a per-operation basis. In our example `//` has the highest precedence, so it will be processed first. `7` and `3` are both integers and `//` is the integer division operator – the result of this operation is the integer `2`. Now we are left with `8.5 + 2 - 2.5`. The addition and subtraction are at the same level of precedence, so they are evaluated left-to-right, starting with addition. First `2` is converted to the floating-point number `2.0`, and the two floating-point numbers are added, which leaves us with `10.5 - 2.5`. The result of this floating-point subtraction is `2.0`, which is assigned to `result`.

## Explicit conversion

Converting numbers from `float` to `int` will result in a loss of precision. For example, try to convert `5.834` to an `int` – it is not possible to do this without losing precision. In order for this to happen, we must explicitly tell Python that we are aware that precision will be lost. For example, we need to tell the compiler to convert a `float` to an `int` like this:

```
i = int(5.834)
```

The `int` function converts a `float` to an `int` by discarding the fractional part – it will always round down! If we want more control over the way in which the number is rounded, we will need to use a different function:

```
# the floor and ceil functions are in the math module
import math

# ceil returns the closest integer greater than or equal to the number
# (so it always rounds up)
i = math.ceil(5.834)

# floor returns the closest integer less than or equal to the number
# (so it always rounds down)
i = math.floor(5.834)

# round returns the closest integer to the number
# (so it rounds up or down)
# Note that this is a built-in function -- we don't need to import math to use it.
i = round(5.834)
```

Explicit conversion is sometimes also called *casting* – we may read about a `float` being *cast* to `int` or vice-versa.

## Converting to and from strings

As we saw in the earlier sections, Python seldom performs implicit conversions to and from `str` – we usually have to convert values explicitly. If we pass a single number (or any other value) to the `print` function, it will be converted to a string automatically – but if we try to add a number and a string, we will get an error:

```
# This is OK
print(5)
print(6.7)

# This is not OK
print("3" + 4)

# Do you mean this...
print("3%d" % 4) # concatenate "3" and "4" to get "34"

# Or this?
print(int("3") + 4) # add 3 and 4 to get 7
```

To convert numbers to strings, we can use string formatting – this is usually the cleanest and most readable way to insert multiple values into a message. If we want to convert a single number to a string, we can also use the `str` function explicitly:

```
# These lines will do the same thing
print("3%d" % 4)
print("3" + str(4))
```

## More about conversions

In Python, functions like `str`, `int` and `float` will try to convert *anything* to their respective types – for example, we can use the `int` function to convert strings to integers or to convert floating-point numbers to integers. Note that although `int` can convert a float to an integer it can't convert a string containing a float to an integer directly!

```
# This is OK
int("3")

# This is OK
int(3.7)

# This is not OK
int("3.7") # This is a string representation of a float, not an integer!

# We have to convert the string to a float first
int(float("3.7"))
```

Values of type `bool` can contain the value `True` or `False`. These values are used extensively in conditional statements, which execute or do not execute parts of our program depending on some binary condition:

```
my_flag = True

if my_flag:
    print("Hello!")
```

The condition is often an expression which evaluates to a boolean value:

```
if 3 > 4:
    print("This will not be printed.")
```

However, almost any value can implicitly be converted to a boolean if it is used in a statement like this:

```
my_number = 3

if my_number:
    print("My number is non-zero!")
```

This usually behaves in the way that you would expect: non-zero numbers are `True` values and zero is `False`. However, we need to be careful when using strings – the empty string is treated as `False`, but any other string is `True` – even `"0"` and `"False"`!

```
# bool is a function which converts values to booleans
bool(34) # True
bool(0) # False
bool(1) # True

bool("") # False
bool("Jane") # True
bool("0") # True!
bool("False") # Also True!
```

### Exercise 3

1. Convert `"8.8"` to a float.
2. Convert `8.8` to an integer (with rounding).
3. Convert `"8.8"` to an integer (with rounding).
4. Convert `8.8` to a string.
5. Convert `8` to a string.
6. Convert `8` to a float.
7. Convert `8` to a boolean.

## Answers to exercises

### Answer to exercise 1

1. `a` is a local variable in the scope of `my_function` because it is an argument name. `b` is also a local variable inside `my_function`, because it is assigned a value inside `my_function`. `c` and `d` are both global variables. It doesn't matter that `d` is created inside an `if` block, because the inside of an `if` block is not a new scope – everything inside the block is part of the same scope as the outside (in this case the global scope). Only function definitions (which start with `def`) and class definitions (which start with `class`) indicate the start of a new level of scope.
2. Both `a` and `b` will be created every time `my_function` is called and destroyed when `my_function` has finished executing. `c` is created when it is assigned the value `3`, and exists for the remainder of the program's execution. `d` is created inside the `if` block (when it is assigned the value which is returned from the function), and also exists for the remainder of the program's execution.
3. As we will learn in the next chapter, `if` blocks are executed *conditionally*. If `c` were not greater than `3` in this program, the `if` block would not be executed, and if that were to happen the variable `d` would never be created.

4. We may use the variable later in the code, assuming that it always exists, and have our program crash unexpectedly if it doesn't. It is considered poor coding practice to allow a variable to be defined or undefined depending on the outcome of a conditional statement. It is better to ensure that is always defined, no matter what – for example, by assigning it some default value at the start. It is much easier and cleaner to check if a variable has the default value than to check whether it exists at all.

## Answer to exercise 2

1. Here is an example program:

```
T_f = float(input("Please enter a temperature in °F: "))
T_c = (5/9) * (T_f - 32)
print("%g°F = %g°C" % (T_f, T_c))
```

**Note:** The formatting symbol `%g` is used with floats, and instructs Python to pick a sensible human-readable way to display the float.

2. The program could crash if the user enters a value which cannot be converted to a floating-point number. We would need to add some kind of error checking to make sure that this doesn't happen – for example, by storing the string value and checking its contents. If we find that the entered value is invalid, we can either print an error message and exit or keep prompting the user for input until valid input is entered.

## Answer to exercise 3

Here are example answers:

```
import math

a_1 = float("8.8")
a_2 = math.round(8.8)
a_3 = math.round("8.8")
a_4 = "%g" % 8.8
a_5 = "%d" % 8
a_6 = float(8)
a_7 = bool(8)
```

CHAPTER 4

# Selection control statements

## Introduction

In the last chapter, you were introduced to the concept of flow of control: the sequence of statements that the computer executes. In procedurally written code, the computer usually executes instructions in the order that they appear. However, this is not always the case. One of the ways in which programmers can change the flow of control is the use of selection control statements.

In this chapter we will learn about selection statements, which allow a program to choose when to execute certain instructions. For example, a program might choose how to proceed on the basis of the user's input. As you will be able to see, such statements make a program more versatile.

We will also look at different kinds of programming errors and discuss strategies for finding and correcting them.

## Selection: `if` statement

People make decisions on a daily basis. What should I have for lunch? What should I do this weekend? Every time you make a decision you base it on some criterion. For example, you might decide what to have for lunch based on your mood at the time, or whether you are on some kind of diet. After making this decision, you act on it. Thus decision-making is a two step process – first deciding what to do based on a criterion, and secondly taking an action.

Decision-making by a computer is based on the same two-step process. In Python, decisions are made with the `if` statement, also known as the selection statement. When processing an `if` statement, the computer first evaluates some criterion or condition. If it is met, the specified action is performed. Here is the syntax for the `if` statement:

```
if condition:
    if_body
```

When it reaches an `if` statement, the computer only executes the body of the statement only if the condition is true. Here is an example in Python, with a corresponding flowchart:

```
if age < 18:
    print("Cannot vote")
```

blockdiag-17428c50dc8e8b45de1cacd260b64e0de9806cc5.png

As we can see from the flowchart, the instructions in the `if` body are only executed if the condition is met (i.e. if it is true). If the condition is not met (i.e. false), the instructions in the `if` body are skipped.

## Relational operators

Many `if` statements compare two values in order to make a decision. In the last example, we compared the variable `age` to the integer `18` to test if age less than 18. We used the operator < for the comparison. This operator is one of the relational operators that can be used in Python. The table below shows Python's relational operators.

| Operator | Description | Example |
|---|---|---|
| == | equal to | if (age == 18) |
| != | not equal to | if (score != 10) |
| > | greater than | if (num_people > 50) |
| < | less than | if (price < 25) |
| >= | greater than or equal to | if (total >= 50) |
| <= | less than or equal to | if (value <= 30) |

Note that the condition statement can either be true or false. Also note that the operator for equality is == – a double equals sign. Remember that =, the single equals sign, is the assignment operator. If we accidentally use = when we mean ==, we are likely to get a syntax error:

```
>>> if choice = 3:
File "<stdin>", line 1
  if choice = 3:
            ^
SyntaxError: invalid syntax
```

This is correct:

```
if choice == 3:
    print("Thank you for using this program.")
```

**Note:** in some languages, an assignment statement *is* a valid conditional expression: it is evaluated as true if the assignment is executed successfully, and as false if it is not. In such languages, it is easier to use the wrong operator by accident and not notice!

## Value vs identity

So far, we have only compared integers in our examples. We can also use any of the above relational operators to compare floating-point numbers, strings and many other types:

```
# we can compare the values of strings
if name == "Jane":
    print("Hello, Jane!")

# ... or floats
if size < 10.5:
    print(size)
```

When comparing variables using ==, we are doing a *value* comparison: we are checking whether the two variables have the same value. In contrast to this, we might want to know if two objects such as lists, dictionaries or custom objects that we have created ourselves are *the exact same object*. This is a test of *identity*. Two objects might have identical contents, but be two different objects. We compare identity with the `is` operator:

```python
a = [1,2,3]
b = [1,2,3]

if a == b:
    print("These lists have the same value.")

if a is b:
    print("These lists are the same list.")
```

It is generally the case (with some caveats) that if two variables are the same object, they are also equal. The reverse is not true – two variables could be equal in value, but not the same object.

To test whether two objects are *not* the same object, we can use the `is not` operator:

```python
if a is not b:
    print("a and b are not the same object.")
```

**Note:** In many cases, variables of built-in immutable types which have the same value will also be identical. In some cases this is because the Python interpreter saves memory (and comparison time) by representing multiple values which are equal by the same object. You shouldn't rely on this behaviour and make value comparisons using `is` – if you want to compare values, always use `==`.

### Using indentation

In the examples which have appeared in this chapter so far, there has only been one statement appearing in the `if` body. Of course it is possible to have more than one statement there; for example:

```python
if choice == 1:
    count += 1
    print("Thank you for using this program.")
print("Always print this.") # this is outside the if block
```

The interpreter will treat all the statements inside the indented block as one statement – it will process all the instructions in the block before moving on to the next instruction. This allows us to specify multiple instructions to be executed when the condition is met.

`if` is referred to as a *compound statement* in Python because it combines multiple other statements together. A compound statement comprises one or more *clauses*, each of which has a *header* (like `if`) and a *suite* (which is a list of statements, like the `if` body). The contents of the suite are delimited with indentation – we have to indent lines to the same level to put them in the same block.

### The `else` clause

An optional part of an if statement is the `else` clause. It allows us to specify an alternative instruction (or set of instructions) to be executed if the condition is *not* met:

```python
if condition:
    if_body
else:
    else_body
```

To put it another way, the computer will execute the `if` body if the condition is true, otherwise it will execute the `else` body. In the example below, the computer will add 1 to x if it is zero, otherwise it will subtract 1 from x:

```
if x == 0:
    x += 1
else:
    x -= 1
```

This flowchart represents the same statement:

blockdiag-111c4bef96ad0cd3b492738e37fc40ddc7539cd4.png

The computer will execute one of the branches before proceeding to the next instruction.

## Exercise 1

1. Which of these fragments are valid and invalid first lines of `if` statements? Explain why:
   (a) `if (x > 4)`
   (b) `if x == 2`
   (c) `if (y =< 4)`
   (d) `if (y = 5)`
   (e) `if (3 <= a)`
   (f) `if (1 - 1)`
   (g) `if ((1 - 1) <= 0)`
   (h) `if (name == "James")`
2. What is the output of the following code fragment? Explain why.

```
x = 2

if x > 3:
    print("This number")
print("is greater")
print("than 3.")
```

3. How can we simplify these code fragments?

```
(a) if bool(a) == True:
        print("a is true")
```

```
(b) if x > 50:
        b += 1
        a = 5
    else:
        b -= 1
        a = 5
```

## More on the `if` statement

### Nested `if` statements

In some cases you may want one decision to depend on the result of an earlier decision. For example, you might only have to choose which shop to visit if you decide that you are going to do your shopping, or what to have for dinner after you have made a decision that you are hungry enough for dinner.

blockdiag-57fa823c6ffd39e4c72811bb9efc8701319929e6.png

In Python this is equivalent to putting an `if` statement within the body of either the `if` or the `else` clause of another `if` statement. The following code fragment calculates the cost of sending a small parcel. The post office charges R5 for the first 300g, and R2 for every 100g thereafter (rounded up), up to a maximum weight of 1000g:

```
if weight <= 1000:
    if weight <= 300:
        cost = 5
    else:
        cost = 5 + 2 * round((weight - 300)/100)

    print("Your parcel will cost R%d." % cost)

else:
    print("Maximum weight for small parcel exceeded.")
    print("Use large parcel service instead.")
```

Note that the bodies of the outer `if` and `else` clauses are indented, and the bodies of the inner `if` and `else` clauses are indented one more time. It is important to keep track of indentation, so that each statement is in the correct block. It doesn't matter that there's an empty line between the last line of the inner `if` statement and the following print statement – they are still both part of the same block (the outer `if` body) because they are indented by the same amount. We can use empty lines (sparingly) to make our code more readable.

### The `elif` clause and `if` ladders

The addition of the else keyword allows us to specify actions for the case in which the condition is false. However, there may be cases in which we would like to handle more than two alternatives. For example, here is a flowchart of a program which works out which grade should be assigned to a particular mark in a test:

blockdiag-b494fae1cfe99f5b6f81ffeea3301ff3f26db08d.png

We should be able to write a code fragment for this program using nested if statements. It might look something like this:

```
if mark >= 80:
    grade = A
```

```
else:
    if mark >= 65:
        grade = B
    else:
        if mark >= 50:
            grade = C
        else:
            grade = D
```

This code is a bit difficult to read. Every time we add a nested `if`, we have to increase the indentation, so all of our alternatives are indented differently. We can write this code more cleanly using `elif` clauses:

```
if mark >= 80:
    grade = A
elif mark >= 65:
    grade = B
elif mark >= 50:
    grade = C
else:
    grade = D
```

Now all the alternatives are clauses of one `if` statement, and are indented to the same level. This is called an *if ladder*. Here is a flowchart which more accurately represents this code:

blockdiag-92200f5670e7241c7f70e1c06312813511774384.png

The default (catch-all) condition is the `else` clause at the end of the statement. If none of the conditions specified earlier is matched, the actions in the `else` body will be executed. It is a good idea to include a final `else` clause in each ladder to make sure that we are covering all cases, especially if there's a possibility that the options will change in the future. Consider the following code fragment:

```
if course_code == "CSC":
    department_name = "Computer Science"
elif course_code == "MAM":
    department_name = "Mathematics and Applied Mathematics"
elif course_code == "STA":
    department_name = "Statistical Sciences"
else:
    department_name = None
    print("Unknown course code: %s" % course_code)

if department_name:
    print("Department: %s" % department_name)
```

What if we unexpectedly encounter an informatics course, which has a course code of `"INF"`? The catch-all `else` clause will be executed, and we will immediately see a printed message that this course code is unsupported. If the `else` clause were omitted, we might not have noticed that anything was wrong until we tried to use `department_name` and discovered that it had never been assigned a value. Including the `else` clause helps us to pick up potential errors caused by missing options early.

## Boolean values, operators and expressions

### The `bool` type

In Python there is a value type for variables which can either be true or false: the boolean type, `bool`. The true value is `True` and the false value is `False`. Python will implicitly convert any other value type to a boolean if we use it like a boolean, for example as a condition in an `if` statement. We will almost never have to cast values to `bool` explicitly. We also don't have to use the == operator explicitly to check if a variable's value evaluates to `True` – we can use the variable name by itself as a condition:

```
name = "Jane"

# This is shorthand for checking if name evaluates to True:
if name:
    print("Hello, %s!" % name)

# It means the same thing as this:
if bool(name) == True:
    print("Hello, %s!" % name)

# This won't give us the answer we expect:
if name == True:
    print("Hello, %s!" % name)
```

Why won't the last `if` statement do what we expect? If we cast the string `"Jane"` to a boolean, it will be equal to `True`, but it isn't equal to `True` while it's still a string – so the condition in the last `if` statement will evaluate to `False`. This is why we should always use the shorthand syntax, as shown in the first statement – Python will then do the implicit cast for us.

**Note:** For historical reasons, the numbers `0` and `0.0` are actually equal to `False` and `1` and `1.0` are equal to `True`. They are not, however, identical objects – you can test this by comparing them with the `is` operator.

At the end of the previous chapter, we discussed how Python converts values to booleans implicitly. Remember that all non-zero numbers and all non-empty strings are `True` and zero and the empty string (`""`) are `False`. Other built-in data types that can be considered to be "empty" or "not empty" follow the same pattern.

### Boolean operations

Decisions are often based on more than one factor. For example, you might decide to buy a shirt only if you like it AND it costs less than R100. Or you might decide to go out to eat tonight if you don't have anything in the fridge OR you don't feel like cooking. You can also alter conditions by negating them – for example you might only want to go to the concert tomorrow if it is NOT raining. Conditions which consist of simpler conditions joined together with AND, OR and NOT are referred to as *compound conditions*. These operators are known as *boolean operators*.

### The `and` operator

The AND operator in Python is `and`. A compound expression made up of two subexpressions and the `and` operator is only true when *both* subexpressions are true:

```
if mark >= 50 and mark < 65:
    print("Grade B")
```

The compound condition is only true if the given mark is less than 50 *and* it is less than 65. The `and` operator works in the same way as its English counterpart. We can define the `and` operator formally with a truth table such as the one below. The table shows the truth value of `a and b` for every possible combination of subexpressions `a` and `b`. For example, if `a` is true and `b` is true, then `a and b` is true.

| a | b | a and b |
|---|---|---|
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

`and` is a binary operator so it must be given two operands. Each subexpression must be a valid complete expression:

```
# This is correct:
if (x > 3 and x < 300):
    x += 1

# This will give us a syntax error:
if (x > 3 and < 300): # < 300 is not a valid expression!
    x += 1
```

We can join three or more subexpressions with `and` – they will be evaluated from left to right:

```
condition_1 and condition_2 and condition_3 and condition_4
# is the same as
((condition_1 and condition_2) and condition_3) and condition_4
```

**Note:** for the special case of testing whether a number falls within a certain range, we don't have to use the `and` operator at all. Instead of writing `mark >= 50 and mark < 65` we can simply write `50 <= mark < 65`. This doesn't work in many other languages, but it's a useful feature of Python.

## Short-circuit evaluation

Note that if `a` is false, the expression `a and b` is false whether `b` is true or not. The interpreter can take advantage of this to be more efficient: if it evaluates the first subexpression in an AND expression to be false, it does not bother to evaluate the second subexpression. We call `and` a *shortcut operator* or *short-circuit* operator because of this behaviour.

This behaviour doesn't just make the interpreter slightly faster – we can also use it to our advantage when writing programs. Consider this example:

```
if x > 0 and 1/x < 0.5:
    print("x is %f" % x)
```

What if x is zero? If the interpreter were to evaluate both of the subexpressions, we would get a divide by zero error. But because `and` is a short-circuit operator, the second subexpression will only be evaluated if the first subexpression is true. If x is zero, it will evaluate to false, and the second subexpression will not be evaluated at all.

We could also have used nested `if` statements, like this:

```
if x > 0:
    if 1/x < 0.5:
        print("x is %f" % x)
```

Using `and` instead is more compact and readable – especially if we have more than two conditions to check. These two snippets do the same thing:

```
if x != 0:
    if y != 0:
        if z != 0:
            print(1/(x*y*z))

if x != 0 and y != 0 and z != 0:
    print(1/(x*y*z))
```

This often comes in useful if we want to access an object's attribute or an element from a list or a dictionary, and we first want to check if it exists:

```
if hasattr(my_person, "name") and len(myperson.name) > 30:
    print("That's a long name, %s!" % myperson.name)

if i < len(mylist) and mylist[i] == 3:
    print("I found a 3!")

if key in mydict and mydict[key] == 3:
    print("I found a 3!")
```

## The `or` operator

The OR operator in Python is `or`. A compound expression made up of two subexpressions and the `or` operator is true when *at least one* of the subexpressions is true. This means that it is only false in the case where both subexpressions are false, and is true for all other cases. This can be seen in the truth table below:

| a | b | a or b |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

The following code fragment will print out a message if the given age is less than 0 *or* if it is more than 120:

```
if age < 0 or age > 120:
    print("Invalid age: %d" % age)
```

The interpreter also performs a short-circuit evaluation for `or` expressions. If it evaluates the first subexpression to be true, it will not bother to evaluate the second, because this is sufficient to determine that the whole expression is true.

The || operator is also binary:

```
# This is correct:
if x < 3 or x > 300:
    x += 1

# This will give us a syntax error:
if x < 3 or > 300: # > 300 is not a valid expression!
    x += 1

# This may not do what we expect:
if x == 2 or 3:
    print("x is 2 or 3")
```

The last example won't give us an error, because `3` is a valid subexpression – and since it is a non-zero number it evaluates to `True`. So the last `if` body will always execute, regardless of the value of `x`!

## The `not` operator

The NOT operator, `not` in Python, is a unary operator: it only requires one operand. It is used to reverse an expression, as shown in the following truth table:

| a | not a |
|---|---|
| True | False |
| False | True |

The `not` operator can be used to write a less confusing expression. For example, consider the following example in which we want to check whether a string *doesn't* start with "A":

```
if name.startswith("A"):
    pass # a statement body can't be empty -- this is an instruction which does nothing.
else:
    print("'%s' doesn't start with A!" % s)

# That's a little clumsy -- let's use "not" instead!
if not name.startswith("A"):
    print("'%s' doesn't start with A!" % s)
```

Here are a few other examples:

```
# Do something if a flag is False:
if not my_flag:
    print("Hello!")

# This...
if not x == 5:
    x += 1

# ... is equivalent to this:
if x != 5:
    x += 1
```

## Precedence rules for boolean expressions

Here is a table indicating the relative level of precedence for all the operators we have seen so far, including the arithmetic, relational and boolean operators.

| Operators |
|---|
| `()` (highest) |
| `**` |
| `*, /, %` |
| `+, -` |
| `<, <=, >, >= ==, !=` |
| `is, is not` |
| `not` |
| `and` |
| `or` (lowest) |

It is always a good idea to use brackets to clarify what we mean, even though we can rely on the order of precedence above. Brackets can make complex expressions in our code easier to read and understand, and reduce the opportunity for errors.

## DeMorgan's law for manipulating boolean expressions

The `not` operator can make expressions more difficult to understand, especially if it is used multiple times. Try only to use the `not` operator where it makes sense to have it. Most people find it easier to read positive statements than negative ones. Sometimes we can use the opposite relational operator to avoid using the `not` operator, for example:

```
if not mark < 50:
    print("You passed")

# is the same as

if mark >= 50:
    print("You passed")
```

This table shows each relational operator and its opposite:

| Operator | Opposite |
|---|---|
| == | != |
| > | <= |
| < | >= |

There are other ways to rewrite boolean expressions. The 19th-century logician DeMorgan proved two properties of negation that we can use.

Consider this example in English: *it is not both cool and rainy today.* Does this sentence mean the same thing as *it is not cool and not rainy today?* No, the first sentence says that both conditions are not true, but either one of them could be true. The correct equivalent sentence is *it is not cool or not rainy today.*

We have just used DeMorgan's law to distribute NOT over the first sentence. Formally, DeMorgan's laws state:

1. NOT (a AND b) = (NOT a) OR (NOT b)
2. NOT (a OR b) = (NOT a) AND (NOT b)

We can use these laws to distribute the `not` operator over boolean expressions in Python. For example:

```
if not (age > 0 and age <= 120):
    print("Invalid age")

# can be rewritten as

if age <= 0 or age > 120:
    print("Invalid age")
```

Instead of negating each operator, we used its opposite, eliminating `not` altogether.

## Exercise 2

1. For what values of `input` will this program print `"True"`?

```
    if not input > 5:
        print("True")
```

2. For what values of `absentee_rate` and `overall_mark` will this program print `"You have passed the course."`?

```
    if absentee_rate <= 5 and overall_mark >= 50:
        print("You have passed the course.")
```

3. For what values of `x` will this program print `"True"`?

```
    if x > 1 or x <= 8:
        print("True")
```

4. Eliminate `not` from each of these boolean expressions:

```
    not total <= 2
    not count > 40
    not (value > 20.0 and total != 100.0)
    not (angle > 180 and width == 5)
    not (count == 5 and not (value != 10) or count > 50)
    not (value > 200 or value < 0 and not total == 0)
```

## The None value

We often initialise a number to zero or a string to an empty string before we give it a more meaningful value. Zero and various "empty" values evaluate to `False` in boolean expressions, so we can check whether a variable has a meaningful value like this:

**if (my_variable):** print(my_variable)

Sometimes, however, a zero or an empty string *is* a meaningful value. How can we indicate that a variable isn't set to anything if we *can't* use zero or an empty string? We can set it to `None` instead.

In Python, `None` is a special value which means "nothing". Its type is called `NoneType`, and only one `None` value exists at a time – all the `None` values we use are actually the same object:

```
print(None is None) # True
```

`None` evaluates to `False` in boolean expressions. If we don't care whether our variable is `None` or some other value which is also false, we can just check its value like this:

```
if my_string:
    print("My string is '%s'." % my_string)
```

If, however, we want to distinguish between the case when our variable is `None` and when it is empty (or zero, or some other false value) we need to be more specific:

```
if my_number is not None:
    print(my_number) # could still be zero

if my_string is None:
    print("I haven't got a string at all!")
elif not my_string: # another false value, i.e. an empty string
    print("My string is empty!")
else:
    print("My string is '%s'." % my_string)
```

### Switch statements and dictionary-based dispatch

`if` ladders can get unwieldy if they become very long. Many languages have a control statement called a switch, which tests the value of a single variable and makes a decision on the basis of that value. It is similar to an `if` ladder, but can be a little more readable, and is often optimised to be faster.

Python does not have a switch statement, but we can achieve something similar by using a dictionary. This example will be clearer when we have read more about dictionaries, but all we need to know for now is that a dictionary is a store of key and value pairs – we retrieve a value by its key, the way we would retrieve a list element by its index. Here is how we can rewrite the course code example:

```
DEPARTMENT_NAMES = {
    "CSC": "Computer Science",
    "MAM": "Mathematics and Applied Mathematics",
    "STA": "Statistical Sciences", # Trailing commas like this are allowed in Python!
}

if course_code in DEPARTMENT_NAMES: # this tests whether the variable is one of the dictionary's key
    print("Department: %s" % DEPARTMENT_NAMES[course_code])
else:
    print("Unknown course code: %s" % course_code)
```

We are not limited to storing simple values like strings in the dictionary. In Python, functions can be stored in variables just like any other object, so we can even use this dispatch method to execute completely different statements in response to different values:

```
def reverse(string):
    print("'%s' reversed is '%s'." % (string, string[::-1]))

def capitalise(string):
    print("'%s' capitalised is '%s'." % (string, string.upper()))

ACTIONS = {
    "r": reverse, # use the function name without brackets to refer to the function without calling
    "c": capitalise,
}

my_function = ACTIONS[my_action] # now we retrieve the function
my_function(my_string) # and now we call it
```

## The conditional operator

Python has another way to write a selection in a program – the conditional operator. It can be used within an expression (i.e. it can be evaluated) – in contrast to `if` and `if-else`, which are just statements and not expressions. It is often called the *ternary* operator because it has *three* operands (binary operators have two, and unary operators have one). The syntax is as follows:

*true expression* `if` *condition* `else` *false expression*

For example:

```
result = "Pass" if (score >= 50) else "Fail"
```

This means that if `score` is at least 50, `result` is assigned `"Pass"`, otherwise it is assigned `"Fail"`. This is equivalent to the following `if` statement:

**if (score >= 50):** result = “Pass”

**else:** result = “Fail”

The ternary operator can make simple `if` statements shorter and more legible, but some people may find this code harder to understand. There is no functional or efficiency difference between a normal `if-else` and the ternary operator. You should use the operator sparingly.

### Exercise 3

1. Rewrite the following fragment as an if-ladder (using `elif` statements):

```
if temperature < 0:
    print("Below freezing")
else:
    if temperature < 10:
        print("Very cold")
    else:
        if temperature < 20:
            print(Chilly)
        else:
            if temperature < 30:
                print("Warm")
            else:
                if temperature < 40:
                    print("Hot")
                else:
                    print("Too hot")
```

2. Write a Python program to assign grades to students at the end of the year. The program must do the following:

    (a) Ask for a student number.

    (b) Ask for the student's tutorial mark.

    (c) Ask for the student's test mark.

    (d) Calculate whether the student's average so far is high enough for the student to be permitted to write the examination. If the average (mean) of the tutorial and test marks is lower than 40%, the student should automatically get an F grade, and the program should print the grade and exit without performing the following steps.

    (e) Ask for the student's examination mark.

    (f) Calculate the student's final mark. The tutorial and test marks should count for 25% of the final mark each, and the final examination should count for the remaining 50%.

    (g) Calculate and print the student's grade, according to the following table:

| Weighted final score | Final grade |
|---|---|
| 80 <= mark <= 100 | A |
| 70 <= mark < 80 | B |
| 60 <= mark < 70 | C |
| 50 <= mark < 60 | D |
| mark < 50 | E |

## Answers to exercises

### Answer to exercise 1

1. (a) `if (x > 4)` – valid

    (b) `if x == 2` – valid (brackets are not compulsory)

    (c) `if (y =< 4)` – invalid (=< is not a valid operator; it should be <=)

    (d) `if (y = 5)` – invalid (= is the assignment operator, not a comparison operator)

   (e) `if (3 <= a)` – valid
   (f) `if (1 - 1)` – valid (`1 - 1` evaluates to zero, which is false)
   (g) `if ((1 - 1) <= 0)` – valid
   (h) `if (name == "James")` – valid

2. **The program will print out:**

   is greater
   than 3.

   This happens because the last two print statements are not indented – they are outside the `if` statement, which means that they will always be executed.

3. (a) We don't have to compare variables to boolean values and compare them to `True` explicitly. This will be done implicitly if we just evaluate the variable in the condition of the `if` statement:

```
if a:
    print("a is true")
```

   (b) We set `a` to the same value whether we execute the `if` block or the `else` block, so we can move this line outside the `if` statement and only write it once.

```
if x > 50:
    b += 1
else:
    b -= 1

a = 5
```

## Answer to exercise 2

1. The program will print `"True"` if `input` is less than or equal to 5.
2. The program will print `"You have passed the course."` if `absentee_rate` is less than or equal to `5` and `overall_mark` is greater than or equal to `50`.
3. The program will print `"True"` for any value of `x`.
4.
```
total > 2
count <= 40
value <= 20.0 or total == 100.0
angle <= 180 or width != 5
(count != 5 or value != 10) and count <= 50
value <= 200 and (value >= 0 or total == 0)
```

## Answer to exercise 3

1.
```
if temperature < 0:
    print("Below freezing")
elif temperature < 10:
    print("Very cold")
elif temperature < 20:
    print(Chilly)
elif temperature < 30:
    print("Warm")
elif temperature < 40:
    print("Hot")
else:
    print("Too hot")
```

2. Here is an example program:

```
student_number = input("Please enter a student number: ")
tutorial_mark = float(input("Please enter the student's tutorial mark: "))
test_mark = float(input("Please enter the student's test mark: "))

if (tutorial_mark + test_mark) / 2 < 40:
    grade = "F"
else:
    exam_mark = float(input("Please enter the student's final examination mark: "))
    mark = (tutorial_mark + test_mark + 2 * exam_mark) / 4

    if 80 <= mark <= 100:
        grade = "A"
    elif 70 <= mark < 80:
        grade = "B"
    elif 60 <= mark < 70:
        grade = "C"
    elif 50 <= mark < 60:
        grade = "D"
    else:
        grade = "E"

print "%s's grade is %s." % (student_number, grade)
```

CHAPTER 5

# Collections

We have already encountered some simple Python types like numbers, strings and booleans. Now we will see how we can group multiple values together in a *collection* – like a *list* of numbers, or a *dictionary* which we can use to store and retrieve key-value pairs. Many useful collections are built-in types in Python, and we will encounter them quite often.

## Lists

The Python list type is called `list`. It is a type of sequence – we can use it to store multiple values, and access them sequentially, by their position, or *index*, in the list. We define a list *literal* by putting a comma-separated list of values inside square brackets (`[` and `]`):

```
# a list of strings
animals = ['cat', 'dog', 'fish', 'bison']

# a list of integers
numbers = [1, 7, 34, 20, 12]

# an empty list
my_list = []

# a list of variables we defined somewhere else
things = [
    one_variable,
    another_variable,
    third_variable, # this trailing comma is legal in Python
]
```

As you can see, we have used plural nouns to name most of our list variables. This is a common convention, and it's useful to follow it in most cases.

To refer to an element in the list, we use the list identifier followed by the index inside square brackets. Indices are integers which *start from zero*:

```
print(animals[0]) # cat
print(numbers[1]) # 7

# This will give us an error, because the list only has four elements
print(animals[6])
```

We can also count from the end:

```
print(animals[-1]) # the last element -- bison
print(numbers[-2]) # the second-last element -- 20
```

We can extract a subset of a list, which will itself be a list, using a *slice*. This uses almost the same syntax as accessing a single element, but instead of specifying a single index between the square brackets we need to specify an upper and lower bound. Note that our sublist will *include* the element at the lower bound, but *exclude* the element at the upper bound:

```
print(animals[1:3]) # ['dog', 'fish']
print(animals[1:-1]) # ['dog', 'fish']
```

If one of the bounds is one of the ends of the list, we can leave it out. A slice with neither bound specified gives us a copy of the list:

```
print(animals[2:]) # ['fish', 'bison']
print(animals[:2]) # ['cat', 'dog']
print(animals[:]) # a copy of the whole list
```

We can even include a third parameter to specify the step size:

```
print(animals[::2]) # ['cat', 'fish']
```

Lists are mutable – we can modify elements, add elements to them or remove elements from them. A list will change size dynamically when we add or remove elements – we don't have to manage this ourselves:

```
# assign a new value to an existing element
animals[3] = "hamster"

# add a new element to the end of the list
animals.append("squirrel")

# remove an element by its index
del animals[2]
```

Because lists are mutable, we can *modify* a list variable without assigning the variable a completely new value. Remember that if we assign the same `list` value to two variables, any in-place changes that we make while referring to the list by one variable name will also be reflected when we access the list through the other variable name:

```
animals = ['cat', 'dog', 'goldfish', 'canary']
pets = animals # now both variables refer to the same list object

animals.append('aardvark')
print(pets) # pets is still the same list as animals

animals = ['rat', 'gerbil', 'hamster'] # now we assign a new list value to animals
print(pets) # pets still refers to the old list

pets = animals[:] # assign a *copy* of animals to pets
animals.append('aardvark')
print(pets) # pets remains unchanged, because it refers to a copy, not the original list
```

We can mix the types of values that we store in a list:

```
my_list = ['cat', 12, 35.8]
```

How do we check whether a list contains a particular value? We use `in` or `not in`, the membership operators:

```
numbers = [34, 67, 12, 29]
my_number = 67
```

```
if number in numbers:
    print("%d is in the list!" % number)

my_number = 90
if number not in numbers:
    print("%d is not in the list!" % number)
```

**Note:** `in` and `not in` fall between the logical operators (`and`, `or` and `not`) and the identity operators (`is` and `is not`) in the order of precedence.

## List methods and functions

There are many built-in functions which we can use on lists and other sequences:

```
# the length of a list
len(animals)

# the sum of a list of numbers
sum(numbers)

# are any of these values true?
any([1,0,1,0,1])

# are all of these values true?
all([1,0,1,0,1])
```

List objects also have useful methods which we can call:

```
numbers = [1, 2, 3, 4, 5]

# we already saw how to add an element to the end
numbers.append(5)

# count how many times a value appears in the list
numbers.count(5)

# append several values at once to the end
numbers.extend([56, 2, 12])

# find the index of a value
numbers.index(3)
# if the value appears more than once, we will get the index of the first one
numbers.index(2)
# if the value is not in the list, we will get a ValueError!
numbers.index(42)

# insert a value at a particular index
numbers.insert(0, 45) # insert 45 at the beginning of the list

# remove an element by its index and assign it to a variable
my_number = numbers.pop(0)

# remove an element by its value
numbers.remove(12)
# if the value appears more than once, only the first one will be removed
```

```
numbers.remove(5)
```

If we want to sort or reverse a list, we can either call a method on the list to modify it *in-place*, or use a function to return a modified copy of the list while leaving the original list untouched:

```
numbers = [3, 2, 4, 1]

# these return a modified copy, which we can print
print(sorted(numbers))
print(list(reversed(numbers)))

# the original list is unmodified
print(numbers)

# now we can modify it in place
numbers.sort()
numbers.reverse()

print(numbers)
```

The `reversed` function actually returns a generator, not a list (we will look at generators in the next chapter), so we have to convert it to a list before we can print the contents. To do this, we call the `list` type like a function, just like we would call `int` or `float` to convert numbers. We can also use `list` as another way to make a copy of a list:

```
animals = ['cat', 'dog', 'goldfish', 'canary']
pets = list(animals)

animals.sort()
pets.append('gerbil')

print(animals)
print(pets)
```

## Using arithmetic operators with lists

Some of the arithmetic operators we have used on numbers before can also be used on lists, but the effect may not always be what we expect:

```
# we can concatenate two lists by adding them
print([1, 2, 3] + [4, 5, 6])

# we can concatenate a list with itself by multiplying it by an integer
print([1, 2, 3] * 3)

# not all arithmetic operators can be used on lists -- this will give us an error!
print([1, 2, 3] - [2, 3])
```

## Lists vs arrays

Many other languages don't have a built-in type which behaves like Python's list. You can use an implementation from a library, or write your own, but often programmers who use these languages use *arrays* instead. Arrays are simpler, more low-level data structures, which don't have all the functionality of a list. Here are some major differences between lists and arrays:

- An array has a fixed size which you specify when you create it. If you need to add or remove elements, you have to make a new array.

- If the language is statically typed, you also have to specify a single type for the values which you are going to put in the array when you create it.
- In languages which have *primitive types*, arrays are usually not objects, so they don't have any methods – they are just containers.

Arrays are less easy to use in many ways, but they also have some advantages: because they are so simple, and there are so many restrictions on what you can do with them, the computer can handle them very efficiently. That means that it is often much faster to use an array than to use an object which behaves like a list. A lot of programmers use them when it is important for their programs to be fast.

Python has a built-in `array` type. It's not quite as restricting as an array in C or Java – you have to specify a type for the contents of the array, and you can only use it to store numeric values, but you can resize it dynamically, like a list. You will probably never need to use it.

### Exercise 1

1. Create a list `a` which contains the first three odd positive integers and a list `b` which contains the first three even positive integers.
2. Create a new list `c` which combines the numbers from both lists (order is unimportant).
3. Create a new list `d` which is a sorted copy of `c`, leaving `c` unchanged.
4. Reverse `d` in-place.
5. Set the fourth element of `c` to `42`.
6. Append `10` to the end of `d`.
7. Append `7`, `8` and `9` to the end of `c`.
8. Print the first three elements of `c`.
9. Print the last element of `d` without using its length.
10. Print the length of `d`.

## Tuples

Python has another sequence type which is called `tuple`. Tuples are similar to lists in many ways, but they are immutable. We define a tuple *literal* by putting a comma-separated list of values inside round brackets ( ( and ) ):

```
WEEKDAYS = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')
```

We can use tuples in much the same way as we use lists, except that we can't modify them:

```
animals = ('cat', 'dog', 'fish')

# an empty tuple
my_tuple = ()

# we can access a single element
print(animals[0])

# we can get a slice
print(animals[1:]) # note that our slice will be a new tuple, not a list

# we can count values or look up an index
```

```
animals.count('cat')
animals.index('cat')

# ... but this is not allowed:
animals.append('canary')
animal[1] = 'gerbil'
```

What are tuples good for? We can use them to create a sequence of values that we don't want to modify. For example, the list of weekday names is never going to change. If we store it in a tuple, we can make sure it is never modified accidentally in an unexpected place:

```
# Here's what can happen if we put our weekdays in a mutable list

WEEKDAYS = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']

def print_funny_weekday_list(weekdays):
    weekdays[5] = 'Caturday' # this is going to modify the original list!
    print(weekdays)

print_funny_weekday_list(WEEKDAYS)

print(WEEKDAYS) # oops
```

We have already been using tuples when inserting multiple values into a formatted string:

```
print("%d %d %d" % (1, 2, 3))
```

How do we define a tuple with a single element? We can't just put round brackets around a value, because round brackets are also used to change the order of precedence in an expression – a value in brackets is just another way of writing the value:

```
print(3)
print((3)) # this is still just 3
```

To let Python know that we want to create a tuple, we have to add a trailing comma:

```
print((3,))
```

### Exercise 2

1. Create a tuple `a` which contains the first four positive integers and a tuple `b` which contains the next four positive integers.
2. Create a tuple `c` which combines all the numbers from `a` and `b` in any order.
3. Create a tuple `d` which is a sorted copy of `c`.
4. Print the third element of `d`.
5. Print the last three elements of `d` without using its length.
6. Print the length of `d`.

## Sets

The Python set type is called `set`. A set is a collection of *unique elements*. If we add multiple copies of the same element to a set, the duplicates will be eliminated, and we will be left with one of each element. To define a set literal, we put a comma-separated list of values inside curly brackets (`{` and `}`):

```
animals = {'cat', 'dog', 'goldfish', 'canary', 'cat'}
print(animals) # the set will only contain one cat
```

We can perform various set operations on sets:

```
even_numbers = {2, 4, 6, 8, 10}
big_numbers = {6, 7, 8, 9, 10}

# subtraction: big numbers which are not even
print(big_numbers - even_numbers)

# union: numbers which are big or even
print(big_numbers | even_numbers)

# intersection: numbers which are big and even
print(big_numbers & even_numbers)

# numbers which are big or even but not both
print(big_numbers ^ even_numbers)
```

It is important to note that unlike lists and tuples sets are *not ordered*. When we print a set, the order of the elements will be random. If we want to process the contents of a set in a particular order, we will first need to convert it to a list or tuple and sort it:

```
print(animals)
print(sorted(animals))
```

The `sorted` function returns a `list` object.

How do we make an empty set? We have to use the `set` function. Dictionaries, which we will discuss in the next section, used curly brackets before sets adopted them, so an empty set of curly brackets is actually an empty dictionary:

```
# this is an empty dictionary
a = {}

# this is how we make an empty set
b = set()
```

We can use the `list`, `tuple`, `dict` and even `int`, `float` or `str` functions in the same way – they all have sensible defaults – but we will probably seldom find a reason to do so.

### Exercise 3

1. Create a set `a` which contains the first four positive integers and a set `b` which contains the first four odd positive integers.
2. Create a set `c` which combines all the numbers which are in `a` or `b` (or both).
3. Create a set `d` which contains all the numbers in `a` but not in `b`.
4. Create a set `e` which contains all the numbers in `b` but not in `a`.
5. Create a set `f` which contains all the numbers which are both in `a` and in `b`.
6. Create a set `g` which contains all the numbers which are either in `a` or in `b` but not in both.
7. Print the number of elements in `c`.

## Ranges

`range` is another kind of immutable sequence type. It is very specialised – we use it to create ranges of integers. Ranges are also *generators*. We will find out more about generators in the next chapter, but for now we just need to know that the numbers in the range are generated one at a time as they are needed, and not all at once. In the examples below, we convert each range to a list so that all the numbers are generated and we can print them out:

```
# print the integers from 0 to 9
print(list(range(10)))

# print the integers from 1 to 10
print(list(range(1, 11)))

# print the odd integers from 1 to 10
print(list(range(1, 11, 2)))
```

We create a range by calling the `range` function. As you can see, if we pass a single parameter to the `range` function, it is used as the upper bound. If we use two parameters, the first is the lower bound and the second is the upper bound. If we use three, the third parameter is the step size. The default lower bound is zero, and the default step size is one. Note that the range *includes* the lower bound and *excludes* the upper bound.

### Exercise 4

1. Create a range `a` which starts from `0` and goes on for 20 numbers.
2. Create a range `b` which starts from `3` and ends on `12`.
3. Create a range `c` which contains every third integer starting from `2` and ending at `50`.

## Dictionaries

The Python dictionary type is called `dict`. We can use a dictionary to store key-value pairs. To define a dictionary literal, we put a comma-separated list of key-value pairs between curly brackets. We use a colon to separate each key from its value. We access values in the dictionary in much the same way as list or tuple elements, but we use keys instead of indices:

```
marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }

personal_details = {
    "name": "Jane Doe",
    "age": 38, # trailing comma is legal
}

print(marbles["green"])
print(personal_details["name"])

# This will give us an error, because there is no such key in the dictionary
print(marbles["blue"])

# modify a value
marbles["red"] += 3
personal_details["name"] = "Jane Q. Doe"
```

The keys of a dictionary don't have to be strings – they can be *any immutable type*, including numbers and even tuples. We can mix different types of keys and different types of values in one dictionary. Keys are unique – if we repeat a key, we will overwrite the old value with the new value. When we store a value in a dictionary, the key doesn't have to exist – it will be created automatically:

```
battleship_guesses = {
    (3, 4): False,
    (2, 6): True,
    (2, 5): True,
}

surnames = {} # this is an empty dictionary
surnames["John"] = "Smith"
surnames["John"] = "Doe"
print(surnames) # we overwrote the older surname

marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }
marbles["blue"] = 30 # this will work
marbles["purple"] += 2 # this will fail -- the increment operator needs an existing value to modify!
```

Like sets, dictionaries are not ordered – if we print a dictionary, the order will be random.

Here are some commonly used methods of dictionary objects:

```
marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }

# Get a value by its key, or None if it doesn't exist
marbles.get("orange")
# We can specify a different default
marbles.get("orange", 0)

# Add several items to the dictionary at once
marbles.update({"orange": 34, "blue": 23, "purple": 36})

# All the keys in the dictionary
marbles.keys()
# All the values in the dictionary
marbles.values()
# All the items in the dictionary
marbles.items()
```

The last three methods return special sequence types which are read-only *views* of various properties of the dictionary. We cannot edit them directly, but they will be updated when we modify the dictionary. We most often access these properties because we want to iterate over them (something we will discuss in the next chapter), but we can also convert them to other sequence types if we need to.

We can check if a key is in the dictionary using `in` and `not in`:

```
print("purple" in marbles)
print("white" not in marbles)
```

We can also check if a value is in the dictionary using `in` in conjunction with the `values` method:

```
print("Smith" in surnames.values())
```

You should avoid using `mykey in mydict.keys()` to check for key membership, however, because it's less efficient than `mykey in mydict`.

**Note:** in Python 2, `keys`, `values` and `items` return list copies of these sequences, `iterkeys`, `itervalues` and `iteritems` return iterator objects, and `viewkeys`, `viewvalues` and `viewitems` return the view objects which are the default in Python 3 (but these are only available in Python 2.7 and above). In Python 2 you should *really* not use `mykey in mydict.keys()` to check for key membership – if you do, you will be searching the entire list of keys sequentially, which is much slower than a direct dictionary lookup.

### Exercise 5

1. Create a dict `directory` which stores telephone numbers (as string values), and populate it with these key-value pairs:

| Name | Telephone number |
|---|---|
| Jane Doe | +27 555 5367 |
| John Smith | +27 555 6254 |
| Bob Stone | +27 555 5689 |

2. Change Jane's number to *+27 555 1024*
3. Add a new entry for a person called *Anna Cooper* with the phone number *+27 555 3237*
4. Print Bob's number.
5. Print Bob's number in such a way that `None` would be printed if Bob's name were not in the dictionary.
6. Print all the keys. The format is unimportant, as long as they're all visible.
7. Print all the values.

## Converting between collection types

### Implicit conversions

If we try to iterate over a collection in a `for` loop (something we will discuss in the next chapter), Python will try to convert it into something that we can iterate over if it knows how to. For example, the dictionary views we saw above are not actually iterators, but Python knows how to make them into iterators – so we can use them in a `for` loop without having to convert them ourselves.

Sometimes the iterator we get by default may not be what we expected – if we iterate over a dictionary in a `for` loop, we will iterate over the *keys*. If what we actually want to do is iterate over the values, or key and value pairs, we will have to specify that ourselves by using the dictionary's `values` or `items` view instead.

### Explicit conversions

We can convert between the different sequence types quite easily by using the type functions to *cast* sequences to the desired types – just like we would use `float` and `int` to convert numbers:

```
animals = ['cat', 'dog', 'goldfish', 'canary', 'cat']

animals_set = set(animals)
animals_unique_list = list(animals_set)
animals_unique_tuple = tuple(animals_unique_list)
```

We have to be more careful when converting a dictionary to a sequence: do we want to use the keys, the values or pairs of keys and values?

```
marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }

colours = list(marbles) # the keys will be used by default
counts = tuple(marbles.values()) # but we can use a view to get the values
marbles_set = set(marbles.items()) # or the key-value pairs
```

If we convert the key-value pairs of a dictionary to a sequence, each pair will be converted to a tuple containing the key followed by the value.

We can also convert a sequence to a dictionary, but only if it's a sequence of *pairs* – each pair must itself be a sequence with two values:

```
# Python doesn't know how to convert this into a dictionary
dict([1, 2, 3, 4])

# but this will work
dict([(1, 2), (3, 4)])
```

We will revisit conversions in the next chapter, when we learn about *comprehensions* – an efficient syntax for filtering sequences or dictionaries. By using the right kind of comprehension, we can filter a collection and convert it to a different type of collection at the same time.

## Another look at strings

Strings are also a kind of sequence type – they are sequences of characters, and share some properties with other sequences. For example, we can find the length of a string or the index of a character in the string, and we can access individual elements of strings or slices:

```
s = "abracadabra"

print(len(s))
print(s.index("a"))

print(s[0])
print(s[3:5])
```

Remember that strings are immutable – modifying characters in-place isn't allowed:

```
# this will give us an error
s[0] = "b"
```

The membership operator has special behaviour when applied to strings: we can use it to determine if a string contains a single character as an element, but we can also use it to check if a string contains a substring:

```
print('a' in 'abcd') # True
print('ab' in 'abcd') # also True

# this doesn't work for lists
print(['a', 'b'] in ['a', 'b', 'c', 'd']) # False
```

We can easily convert a string to a list of characters:

```
abc_list = list("abracadabra")
```

What if we want to convert a list of characters into a string? Using the `str` function on the list will just give us a printable string of the list, including commas, quotes and brackets. To join a sequence of characters (or longer strings) together into a single string, we have to use `join`.

`join` is not a function or a sequence method – it's a *string* method which takes a sequence of strings as a parameter. When we call a string's `join` method, we are using that string to glue the strings in the sequence together. For example, to join a list of single characters into a string, with no spaces between them, we call the `join` method on the *empty string*:

```
l = ['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a']

s = "".join(l)
print(s)
```

We can use any string we like to join a sequence of strings together:

```
animals = ('cat', 'dog', 'fish')

# a space-separated list
print(" ".join(animals))

# a comma-separated list
print(",".join(animals))

# a comma-separated list with spaces
print(", ".join(animals))
```

The opposite of *joining* is *splitting*. We can split up a string into a list of strings by using the `split` method. If called without any parameters, `split` divides up a string into words, using any number of consecutive whitespace characters as a delimiter. We can use additional parameters to specify a different delimiter as well as a limit on the maximum number of splits to perform:

```
print("cat     dog fish\n".split())
print("cat|dog|fish".split("|"))
print("cat, dog, fish".split(", "))
print("cat, dog, fish".split(", ", 1))
```

## Exercise 6

1. Convert a list which contains the numbers `1`, `1`, `2`, `3` and `3`, and convert it to a tuple `a`.
2. Convert `a` to a list `b`. Print its length.
3. Convert `b` to a set `c`. Print its length.
4. Convert `c` to a list `d`. Print its length.
5. Create a range which starts at `1` and ends at `10`. Convert it to a list `e`.
6. Create the `directory` dict from the previous example. Create a list `t` which contains all the key-value pairs from the dictionary as tuples.
7. Create a list `v` of all the values in the dictionary.
8. Create a list `k` of all the keys in he dictionary.
9. Create a string `s` which contains the word `"antidisestablishmentarianism"`. Use the `sorted` function on it. What is the output type? Concatenate the letters in the output to a string `s2`.
10. Split the string `"the quick brown fox jumped over the lazy dog"` into a list `w` of individual words.

## Two-dimensional sequences

Most of the sequences we have seen so far have been one-dimensional: each sequence is a row of elements. What if we want to use a sequence to represent a two-dimensional data structure, which has both rows and columns? The easiest way to do this is to make a sequence in which each element is also a sequence. For example, we can create a list of lists:

```
my_table = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [10, 11, 12],
]
```

The outer list has four elements, and each of these elements is a list with three elements (which are numbers). To access one of these numbers, we need to use two indices – one for the outer list, and one for the inner list:

```
print(my_table[0][0])

# lists are mutable, so we can do this
my_table[0][0] = 42
```

We have already seen an example of this in the previous chapter, when we created a list of tuples to convert into a dict.

When we use a two-dimensional sequence to represent tabular data, each inner sequence will have the same length, because a table is rectangular – but nothing is stopping us from constructing two-dimensional sequences which don't have this property:

```
my_2d_list = [
    [0],
    [1, 2, 3, 4],
    [5, 6],
]
```

We can also make a three-dimensional sequence by making a list of lists of lists:

```
my_3d_list = [
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]],
]

print(my_3d_list[0][0][0])
```

Of course we can also make a list of lists of lists of lists and so forth – we can nest lists as many times as we like.

If we wanted to make a two-dimensional list to represent a weekly timetable, we could either have days as the outer list and time slots as the inner list or the other way around – we would have to remember which range we picked to be the rows and which the columns.

Suppose that we wanted to initialise the timetable with an empty string in each time slot – let us say that we have 24 hour-long time slots in each day. That's seven lists of 24 elements each – quite long and unwieldy to define using literals, the way we defined the smaller lists in the examples above!

This brings us to a common pitfall. You may recall from a previous section that we can use the multiplication operator on lists – this can be a convenient way to construct a long list in which all the elements are the same:

```
my_long_list = [0] * 100 # a long list of zeros
print(my_long_list)
```

You might think of using this method to construct our timetable. We can certainly use it to create a list of empty strings to represent a day:

```
day = [""] * 24
print(day)
```

But what happens if we repeat a day seven times to make a week?

```
timetable = day * 7
print(timetable)
```

Everything looks fine so far, so what's the problem? Well, let's see what happens when we try to schedule a meeting for Monday afternoon:

```
timetable[0][15] = "meeting with Jane"
print(timetable)
```

Every day has the same afternoon meeting! How did that happen? When we multiplied our day list by seven, we filled our timetable with *the same list object*, repeated seven times. All the elements in our timetable are the same day, so no matter which one we modify we modify all of them at once.

Why didn't this matter when we made the day list by multiplying the same empty string 24 times? Because strings are immutable. We can only change the values of the strings in the day list by assigning them new values – we can't modify them in-place, so it doesn't matter that they all start off as the same string object. But because we *can* modify lists in-place, it does matter that all our day lists are the same list. What we actually want is seven *copies* of a day list in our timetable:

```
timetable = [[""] * 24 for day in range(7)]
```

Here we construct the timetable with a list comprehension instead. We will learn more about comprehensions in the next chapter – for now, it is important for us to know that this method creates a *new* list of empty strings for each day, unlike the multiplication operator.

### Exercise 7

1. Create a list `a` which contains three tuples. The first tuple should contain a single element, the second two elements and the third three elements.
2. Print the second element of the second element of `a`.
3. Create a list `b` which contains four lists, each of which contains four elements.
4. Print the last two elements of the first element of `b`.

## Answers to exercises

### Answer to exercise 1

```
a = [1, 3, 5]
b = [2, 4, 6]

c = a + b

d = sorted(c)
d.reverse()
```

```
c[3] = 42
d.append(10)
d.extend([7, 8, 9])

print(c[:2])
print(d[-1])
print(len(d))
```

### Answer to exercise 2

```
a = (1, 2, 3, 4)
b = (5, 6, 7, 8)

c = a + b
d = sorted(c)

print(d[3])
print(d[-3:])
print(len(d))
```

### Answer to exercise 3

```
a = {1, 2, 3, 4}
b = {1, 3, 5, 7}

c = a | b
d = a - b
e = b - a
f = a & b
g = a ^ b

print(len(c))
```

### Answer to exercise 4

```
a = range(20)
b = range(3, 13)
c = range(2, 51, 3)
```

### Answer to exercise 5

```
directory = {
    "Jane Doe": "+27 555 5367",
    "John Smith": "+27 555 6254",
    "Bob Stone": "+27 555 5689",
}

directory["Jane Doe"] = "+27 555 1024"
directory["Anna Cooper"] = "+27 555 3237"

print(directory["Bob Stone"])
print(directory.get("Bob Stone", None))

print(directory.keys())
print(directory.values())
```

### Answer to exercise 6

```
a = tuple([1, 1, 2, 3, 3])

b = list(a)
print(len(b))

c = set(b)
print(len(c))

d = list(c)
print(len(d))

e = list(range(1, 11))

directory = {
    "Jane Doe": "+27 555 5367",
    "John Smith": "+27 555 6254",
    "Bob Stone": "+27 555 5689",
}

t = list(directory.items())
v = list(directory.values())
k = list(directory)

s = "antidisestablishmentarianism"
s2 = "".join(sorted(s))

w = "the quick brown fox jumped over the lazy dog".split()
```

### Answer to exercise 7

Here is a code example:

```
a = [
    (1,),
    (2, 2),
    (3, 3, 3),
]

print(a[1][1])

b = [
    list(range(10)),
    list(range(10, 20)),
    list(range(20, 30)),
    list(range(30, 40)),
]

print(b[0][1:-1])
```

CHAPTER 6

# Loop control statements

## Introduction

In this chapter, you will learn how to make the computer execute a group of statements over and over as long as certain criterion holds. The group of statements being executed repeatedly is called a loop. There are two loop statements in Python: `for` and `while`. We will discuss the difference between these statements later in the chapter, but first let us look at an example of a loop in the real world.

A petrol attendant performs the following actions when serving a customer:

1. greet customer
2. ask for required type of petrol and amount
3. ask whether customer needs other services
4. ask for required amount of money
5. give money to cashier
6. wait for change and receipt
7. give change and receipt to customer
8. say thank you and goodbye

A petrol attendant performs these steps for each customer, but he does not follow them when there is no customer to serve. He also only performs them when it is his shift. If we were to write a computer program to simulate this behaviour, it would not be enough just to provide the steps and ask the computer to repeat them over and over. We would also need to tell it when to stop executing them.

There are two major kinds of programming loops: counting loops and event-controlled loops.

In a counting loop, the computer knows at the beginning of the loop execution how many times it needs to execute the loop. In Python, this kind of loop is defined with the `for` statement, which executes the loop body *for* every item in some list.

In an event-controlled loop, the computer stops the loop execution when a condition is no longer true. In Python, you can use the `while` statement for this – it executes the loop body *while* the condition is true. The `while` statement checks the condition *before* performing each iteration of the loop. Some languages also have a loop statement which performs the check *after* each iteration, so that the loop is always executed at least once. Python has no such construct, but we will see later how you can simulate one.

Counting loops are actually subset of event-control loop - the loop is repeated until the required number of iterations is reached.

If you wanted to get from Cape Town to Camps Bay, what loop algorithm would you use? If you started by putting your car on the road to Camps Bay, you could:

- drive for exactly 15 minutes. After 15 minutes, stop the car and get out.
- drive for exactly 8km. After 8km, stop the car and get out.
- drive as long as you are not in Camps Bay. When you arrive, stop the car and get out.

The first two algorithms are based on counting – the first counts time, and the second counts distance. Neither of these algorithms guarantees that you will arrive in Camps Bay. In the first case, you might hit heavy traffic or none at all, and either fall short of or overshoot your desired destination. In the second case, you might find a detour and end up nowhere near Camps Bay.

The third algorithm is event-controlled. You carry on driving as long as you are not at the beach. The condition you keep checking is *am I at the beach yet?*.

Many real-life activities are event-controlled. For example, you drink as long as you are thirsty. You read the newspaper as long as you are interested. Some activities are based on multiple events – for example, a worker works as long as there is work to do and the time is not 5pm.

## The `while` statement

Python's event-controlled loop statement is the `while` statement. You should use it when you don't know beforehand how many times you will have to execute the body of the loop. The while-body keeps repeating as long as the condition is true. Here's a flow control diagram for the while statement:

blockdiag-f3a27d8f6b0baede43d98ee8746bc4ed43aa58c8.png

The loop consists of three important parts: the *initialisation*, the *condition*, and the *update*. In the initialisation step, you set up the variable which you're going to use in the condition. In the condition step, you perform a test on the variable to see whether you should terminate the loop or execute the body another time. Then, after each successfully completed execution of the loop body, you update your variable.

Note that the condition is checked before the loop body is executed for the first time – if the condition is false at the start, the loop body will never be executed at all.

Here is a simple Python example which adds the first ten integers together:

```python
total = 0
i = 1

while i <=10:
    total += i
    i += 1
```

The variable used in the loop condition is the number `i`, which you use to count the integers from `1` to `10`. First you *initialise* this number to `1`. In the *condition*, you check whether `i` is less than or equal to `10`, and if this is true you execute the loop body. Then, at the end of the loop body, you *update* `i` by incrementing it by `1`.

It is very important that you increment `i` at the end. If you did not, `i` would always be equal to `1`, the condition would always be true, and your program would never terminate – we call this an infinite loop. Whenever you write a `while` loop, make sure that the variable you use in your condition is updated inside the loop body!

Here are a few common errors which might result in an infinite loop:

```
x = 0
while x < 3:
    y += 1 # wrong variable updated

product = 1
count = 1

while count <= 10:
    product *= count
    # forgot to update count

x = 0
while x < 5:
    print(x)
x += 1 # update statement is indented one level too little, so it's outside the loop body

x = 0
while x != 5:
    print(x)
    x += 2 # x will never equal 5, because we are counting in even numbers!
```

You might be wondering why the Python interpreter cannot catch infinite loops. This is known as the halting problem. It is impossible for a computer to detect all possible infinite loops in another program. It is up to the programmer to avoid infinite loops.

In many of the examples above, we are counting to a predetermined number, so it would really be more appropriate for us to use a `for` loop (which will be introduced in the next section) – that is the loop structure which is more commonly used for counting loops. Here is a more realistic example:

```
# numbers is a list of numbers -- we don't know what the numbers are!

total = 0
i = 0

while i < len(numbers) and total < 100:
    total += numbers[i]
    i +=1
```

Here we add up numbers from a list until the total reaches 100. We don't know how many times we will have to execute the loop, because we don't know the values of the numbers. Note that we might reach the end of the list of numbers before the total reaches 100 – if we try to access an element beyond the end of the list we will get an error, so we should add a check to make sure that this doesn't happen.

### Exercise 1

1. Write a program which uses a `while` loop to sum the squares of integers (starting from `1`) until the total exceeds 200. Print the final total and the last number to be squared and added.
2. Write a program which keeps prompting the user to guess a word. The user is allowed up to ten guesses – write your code in such a way that the secret word and the number of allowed guesses are easy to change. Print messages to give the user feedback.

## The `for` statement

Python's other loop statement is the `for` statement. You should use it when you need to do something for some predefined number of steps. Before we look at Python's `for` loop syntax, we will briefly look at the way *for* loops work in other languages.

Here is an example of a *for* loop in Java:

```
for (int count = 1; count <= 8; count++) {
    System.out.println(count);
}
```

You can see that this kind of *for* loop has a lot in common with a *while* loop – in fact, you could say that it's just a special case of a *while* loop. The initialisation step, the condition and the update step are all defined in the section in parentheses on the first line.

*for* loops are often used to perform an operation on every element of some kind of sequence. If you wanted to iterate over a list using the classic-style *for* loop, you would have to count from zero to the end of the list, and then access each list element by its index.

In Python, `for` loops make this use case simple and easy by allowing you to iterate over sequences directly. Here is an example of a `for` statement which counts from 1 to 8:

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

As we saw in the previous chapter, `range` is an immutable sequence type used for ranges of integers – in this case, the range is counting from `1` to `8`. The `for` loop will step through each of the numbers in turn, performing the print action for each one. When the end of the range is reached, the `for` loop will exit.

You can use `for` to iterate over other kinds of sequences too. You can iterate over a list of strings like this:

```
pets = ["cat", "dog", "budgie"]

for pet in pets:
    print(pet)
```

At each iteration of the loop, the next element of the list `pets` is assigned to the variable `pet`, which you can then access inside the loop body. The example above is functionally identical to this:

```
for i in range(len(pets)): # i will iterate over 0, 1 and 2
    pet = pets[i]
    print(pet)
```

That is similar to the way `for` loops are written in, for example, Java. You should avoid doing this, as it's more difficult to read, and unnecessarily complex. If for some reason you need the index inside the loop as well as the list element itself, you can use the `enumerate` function to number the elements:

```
for i, pet in enumerate(pets):
    pets[i] = pet.upper() # rewrite the list in all caps
```

Like `range`, `enumerate` also returns an iterator – each item it generates is a tuple in which the first value is the index of the element (starting at zero) and the second is the element itself. In the loop above, at each iteration the value of the index is assigned to the variable `i`, and the element is assigned to the variable `pet`, as before.

Why couldn't we just write `pet = pet.upper()`? That would just assign a new value to the variable `pet` inside the loop, without changing the original list.

This brings us to a common `for` loop pitfall: modifying a list while you're iterating over it. The example above only modifies elements in-place, and doesn't change their order around, but you can cause all kinds of errors and unintended behaviour if you insert or delete list elements in the middle of iteration:

```
numbers = [1, 2, 2, 3]

for i, num in enumerate(numbers):
    if num == 2:
        del numbers[i]

print(numbers) # oops -- we missed one, because we shifted the elements around while we were iterati
```

Sometimes you can avoid this by iterating over a *copy* of the list instead, but it won't help you in this case – as you delete elements from the original list, it will shrink, so the indices from the unmodified list copy will soon exceed the length of the modified list and you will get an error. In general, if you want to select a subset of elements from a list on the basis of some criterion, you should use a *list comprehension* instead. We will look at them at the end of this chapter.

### Exercise 2

1. Write a program which sums the integers from 1 to 10 using a `for` loop (and prints the total at the end).
2. Can you think of a way to do this without using a loop?
3. Write a program which finds the factorial of a given number. E.g. 3 factorial, or **3!** is equal to **3 x 2 x 1**; **5!** is equal to **5 x 4 x 3 x 2 x 1**, etc.. Your program should only contain a single loop.
4. Write a program which prompts the user for 10 floating-point numbers and calculates their sum, product and average. Your program should only contain a single loop.
5. Rewrite the previous program so that it has two loops – one which collects and stores the numbers, and one which processes them.

## Nested loops

We saw in the previous chapter that we can create multi-dimensional sequences – sequences in which each element is another sequence. How do we iterate over all the values of a multi-dimensional sequence? We need to use loops inside other loops. When we do this, we say that we are *nesting* loops.

Consider the timetable example from the previous chapter – let us say that the timetable contains seven days, and each day contains 24 time slots. Each time slot is a string, which is empty if there is nothing scheduled for that slot. How can we iterate over all the time slots and print out all our scheduled events?

```
# first let's define weekday names
WEEKDAYS = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')

# now we iterate over each day in the timetable
for day in timetable:
    # and over each timeslot in each day
    for i, event in enumerate(day):
        if event: # if the slot is not an empty string
            print("%s at %02d:00 -- %s" % (WEEKDAYS[day], i, event))
```

Note that we have two `for` loops – the inner loop will be executed once for every step in the outer loop's iteration. Also note that we are using the `enumerate` function when iterating over the days – because we need both the index of each time slot (so that we can print the hour) and the contents of that slot.

You may have noticed that we look up the name of the weekday once for every iteration of the inner loop – but the name only changes once for every iteration of the outer loop. We can make our loop a little more efficient by moving this lookup out of the inner loop, so that we only perform it seven times and not 168 times!

```
for day in timetable:
    day_name = WEEKDAYS[day]
    for i, event in enumerate(day):
        if event:
            print("%s at %02d:00 -- %s" % (day_name, i, event))
```

This doesn't make much difference when you are looking up a value in a short tuple, but it could make a big difference if it were an expensive, time-consuming calculation and you were iterating over hundreds or thousands of values.

### Exercise 3

1. Write a program which uses a nested `for` loop to populate a three-dimensional list representing a calendar: the top-level list should contain a sub-list for each month, and each month should contain four weeks. Each week should be an empty list.
2. Modify your code to make it easier to access a month in the calendar by a human-readable month name, and each week by a name which is numbered starting from 1. Add an event (in the form of a string description) to the second week in July.

## Iterables, iterators and generators

In Python, any type which can be iterated over with a `for` loop is an *iterable*. Lists, tuples, strings and dicts are all commonly used iterable types. Iterating over a list or a tuple simply means processing each value in turn.

Sometimes we use a sequence to store a series of values which don't follow any particular pattern: each value is unpredictable, and can't be calculated on the fly. In cases like this, we have no choice but to store each value in a list or tuple. If the list is very large, this can use up a lot of memory.

What if the values in our sequence *do* follow a pattern, and *can* be calculated on the fly? We can save a lot of memory by calculating values only when we need them, instead of calculating them all up-front: instead of storing a big list, we can store only the information we need for the calculation.

Python has a lot of built-in iterable types that generate values on demand – they are often referred to as *generators*. We have already seen some examples, like `range` and `enumerate`. You can mostly treat a generator just like any other sequence if you only need to access its elements one at a time – for example, if you use it in a `for` loop:

```
# These two loops will do exactly the same thing:

for i in (1, 2, 3, 4, 5):
    print(i)

for i in range(1, 6):
    print(i)
```

You may notice a difference if you try to print out the generator's contents – by default all you will get is Python's standard string representation of the object, which shows you the object's type and its unique identifier. To print out all the values of generator, we need to convert it to a sequence type like a list, which will force all of the values to be generated:

```
# this will not be very helpful
print(range(100))
```

```
# this will show you all the generated values
print(list(range(100)))
```

You can use all these iterables almost interchangeably because they all use the same interface for iterating over values: every *iterable* object has a method which can be used to return an *iterator* over that object. The iterable and the iterator together form a consistent interface which can be used to loop over a sequence of values – whether those values are all stored in memory or calculated as they are needed:

- The *iterable* has a method for accessing an item by its index. For example, a list just returns the item which is stored in a particular position. A range, on the other hand, *calculates* the integer in the range which corresponds to a particular index.
- The *iterator* "keeps your place" in the sequence, and has a method which lets you access the next element. There can be multiple iterators associated with a single iterable at the same time – each one in a different place in the iteration. For example, you can iterate over the same list in both levels of a nested loop – each loop uses its own *iterator*, and they do not interfere with each other:

```
    animals = ['cat', 'dog', 'fish']

    for first_animal in animals:
        for second_animal in animals:
            print("Yesterday I bought a %s. Today I bought a %s." % (first_animal, second_animal))
```

We will look in more detail at how these methods are defined in a later chapter, when we discuss writing custom objects. For now, here are some more examples of built-in generators defined in Python's `itertools` module:

```
# we need to import the module in order to use it
import itertools

# unlike range, count doesn't have an upper bound, and is not restricted to integers
for i in itertools.count(1):
    print(i) # 1, 2, 3....

for i in itertools.count(1, 0.5):
    print(i) # 1.0, 1.5, 2.0....

# cycle repeats the values in another iterable over and over
for animal in itertools.cycle(['cat', 'dog']):
    print(animal) # 'cat', 'dog', 'cat', 'dog'...

# repeat repeats a single item
for i in itertools.repeat(1): # ...forever
    print(i) # 1, 1, 1....

for i in itertools.repeat(1, 3): # or a set number of times
    print(i) # 1, 1, 1

# chain combines multiple iterables sequentially
for i in itertools.chain(numbers, animals):
    print(i) # print all the numbers and then all the animals
```

Some of these generators can go on for ever, so if you use them in a `for` loop you will need some other check to make the loop terminate!

There is also a built-in function called `zip` which allows us to combine multiple iterables pairwise. It also outputs a generator:

```
for i in zip((1, 2, 3), (4, 5, 6)):
    print(i)
```

```
for i in zip(range(5), range(5, 10), range(10, 15)):
    print(i)
```

The combined iterable will be the same length as the shortest of the component iterables – if any of the component iterables are longer than that, their trailing elements will be discarded.

### Exercise 4

1. Create a tuple of month names and a tuple of the number of days in each month (assume that February has 28 days). Using a single `for` loop, construct a dictionary which has the month names as keys and the corresponding day numbers as values.
2. Now do the same thing without using a `for` loop.

## Comprehensions

Suppose that we have a list of numbers, and we want to build a new list by doubling all the values in the first list. Or that we want to extract all the even numbers from a list of numbers. Or that we want to find and capitalise all the animal names in a list of animal names that start with a vowel. We can do each of these things by iterating over the original list, performing some kind of check on each element in turn, and appending values to a new list as we go:

```
numbers = [1, 5, 2, 12, 14, 7, 18]

doubles = []
for number in numbers:
    doubles.append(2 * number)

even_numbers = []
for number in numbers:
    if number % 2 == 0:
        even_numbers.append(number)

animals = ['aardvark', 'cat', 'dog', 'opossum']

vowel_animals = []
for animal in animals:
    if animal[0] in 'aeiou':
        vowel_animals.append(animal.title())
```

That's quite an unwieldy way to do something very simple. Fortunately, we can rewrite simple loops like this to use a cleaner and more readable syntax by using *comprehensions*.

A comprehension is a kind of filter which we can define on an iterable based on some condition. The result is another iterable. Here are some examples of list comprehensions:

```
doubles = [2 * number for number in numbers]
even_numbers = [number for number in numbers if number % 2 == 0]
vowel_animals = [animal.title() for animal in animals if animal[0] in 'aeiou']
```

The comprehension is the part written between square brackets on each line. Each of these comprehensions results in the creation of a new `list` object.

You can think of the comprehension as a compact form of `for` loop, which has been rearranged slightly.

- The first part (`2 * number` or `number` or `animal.title()`) defines what is going to be inserted into the new list at each step of the loop. This is usually some function of each item in the original iterable as it is processed.
- The middle part (`for number in numbers` or `for animal in animals`) corresponds to the first line of a `for` loop, and defines what iterable is being iterated over and what variable name each item is given inside the loop.
- The last part (nothing or `if number % 2 == 0` or `if animal[0] in 'aeiou'`) is a condition which filters out some of the original items. Only items for which the condition is true will be processed (as described in the first part) and included in the new list. You don't have to include this part – in the first example, we want to double *all* the numbers in the original list.

List comprehensions can be used to replace loops that are a lot more complicated than this – even nested loops. The more complex the loop, the more complicated the corresponding list comprehension is likely to be. A long and convoluted list comprehension can be very difficult for someone reading your code to understand – sometimes it's better just to write the loop out in full.

The final product of a comprehension doesn't have to be a list. You can create dictionaries or generators in a very similar way – a generator expression uses round brackets instead of square brackets, a set comprehension uses curly brackets, and a dict comprehension uses curly brackets *and* separates the key and the value using a colon:

```
numbers = [1, 5, 2, 12, 14, 7, 18]

# a generator comprehension
doubles_generator = (2 * number for number in numbers)

# a set comprehension
doubles_set = {2 * number for number in numbers}

# a dict comprehension which uses the number as the key and the doubled number as the value
doubles_dict = {number: 2 * number for number in numbers}
```

If your generator expression is a parameter being passed to a function, like `sum`, you can leave the round brackets out:

```
sum_doubles = sum(2 * number for number in numbers)
```

**Note:** dict and set comprehensions were introduced in Python 3. In Python 2 you have to create a list or generator instead and convert it to a set or a dict yourself.

## Exercise 5

1. Create a string which contains the first ten positive integers separated by commas and spaces. Remember that you can't join numbers – you have to convert them to strings first. Print the output string.
2. Rewrite the calendar program from exercise 3 using nested comprehensions instead of nested loops. Try to append a string to one of the week lists, to make sure that you haven't reused the same list instead of creating a separate list for each week.
3. Now do something similar to create a calendar which is a list with 52 empty sublists (one for each week in the whole year). Hint: how would you modify the nested `for` loops?

## The `break` and `continue` statements

### `break`

Inside the loop body, you can use the `break` statement to exit the loop immediately. You might want to test for a special case which will result in immediate exit from the loop. For example:

```
x = 1

while x <= 10:
    if x == 5:
        break

    print(x)
    x += 1
```

The code fragment above will only print out the numbers `1` to `4`. In the case where `x` is `5`, the `break` statement will be encountered, and the flow of control will leave the loop immediately.

### `continue`

The `continue` statement is similar to the `break` statement, in that it causes the flow of control to exit the current loop body at the point of encounter – but the loop itself is not exited. For example:

```
for x in range(1, 10 + 1): # this will count from 1 to 10
    if x == 5:
        continue

    print(x)
```

This fragment will print all the numbers from `1` to `10` *except* `5`. In the case where `x` is `5`, the `continue` statement will be encountered, and the flow of control will leave that loop body – but then the loop will *continue* with the next element in the range.

Note that if we replaced `break` with `continue` in the first example, we would get an infinite loop – because the `continue` statement would be triggered before `x` could be updated. `x` would stay equal to `5`, and keep triggering the `continue` statement, for ever!

## Using `break` to simulate a *do-while* loop

Recall that a `while` loop checks the condition *before* executing the loop body for the first time. Sometimes this is convenient, but sometimes it's not. What if you always need to execute the loop body at least once?

```
age = input("Please enter your age: ")
while not valid_number(age): # let's assume that we've defined valid_number elsewhere
    age = input("Please enter your age: ")
```

We have to ask the user for input at least once, because the condition depends on the user input – so we have to do it once outside the loop. This is inconvenient, because we have to repeat the contents of the loop body – and unnecessary repetition is usually a bad idea. What if we want to change the message to the user later, and forget to change it in both places? What if the loop body contains many lines of code?

Many other languages offer a structure called a *do-while* loop, or a *repeat-until* loop, which checks the condition *after* executing the loop body. That means that the loop body will always be executed at least once. Python doesn't have a structure like this, but we can simulate it with the help of the `break` statement:

```
while True:
    age = input("Please enter your age: ")
    if valid_number(age):
        break
```

We have moved the condition *inside* the loop body, and we can check it at the end, *after* asking the user for input. We have replaced the condition in the `while` statement with `True` – which is, of course, always true. Now the `while` statement will *never* terminate after checking the condition – it can *only* terminate if the `break` statement is triggered.

This trick can help us to make this particular loop use case look better, but it has its disadvantages. If we accidentally leave out the `break` statement, or write the loop in such a way that it can never be triggered, we will have an infinite loop! This code can also be more difficult to understand, because the actual condition which makes the loop terminate is hidden inside the body of the loop. You should therefore use this construct sparingly. Sometimes it's possible to rewrite the loop in such a way that the condition can be checked before the loop body *and* repetition is avoided:

```
age = None # we can initialise age to something which is not a valid number
while not valid_number(age): # now we can use the condition before asking the user anything
    age = input("Please enter your age: ")
```

### Exercise 6

1. Write a program which repeatedly prompts the user for an integer. If the integer is even, print the integer. If the integer is odd, don't print anything. Exit the program if the user enters the integer `99`.
2. Some programs ask the user to input a variable number of data entries, and finally to enter a specific character or string (called a *sentinel*) which signifies that there are no more entries. For example, you could be asked to enter your PIN followed by a hash (`#`). The hash is the sentinel which indicates that you have finished entering your PIN.

   Write a program which averages positive integers. Your program should prompt the user to enter integers until the user enters a negative integer. The negative integer should be discarded, and you should print the average of all the previously entered integers.
3. Implement a simple calculator with a menu. Display the following options to the user, prompt for a selection, and carry out the requested action (e.g. prompt for two numbers and add them). After each operation, return the user to the menu. Exit the program when the user selects `0`. If the user enters a number which is not in the menu, ignore the input and redisplay the menu. You can assume that the user will enter a valid integer:

```
    -- Calculator Menu --
    0. Quit
    1. Add two numbers
    2. Subtract two numbers
    3. Multiply two numbers
    4. Divide two numbers
```

### Using loops to simplify code

We can use our knowledge of loops to simplify some kinds of redundant code. Consider this example, in which we prompt a user for some personal details:

```
name = input("Please enter your name: ")
surname = input("Please enter your surname: ")
# let's store these as strings for now, and convert them to numbers later
age = input("Please enter your age: ")
height = input("Please enter your height: ")
weight = input("Please enter your weight: ")
```

There's a lot of repetition in this snippet of code. Each line is exactly the same except for the name of the variable and the name of the property we ask for (and these values match each other, so there's really only one difference). When we write code like this we're likely to do a lot of copying and pasting, and it's easy to make a mistake. If we ever want to change something, we'll need to change each line.

How can we improve on this? We can separate the parts of these lines that differ from the parts that don't, and use a loop to iterate over them. Instead of storing the user input in separate variables, we are going to use a dictionary – we can easily use the property names as keys, and it's a sensible way to group these values:

```
person = {}

for prop in ["name", "surname", "age", "height", "weight"]:
    person[prop] = input("Please enter your %s: " % prop)
```

Now there is no unnecessary duplication. We can easily change the string that we use as a prompt, or add more code to execute for each property – we will only have to edit the code in one place, not in five places. To add another property, all we have to do is add another name to the list.

### Exercise 7

1. Modify the example above to include type conversion of the properties: age should be an integer, height and weight should be floats, and name and surname should be strings.

## Answers to exercises

### Answer to exercise 1

1. Here is an example program:

```
    total = 0
    number = 0

    while total < 200:
        number += 1
        total += number**2

    print("Total: %d" % total)
    print("Last number: %d" % number)
```

2. Here is an example program:

```
    GUESSES_ALLOWED = 10
    SECRET_WORD = "caribou"

    guesses_left = GUESSES_ALLOWED
    guessed_word = None

    while guessed_word != SECRET_WORD and guesses_left:
        guessed_word = input("Guess a word: ")

        if guessed_word == SECRET_WORD:
            print("You guessed! Congratulations!")
        else:
            guesses_left -= 1
            print("Incorrect! You have %d guesses left." % guesses_left)
```

### Answer to exercise 2

1. Here is an example program:

```
total = 0

for i in range(1, 10 + 1):
    total += i

print(total)
```

2. Remember that we can use the `sum` function to sum a sequence:

```
print(sum(range(1, 10 + 1)))
```

3. Here is an example program:

```
num = int(input("Please enter an integer: "))

num_fac = 1
for i in range(1, num + 1):
    num_fac *= i

print("%d! = %d" % (num, num_fac))
```

4. Here is an example program:

```
total = 0
product = 1

for i in range(1, 10 + 1):
    num = float(input("Please enter number %d: " % i))
    total += num
    product *= num

average = total/10

print("Sum: %g\nProduct: %g\nAverage: %g" % (total, product, average))
```

5. Here is an example program:

```
numbers = []

for i in range(10):
    numbers[i] = float(input("Please enter number %d: " % (i + 1)))

total = 0
product = 1

for num in numbers:
    total += num
    product *= num

average = total/10

print("Sum: %g\nProduct: %g\nAverage: %g" % (total, product, average))
```

## Answer to exercise 3

1. Here is an example program:

```
calendar = []

for m in range(12):
    month = []

    for w in range(4):
        month.append([])

    calendar.append(month)
```

2. Here is an example program:

```
(JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST,
SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER) = range(12)

(WEEK_1, WEEK_2, WEEK_3, WEEK_4) = range(4)

calendar[JULY][WEEK_2].append("Go on holiday!")
```

## Answer to exercise 4

1. Here is an example program:

```
months = ("January", "February", "March", "April", "May", "June",
          "July", "August", "September", "October",
          "November", "December")

num_days = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

month_dict = {}

for month, days in zip(months, days):
    month_dict[month] = days
```

2. Here is an example program:

```
months = ("January", "February", "March", "April", "May", "June",
          "July", "August", "September", "October",
          "November", "December")

num_days = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

# the zipped output is a sequence of two-element tuples,
# so we can just use a dict conversion.
month_dict = dict(zip(months, days))
```

## Answer to exercise 5

1. Here is an example program:

```
number_string = ", ".join(str(n) for n in range(1, 11))
print(number_string)
```

2. Here is an example program:

```
calendar = [[[] for w in range(4)] for m in range(12)]

(JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST,
SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER) = range(12)

(WEEK_1, WEEK_2, WEEK_3, WEEK_4) = range(4)

calendar[JULY][WEEK_2].append("Go on holiday!")
```

3. `calendar = [[] for w in range(4) for m in range(12)]`

## Answer to exercise 6

1. Here is an example program:

```
while (True):
    num = int(input("Enter an integer: "))
    if num == 99:
        break
    if num % 2:
        continue
    print num
```

2. Here is an example program:

```
print("Please enter positive integers to be averaged. Enter a negative integer to terminate the

nums = []

while True:
    num = int(input("Enter a number: "))

    if num < 0:
        break

    nums.append(num)

average = float(sum(nums))/len(nums)
print("average = %g" % average)
```

3. Here is an example program:

```
menu = """-- Calculator Menu --
0. Quit
1. Add two numbers
2. Subtract two numbers
3. Multiply two numbers
4. Divide two numbers"""

selection = None

while selection != 0:
    print(menu)
    selection = int(input("Select an option: "))

    if selection not in range(5):
        print("Invalid option: %d" % selection)
        continue

    if selection == 0:
        continue

    a = float(input("Please enter the first number: "))
    b = float(input("Please enter the second number: "))

    if selection == 1:
        result = a + b
    elif selection == 2:
        result = a - b
    elif selection == 3:
        result = a * b
    elif selection == 4:
        result = a / b

    print("The result is %g." % result)
```

## Answer to exercise 7

1. Here is an example program:

```
person = {}

properties = [
    ("name", str),
    ("surname", str),
    ("age", int),
    ("height", float),
    ("weight", float),
]

for prop, p_type in properties:
    person[prop] = p_type(input("Please enter your %s: " % prop))
```

CHAPTER 7

# Errors and exceptions

## Errors

Errors or mistakes in a program are often referred to as bugs. They are almost always the fault of the programmer. The process of finding and eliminating errors is called debugging. Errors can be classified into three major groups:

- Syntax errors
- Runtime errors
- Logical errors

### Syntax errors

Python will find these kinds of errors when it tries to parse your program, and exit with an error message without running anything. Syntax errors are mistakes in the use of the Python language, and are analogous to spelling or grammar mistakes in a language like English: for example, the sentence *Would you some tea?* does not make sense – it is missing a verb.

Common Python syntax errors include:

- leaving out a keyword
- putting a keyword in the wrong place
- leaving out a symbol, such as a colon, comma or brackets
- misspelling a keyword
- incorrect indentation
- empty block

**Note:** it is illegal for any block (like an `if` body, or the body of a function) to be left completely empty. If you want a block to do nothing, you can use the `pass` statement inside the block.

Python will do its best to tell you where the error is located, but sometimes its messages can be misleading: for example, if you forget to escape a quotation mark inside a string you may get a syntax error referring to a place later in your code, even though that is not the real source of the problem. If you can't see anything wrong on the line specified in the error message, try backtracking through the previous few lines. As you program more, you will get better at identifying and fixing errors.

Here are some examples of syntax errors in Python:

```
myfunction(x, y):
    return x + y

else:
    print("Hello!")

if mark >= 50
    print("You passed!")

if arriving:
    print("Hi!")
esle:
    print("Bye!")

if flag:
print("Flag is set!")
```

## Runtime errors

If a program is syntactically correct – that is, free of syntax errors – it will be run by the Python interpreter. However, the program may exit unexpectedly during execution if it encounters a *runtime error* – a problem which was not detected when the program was parsed, but is only revealed when a particular line is executed. When a program comes to a halt because of a runtime error, we say that it has crashed.

Consider the English instruction *flap your arms and fly to Australia.* While the instruction is structurally correct and you can understand its meaning perfectly, it is impossible for you to follow it.

Some examples of Python runtime errors:

- division by zero
- performing an operation on incompatible types
- using an identifier which has not been defined
- accessing a list element, dictionary value or object attribute which doesn't exist
- trying to access a file which doesn't exist

Runtime errors often creep in if you don't consider all possible values that a variable could contain, especially when you are processing user input. You should always try to add checks to your code to make sure that it can deal with bad input and edge cases gracefully. We will look at this in more detail in the chapter about exception handling.

## Logical errors

Logical errors are the most difficult to fix. They occur when the program runs without crashing, but produces an incorrect result. The error is caused by a mistake in the program's logic. You won't get an error message, because no syntax or runtime error has occurred. You will have to find the problem on your own by reviewing all the relevant parts of your code – although some tools can flag suspicious code which looks like it could cause unexpected behaviour.

Sometimes there can be absolutely nothing wrong with your Python implementation of an algorithm – the algorithm itself can be incorrect. However, more frequently these kinds of errors are caused by programmer carelessness. Here are some examples of mistakes which lead to logical errors:

- using the wrong variable name
- indenting a block to the wrong level
- using integer division instead of floating-point division

- getting operator precedence wrong
- making a mistake in a boolean expression
- off-by-one, and other numerical errors

If you misspell an identifier name, you may get a runtime error or a logical error, depending on whether the misspelled name is defined.

A common source of variable name mix-ups and incorrect indentation is frequent copying and pasting of large blocks of code. If you have many duplicate lines with minor differences, it's very easy to miss a necessary change when you are editing your pasted lines. You should always try to factor out excessive duplication using functions and loops – we will look at this in more detail later.

### Exercise 1

1. Find all the syntax errors in the code snippet above, and explain why they are errors.
2. Find potential sources of runtime errors in this code snippet:

```
dividend = float(input("Please enter the dividend: "))
divisor = float(input("Please enter the divisor: "))
quotient = dividend / divisor
quotient_rounded = math.round(quotient)
```

3. Find potential sources of runtime errors in this code snippet:

```
for x in range(a, b):
    print("(%f, %f, %f)" % my_list[x])
```

4. Find potential sources of logic errors in this code snippet:

```
product = 0
for i in range(10):
    product *= i

sum_squares = 0
for i in range(10):
    i_sq = i**2
sum_squares += i_sq

nums = 0
for num in range(10):
    num += num
```

## Handling exceptions

Until now, the programs that we have written have generally ignored the fact that things can go wrong. We have have tried to prevent runtime errors by checking data which may be incorrect before we used it, but we haven't yet seen how we can handle errors when they do occur – our programs so far have just crashed suddenly whenever they have encountered one.

There are some situations in which runtime errors are likely to occur. Whenever we try to read a file or get input from a user, there is a chance that something unexpected will happen – the file may have been moved or deleted, and the user may enter data which is not in the right format. Good programmers should add safeguards to their programs so that common situations like this can be handled gracefully – a program which crashes whenever it encounters an easily foreseeable problem is not very pleasant to use. Most users expect programs to be robust enough to recover from these kinds of setbacks.

If we know that a particular section of our program is likely to cause an error, we can tell Python what to do if it does happen. Instead of letting the error crash our program we can intercept it, do something about it, and allow the program to continue.

All the runtime (and syntax) errors that we have encountered are called *exceptions* in Python – Python uses them to indicate that something *exceptional* has occurred, and that your program cannot continue unless it is *handled*. All exceptions are subclasses of the `Exception` class – we will learn more about classes, and how to write your own exception types, in later chapters.

## The `try` and `except` statements

To handle possible exceptions, we use a *try-except* block:

```
try:
    age = int(input("Please enter your age: "))
    print("I see that you are %d years old." % age)
except ValueError:
    print("Hey, that wasn't a number!")
```

Python will *try* to process all the statements inside the *try* block. If a `ValueError` occurs at any point as it is executing them, the flow of control will immediately pass to the *except* block, and any remaining statements in the *try* block will be skipped.

In this example, we know that the error is likely to occur when we try to convert the user's input to an integer. If the input string is not a number, this line will trigger a `ValueError` – that is why we specified it as the type of error that we are going to handle.

We could have specified a more general type of error – or even left the type out entirely, which would have caused the `except` clause to match *any* kind of exception – but that would have been a bad idea. What if we got a completely different error that we hadn't predicted? It would be handled as well, and we wouldn't even notice that anything unusual was going wrong. We may also want to react in different ways to different kinds of errors. We should always try pick specific rather than general error types for our `except` clauses.

It is possible for one `except` clause to handle more than one kind of error: we can provide a tuple of exception types instead of a single type:

```
try:
    dividend = int(input("Please enter the dividend: "))
    divisor = int(input("Please enter the divisor: "))
    print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except(ValueError, ZeroDivisionError):
    print("Oops, something went wrong!")
```

A *try-except* block can also have multiple `except` clauses. If an exception occurs, Python will check each `except` clause from the top down to see if the exception type matches. If none of the `except` clauses match, the exception will be considered *unhandled*, and your program will crash:

```
try:
    dividend = int(input("Please enter the dividend: "))
    divisor = int(input("Please enter the divisor: "))
    print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except ValueError:
    print("The divisor and dividend have to be numbers!")
except ZeroDivisionError:
    print("The dividend may not be zero!")
```

Note that in the example above if a `ValueError` occurs we won't know whether it was caused by the dividend or the divisor not being an integer – either one of the input lines could cause that error. If we want to give the user more specific feedback about which input was wrong, we will have to wrap each input line in a separate *try-except* block:

```
try:
    dividend = int(input("Please enter the dividend: "))
except ValueError:
    print("The dividend has to be a number!")

try:
    divisor = int(input("Please enter the divisor: "))
except ValueError:
    print("The divisor has to be a number!")

try:
    print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except ZeroDivisionError:
    print("The dividend may not be zero!")
```

In general, it is a better idea to use exception handlers to protect small blocks of code against specific errors than to wrap large blocks of code and write vague, generic error recovery code. It may sometimes seem inefficient and verbose to write many small *try-except* statements instead of a single catch-all statement, but we can mitigate this to some extent by making effective use of loops and functions to reduce the amount of code duplication.

## How an exception is handled

When an exception occurs, the normal flow of execution is interrupted. Python checks to see if the line of code which caused the exception is inside a *try* block. If it is, it checks to see if any of the *except* blocks associated with the *try* block can handle that type of exception. If an appropriate handler is found, the exception is handled, and the program continues from the next statement after the end of that *try-except*.

If there is no such handler, or if the line of code was *not* in a *try* block, Python will go up one level of scope: if the line of code which caused the exception was inside a *function*, that function will exit immediately, and the line which *called* the function will be treated as if *it* had thrown the exception. Python will check if *that* line is inside a *try* block, and so on. When a function is called, it is placed on Python's *stack*, which we will discuss in the chapter about functions. Python traverses this stack when it tries to handle an exception.

If an exception is thrown by a line which is in the main body of your program, not inside a function, the program will terminate. When the exception message is printed, you should also see a *traceback* – a list which shows the path the exception has taken, all the way back to the original line which caused the error.

## Error checks vs exception handling

Exception handling gives us an alternative way to deal with error-prone situations in our code. Instead of performing more checks before we do something to make sure that an error will not occur, we just try to do it – and if an error does occur we handle it. This can allow us to write simpler and more readable code. Let's look at a more complicated input example – one in which we want to keep asking the user for input until the input is correct. We will try to write this example using the two different approaches:

```
# with checks

n = None
while n is None:
    s = input("Please enter an integer: ")
    if s.lstrip('-').isdigit():
        n = int(s)
    else:
        print("%s is not an integer." % s)

# with exception handling

n = None
while n is None:
    try:
        s = input("Please enter an integer: ")
        n = int(s)
    except ValueError:
        print("%s is not an integer." % s)
```

In the first code snippet, we have to write quite a convoluted check to test whether the user's input is an integer – first we strip off a minus sign if it exists, and then we check if the rest of the string consists only of digits. But there's a very simple criterion which is also what we really want to know: will this string cause a `ValueError` if we try to convert it to an integer? In the second snippet we can in effect check for exactly the right condition instead of trying to replicate it ourselves – something which isn't always easy to do. For example, we could easily have forgotten that integers can be negative, and written the check in the first snippet incorrectly.

Here are a few other advantages of exception handling:

- It separates normal code from code that handles errors.
- Exceptions can easily be passed along functions in the stack until they reach a function which knows how to handle them. The intermediate functions don't need to have any error-handling code.
- Exceptions come with lots of useful error information built in – for example, they can print a traceback which helps us to see exactly where the error occurred.

## The `else` and `finally` statements

There are two other clauses that we can add to a *try-except* block: `else` and `finally`. `else` will be executed only if the `try` clause doesn't raise an exception:

```
try:
    age = int(input("Please enter your age: "))
except ValueError:
    print("Hey, that wasn't a number!")
else:
    print("I see that you are %d years old." % age)
```

We want to print a message about the user's age only if the integer conversion succeeds. In the first exception handler example, we put this print statement directly after the conversion inside the `try` block. In both cases, the statement will only be executed if the conversion statement doesn't raise an exception, but putting it in the `else` block is better practice – it means that the only code inside the `try` block is the single line that is the potential source of the error that we want to handle.

When we edit this program in the future, we may introduce additional statements that should also be executed if the age input is successfully converted. Some of these statements may also potentially raise a `ValueError`. If we don't notice this, and put them inside the `try` clause, the `except` clause will also handle these errors if they occur. This is likely to cause some odd and unexpected behaviour. By putting all this extra code in the `else` clause instead, we avoid taking this risk.

The `finally` clause will be executed at the end of the *try-except* block no matter what – if there is no exception, if an exception is raised and handled, if an exception is raised and not handled, and even if we exit the block using `break`, `continue` or `return`. We can use the `finally` clause for cleanup code that we always want to be executed:

```
try:
    age = int(input("Please enter your age: "))
except ValueError:
    print("Hey, that wasn't a number!")
else:
    print("I see that you are %d years old." % age)
finally:
    print("It was really nice talking to you.  Goodbye!")
```

### Exercise 2

1. Extend the program in exercise 7 of the loop control statements chapter to include exception handling. Whenever the user enters input of the incorrect type, keep prompting the user for the same value until it is entered correctly. Give the user sensible feedback.
2. Add a *try-except* statement to the body of this function which handles a possible `IndexError`, which could occur if the index provided exceeds the length of the list. Print an error message if this happens:

```
def print_list_element(thelist, index):
    print(thelist[index])
```

3. This function adds an element to a list inside a dict of lists. Rewrite it to use a *try-except* statement which handles a possible `KeyError` if the list with the name provided doesn't exist in the dictionary yet, instead of checking beforehand whether it does. Include `else` and `finally` clauses in your *try-except* block:

```
def add_to_list_in_dict(thedict, listname, element):
    if listname in thedict:
        l = thedict[listname]
        print("%s already has %d elements." % (listname, len(l)))
    else:
        thedict[listname] = []
        print("Created %s." % listname)

    thedict[listname].append(element)

    print("Added %s to %s." % (element, listname))
```

## The `with` statement

## Using the exception object

Python's exception objects contain more information than just the error type. They also come with some kind of message – we have already seen some of these messages displayed when our programs have crashed. Often these messages aren't very user-friendly – if we want to report an error to the user we usually need to write a more descriptive message which explains how the error is related to what the user did. For example, if the error was caused by incorrect input, it is helpful to tell the user which of the input values was incorrect.

Sometimes the exception message contains useful information which we want to display to the user. In order to access the message, we need to be able to access the exception object. We can assign the object to a variable that we can use inside the `except` clause like this:

```
try:
    age = int(input("Please enter your age: "))
except ValueError as err:
    print(err)
```

`err` is not a string, but Python knows how to convert it into one – the string representation of an exception is the message, which is exactly what we want. We can also combine the exception message with our own message:

```
try:
    age = int(input("Please enter your age: "))
except ValueError as err:
    print("You entered incorrect age input: %s" % err)
```

Note that inserting a variable into a formatted string using `%s` also converts the variable to a string.

## Raising exceptions

We can raise exceptions ourselves using the `raise` statement:

```
try:
    age = int(input("Please enter your age: "))
    if age < 0:
        raise ValueError("%d is not a valid age. Age must be positive or zero.")
except ValueError as err:
    print("You entered incorrect age input: %s" % err)
else:
    print("I see that you are %d years old." % age)
```

We can raise our own `ValueError` if the age input is a valid integer, but it's negative. When we do this, it has exactly the same effect as any other exception – the flow of control will immediately exit the `try` clause at this point and pass to the `except` clause. This `except` clause can match our exception as well, since it is also a `ValueError`.

We picked `ValueError` as our exception type because it's the most appropriate for this kind of error. There's nothing stopping us from using a completely inappropriate exception class here, but we should try to be consistent. Here are a few common exception types which we are likely to raise in our own code:

- `TypeError`: this is an error which indicates that a variable has the wrong *type* for some operation. We might raise it in a function if a parameter is not of a type that we know how to handle.
- `ValueError`: this error is used to indicate that a variable has the right *type* but the wrong *value*. For example, we used it when `age` was an integer, but the wrong *kind* of integer.
- `NotImplementedError`: we will see in the next chapter how we use this exception to indicate that a class's method has to be implemented in a child class.

We can also write our own custom exception classes which are based on existing exception classes – we will see some examples of this in a later chapter.

Something we may want to do is raise an exception that we have just intercepted – perhaps because we want to handle it partially in the current function, but also want to respond to it in the code which called the function:

```
try:
    age = int(input("Please enter your age: "))
except ValueError as err:
    print("You entered incorrect age input: %s" % err)
    raise err
```

## Exercise 3

1. Rewrite the program from the first question of exercise 2 so that it prints the text of Python's original exception inside the `except` clause instead of a custom message.
2. Rewrite the program from the second question of exercise 2 so that the exception which is caught in the `except` clause is re-raised after the error message is printed.

## Debugging programs

Syntax errors are usually quite straightforward to debug: the error message shows us the line in the file where the error is, and it should be easy to find it and fix it.

Runtime errors can be a little more difficult to debug: the error message and the traceback can tell us exactly where the error occurred, but that doesn't necessarily tell us what the problem is. Sometimes they are caused by something obvious, like an incorrect identifier name, but sometimes they are triggered by a particular state of the program – it's not always clear which of many variables has an unexpected value.

Logical errors are the most difficult to fix because they don't cause any errors that can be traced to a particular line in the code. All that we know is that the code is not behaving as it should be – sometimes tracking down the area of the code which is causing the incorrect behaviour can take a long time.

It is important to test your code to make sure that it behaves the way that you expect. A quick and simple way of testing that a function is doing the right thing, for example, is to insert a print statement after every line which outputs the intermediate results which were calculated on that line. Most programmers intuitively do this as they are writing a function, or perhaps if they need to figure out why it isn't doing the right thing:

```python
def hypotenuse(x, y):
    print("x is %f and y is %f" % (x, y))
    x_2 = x**2
    print(x_2)
    y_2 = y**2
    print(y_2)
    z_2 = x_2 + y_2
    print(z_2)
    z = math.sqrt(z_2)
    print(z)
    return z
```

This is a quick and easy thing to do, and even experienced programmers are guilty of doing it every now and then, but this approach has several disadvantages:

- As soon as the function is working, we are likely to delete all the print statements, because we don't want our program to print all this debugging information all the time. The problem is that code often changes – the next time we want to test this function we will have to add the print statements all over again.
- To avoid rewriting the print statements if we happen to need them again, we may be tempted to comment them out instead of deleting them – leaving them to clutter up our code, and possibly become so out of sync that they end up being completely useless anyway.
- To print out all these intermediate values, we had to spread out the formula inside the function over many lines. Sometimes it is useful to break up a calculation into several steps, if it is very long and putting it all on one line makes it hard to read, but sometimes it just makes our code unnecessarily verbose. Here is what the function above would normally look like:

```python
def hypotenuse(x, y):
    return math.sqrt(x**2 + y**2)
```

How can we do this better? If we want to *inspect* the values of variables at various steps of a program's execution, we can use a tool like `pdb`. If we want our program to print out informative messages, possibly to a file, and we want to be able to control the level of detail at runtime without having to change anything in the code, we can use *logging*.

Most importantly, to check that our code is working correctly now and will *keep* working correctly, we should write a permanent suite of tests which we can run on our code regularly. We will discuss testing in more detail in a later chapter.

## Debugging tools

There are some automated tools which can help us to debug errors, and also to keep our code as correct as possible to minimise the chances of new errors creeping in. Some of these tools analyse our program's syntax, reporting errors and bad programming style, while others let us analyse the program as it is running.

### Pyflakes, pylint, PyChecker and pep8

These four utilities analyse code for syntax errors as well as some kinds of runtime errors. They also print warnings about bad coding style, and about inefficient and potentially incorrect code – for example, variables and imported modules which are never used.

Pyflakes parses code instead of importing it, which means that it can't detect as many errors as other tools – but it is also safer to use, since there is no risk that it will execute broken code which does permanent damage to our system. This is mostly relevant when we use it as part of an automated system. It also means that Pyflakes is faster than other checkers.

Pylint and PyChecker do import the code that they check, and they produce more extensive lists of errors and warnings. They are used by programmers who find the functionality of pyflakes to be too basic.

Pep8 specifically targets bad coding style – it checks whether our code conforms to Pep 8, a specification document for good coding style.

Here is how we use these programs on the commandline:

```
pyflakes myprogram.py
pylint myprogram.py
pychecker myprogram.py
pep8 myprogram.py
```

### pdb

pdb is a built-in Python module which we can use to debug a program while it's running. We can either import the module and use its functions from inside our code, or invoke it as a script when running our code file. We can use pdb to step through our program, either line by line or in larger increments, inspect the state at each step, and perform a "post-mortem" of the program if it crashes.

Here is how we would use pdb in our code:

```python
import pdb

def our_function():
    bad_idea = 3 + "4"

pdb.run('our_function()')
```

Here is how we would run it as a script:

```
python3 -m pdb ourprogram.py
```

More extensive documentation, including the full list of commands which can be used inside the debugger, can be found at the link above.

## Logging

Sometimes it is valuable for a program to output messages to a console or a file as it runs. These messages can be used as a record of the program's execution, and help us to find errors. Sometimes a bug occurs intermittently, and we don't know what triggers it – if we only add debugging output to our program when we want to begin an active search for the bug, we may be unable to reproduce it. If our program logs messages to a file all the time, however, we may find that some helpful information has been recorded when we check the log after the bug has occurred.

Some kinds of messages are more important than others – errors are noteworthy events which should almost always be logged. Messages which record that an operation has been completed successfully may sometimes be useful, but are not as important as errors. Detailed messages which debug every step of a calculation can be interesting if we are trying to debug the calculation, but if they were printed all the time they would fill the console with noise (or make our log file really, really big).

We can use Python's `logging` module to add logging to our program in an easy and consistent way. Logging statements are almost like print statements, but whenever we log a message we specify a *level* for the message. When we run our program, we set a desired log level for the program. Only messages which have a level *greater than or equal to* the level which we have set will appear in the log. This means that we can temporarily switch on detailed logging and switch it off again just by changing the log level in one place.

There is a consistent set of logging level names which most languages use. In order, from the highest value (most severe) to the lowest value (least severe), they are:

- CRITICAL – for very serious errors
- ERROR – for less serious errors
- WARNING – for warnings
- INFO – for important informative messages
- DEBUG – for detailed debugging messages

These names are used for integer constants defined in the `logging` module. The module also provides methods which we can use to log messages. By default these messages are printed to the console, and the default log level is `WARNING`. We can configure the module to customise its behaviour – for example, we can write the messages to a file instead, raise or lower the log level and change the message format. Here is a simple logging example:

```python
import logging

# log messages to a file, ignoring anything less severe than ERROR
logging.basicConfig(filename='myprogram.log', level=logging.ERROR)

# these messages should appear in our file
logging.error("The washing machine is leaking!")
logging.critical("The house is on fire!")

# but these ones won't
logging.warning("We're almost out of milk.")
logging.info("It's sunny today.")
logging.debug("I had eggs for breakfast.")
```

There's also a special `exception` method which is used for logging exceptions. The level used for these messages is `ERROR`, but additional information about the exception is added to them. This method is intended to be used inside exception handlers instead of `error`:

```python
try:
    age = int(input("How old are you? "))
except ValueError as err:
    logging.exception(err)
```

If we have a large project, we may want to set up a more complicated system for logging – perhaps we want to format certain messages differently, log different messages to different files, or log to multiple locations at the same time. The logging module also provides us with *logger* and *handler* objects for this purpose. We can use multiple loggers to create our messages, customising each one independently. Different handlers are associated with different logging locations. We can connect up our loggers and handlers in any way we like – one logger can use many handlers, and multiple loggers can use the same handler.

### Exercise 4

1. Write logging configuration for a program which logs to a file called `log.txt` and discards all logs less important than `INFO`.
2. Rewrite the second program from exercise 2 so that it uses this logging configuration instead of printing messages to the console (except for the first print statement, which is the purpose of the function).
3. Do the same with the third program from exercise 2.

## Answers to exercises

### Answer to exercise 1

1. There are five syntax errors:
   (a) Missing `def` keyword in function definition
   (b) `else` clause without an `if`
   (c) Missing colon after `if` condition
   (d) Spelling mistake ("esle")
   (e) The `if` block is empty because the `print` statement is not indented correctly
2. (a) The values entered by the user may not be valid integers or floating-point numbers.
   (b) The user may enter zero for the divisor.
   (c) If the `math` library hasn't been imported, `math.round` is undefined.
3. (a) `a`, `b` and `my_list` need to be defined before this snippet.
   (b) The attempt to access the list element with index `x` may fail during one of the loop iterations if the range from `a` to `b` exceeds the size of `my_list`.
   (c) The string formatting operation inside the `print` statement expects `my_list[x]` to be a tuple with three numbers. If it has too many or too few elements, or isn't a tuple at all, the attempt to format the string will fail.
4. (a) If you are accumulating a number total by multiplication, not addition, you need to initialise the total to `1`, not `0`, otherwise the product will always be zero!
   (b) The line which adds `i_sq` to `sum_squares` is not aligned correctly, and will only add the last value of `i_sq` after the loop has concluded.
   (c) The wrong variable is used: at each loop iteration the current number in the range is added to itself and `nums` remains unchanged.

## Answer to exercise 2

1. Here is an example program:

```
person = {}

properties = [
    ("name", str),
    ("surname", str),
    ("age", int),
    ("height", float),
    ("weight", float),
]

for property, p_type in properties:
    valid_value = None

    while valid_value is None:
        try:
            value = input("Please enter your %s: " % property)
            valid_value = p_type(value)
        except ValueError:
            print("Could not convert %s '%s' to type %s. Please try again." % (property, value,

    person[property] = valid_value
```

2. Here is an example program:

```
def print_list_element(thelist, index):
    try:
        print(thelist[index])
    except IndexError:
        print("The list has no element at index %d." % index)
```

3. Here is an example program:

```
def add_to_list_in_dict(thedict, listname, element):
    try:
        l = thedict[listname]
    except KeyError:
        thedict[listname] = []
        print("Created %s." % listname)
    else:
        print("%s already has %d elements." % (listname, len(l)))
    finally:
        thedict[listname].append(element)
        print("Added %s to %s." % (element, listname))
```

## Answer to exercise 3

1. Here is an example program:

```
person = {}

properties = [
    ("name", str),
    ("surname", str),
    ("age", int),
    ("height", float),
    ("weight", float),
]

for property, p_type in properties:
    valid_value = None

    while valid_value is None:
        try:
            value = input("Please enter your %s: " % property)
            valid_value = p_type(value)
        except ValueError as ve:
            print(ve)

    person[property] = valid_value
```

2. Here is an example program:

```
    def print_list_element(thelist, index):
        try:
            print(thelist[index])
        except IndexError as ie:
            print("The list has no element at index %d." % index)
            raise ie
```

## Answer to exercise 4

1. Here is an example of the logging configuration:

```
    import logging
    logging.basicConfig(filename='log.txt', level=logging.INFO)
```

2. Here is an example program:

```
    def print_list_element(thelist, index):
        try:
            print(thelist[index])
        except IndexError:
            logging.error("The list has no element at index %d." % index)
```

3. Here is an example program:

```
    def add_to_list_in_dict(thedict, listname, element):
        try:
            l = thedict[listname]
        except KeyError:
            thedict[listname] = []
            logging.info("Created %s." % listname)
        else:
            logging.info("%s already has %d elements." % (listname, len(l)))
        finally:
            thedict[listname].append(element)
            logging.info("Added %s to %s." % (element, listname))
```

CHAPTER 8

# Functions

## Introduction

A function is a sequence of statements which performs some kind of task. We use functions to eliminate code duplication – instead of writing all the statements at every place in our code where we want to perform the same task, we define them in one place and refer to them by the function name. If we want to change how that task is performed, we will now mostly only need to change code in one place.

Here is a definition of a simple function which takes no parameters and doesn't return any values:

```
def print_a_message():
    print("Hello, world!")
```

We use the `def` statement to indicate the start of a function definition. The next part of the definition is the function name, in this case `print_a_message`, followed by round brackets (the definitions of any parameters that the function takes will go in between them) and a colon. Thereafter, everything that is indented by one level is the body of the function.

Functions *do things*, so you should always choose a function name which explains as simply as accurately as possible *what the function does*. This will usually be a verb or some phrase containing a verb. If you change a function so much that the name no longer accurately reflects what it does, you should consider updating the name – although this may sometimes be inconvenient.

This particular function always does exactly the same thing: it prints the message `"Hello, world!"`.

Defining a function does not make it run – when the flow of control reaches the function definition and executes it, Python just learns about the function and what it will do when we run it. To run a function, we have to *call* it. To call the function we use its name followed by round brackets (with any parameters that the function takes in between them):

```
print_a_message()
```

Of course we have already used many of Python's built-in functions, such as `print` and `len`:

```
print("Hello")
len([1, 2, 3])
```

Many objects in Python are *callable*, which means that you can call them like functions – a callable object has a special method defined which is executed when the object is called. For example, types such as `str`, `int` or `list` can be used as functions, to create new objects of that type (sometimes by converting an existing object):

```
num_str = str(3)
num = int("3")
```

```
people = list() # make a new (empty) list
people = list((1, 2, 3)) # convert a tuple to a new list
```

In general, classes (of which types are a subset) are callable – when we call a class we call its *constructor* method, which is used to create a new object of that class. We will learn more about classes in the next chapter, but you may recall that we already called some classes to make new objects when we raised exceptions:

```
raise ValueError("There's something wrong with your number!")
```

Because functions are objects in Python, we can treat them just like any other object – we can assign a function as the value of a variable. To refer to a function without calling it, we just use the function name without round brackets:

```
my_function = print_a_message

# later we can call the function using the variable name
my_function()
```

Because defining a function does not cause it to execute, we can use an identifier inside a function even if it hasn't been defined yet – as long as it becomes defined by the time we run the function. For example, if we define several functions which all call each other, the order in which we define them doesn't matter as long as they are all defined before we start using them:

```
def my_function():
    my_other_function()

def my_other_function():
    print("Hello!")

# this is fine, because my_other_function is now defined
my_function()
```

If we were to move that function call up, we would get an error:

```
def my_function():
    my_other_function()

# this is not fine, because my_other_function is not defined yet!
my_function()

def my_other_function():
    print("Hello!")
```

Because of this, it's a good idea to put all function definitions near the top of your program, so that they are executed before any of your other statements.

## Exercise 1

1. Create a function called `func_a`, which prints a message.
2. Call the function.
3. Assign the function object as a value to the variable `b`, without calling the function.
4. Now call the function using the variable `b`.

## Input parameters

It is very seldom the case that the task that we want to perform with a function is always exactly the same. There are usually minor differences to what we need to do under different circumstances. We don't want to write a slightly different function for each of these slightly different cases – that would defeat the object of the exercise! Instead, we want to pass information into the function and use it inside the function to tailor the function's behaviour to our exact needs. We express this information as a series of *input parameters*.

For example, we can make the function we defined above more useful if we make the message customisable:

```
def print_a_message(message):
    print(message)
```

More usefully, we can pass in two numbers and add them together:

```
def print_sum(a, b):
    print(a + b)
```

`a` and `b` are parameters. When we call this function, we have to pass two paramenters in, or we will get an error:

```
print_sum() # this won't work

print_sum(2, 3) # this is correct
```

In the example above, we are passing `2` and `3` as parameters to the function when we call it. That means that when the function is executed, the variable `a` will be given the value `2` and the variable `b` will be given the value `3`. You will then be able to refer to these values using the variable names `a` and `b` inside the function.

In languages which are statically typed, we have to declare the types of parameters when we define the function, and we can only use variables of those types when we call the function. If we want to perform a similar task with variables of different types, we must define a separate function which accepts those types.

In Python, parameters have no declared types. We can pass any kind of variable to the `print_message` function above, not just a string. We can use the `print_sum` function to add any two things which can be added: two integers, two floats, an integer and a float, or even two strings. We can also pass in an integer and a string, but although these are permitted as parameters, they cannot be added together, so we will get an error when we actually try to add them inside the function.

The advantage of this is that we don't have to write a lot of different `print_sum` functions, one for each different pair of types, when they would all be identical otherwise. The disadvantage is that since Python doesn't check parameter types against the function definition when a function is called, we may not immediately notice if the wrong type of parameter is passed in – if, for example, another person interacting with code that we have written uses parameter types that we did not anticipate, or if we accidentally get the parameters out of order.

This is why it is important for us to test our code thoroughly – something we will look at in a later chapter. If we intend to write code which is robust, especially if it is also going to be used by other people, it is also often a good idea to check function parameters early in the function and give the user feedback (by raising exceptions) if the are incorrect.

### Exercise 2

1. Create a function called `hypotenuse`, which takes two numbers as parameters and prints the square root of the sum of their squares.
2. Call this function with two floats.
3. Call this function with two integers.
4. Call this function with one integer and one float.

## Return values

The function examples we have seen above don't return any values – they just result in a message being printed. We often want to use a function to calculate some kind of value and then *return* it to us, so that we can store it in a variable and use it later. Output which is returned from a function is called a *return value*. We can rewrite the `print_sum` function to return the result of its addition instead of printing it:

```python
def add(a, b):
    return a + b
```

We use the `return` keyword to define a return value. To access this value when we call the function, we have to *assign* the result of the function to a variable:

```python
c = add(a, b)
```

Here the return value of the function will be assigned to `c` when the function is executed.

A function can only have a single return value, but that value can be a list or tuple, so in practice you can return as many different values from a function as you like. It usually only makes sense to return multiple values if they are tied to each other in some way. If you place several values after the `return` statement, separated by commas, they will automatically be converted to a tuple. Conversely, you can assign a tuple to multiple variables separated by commas at the same time, so you can *unpack* a tuple returned by a function into multiple variables:

```python
def divide(dividend, divisor):
    quotient = dividend // divisor
    remainder = dividend % divisor
    return quotient, remainder

# you can do this
q, r = divide(35, 4)

# but you can also do this
result = divide(67, 9)
q1 = result[0]
q2 = result[1]

# by the way, you can also do this
a, b = (1, 2)
# or this
c, d = [5, 6]
```

What happens if you try to assign one of our first examples, which don't have a return value, to a variable?

```python
mystery_output = print_message("Boo!")
print(mystery_output)
```

All functions do actually return *something*, even if we don't define a return value – the default return value is `None`, which is what our mystery output is set to.

When a `return` statement is reached, the flow of control immediately exits the function – any further statements in the function body will be skipped. We can sometimes use this to our advantage to reduce the number of conditional statements we need to use inside a function:

```python
def divide(dividend, divisor):
    if not divisor:
        return None, None # instead of dividing by zero

    quotient = dividend // divisor
    remainder = dividend % divisor
    return quotient, remainder
```

If the `if` clause is executed, the first `return` will cause the function to exit – so whatever comes after the `if` clause doesn't need to be inside an `else`. The remaining statements can simply be in the main body of the function, since they can only be reached if the `if` clause is not executed.

This technique can be useful whenever we want to check parameters at the beginning of a function – it means that we don't have to indent the main part of the function inside an `else` block. Sometimes it's more appropriate to raise an exception instead of returning a value like `None` if there is something wrong with one of the parameters:

```
def divide(dividend, divisor):
    if not divisor:
        raise ValueError("The divisor cannot be zero!")

    quotient = dividend // divisor
    remainder = dividend % divisor
    return quotient, remainder
```

Having multiple exit points scattered throughout your function can make your code difficult to read – most people expect a single `return` right at the end of a function. You should use this technique sparingly.

**Note:** in some other languages, only functions that return a value are called functions (because of their similarity to mathematical functions). Functions which have no return value are known as *procedures* instead.

### Exercise 3

1. Rewrite the `hypotenuse` function from exercise 2 so that it returns a value instead of printing it. Add exception handling so that the function returns `None` if it is called with parameters of the wrong type.
2. Call the function with two numbers, and print the result.
3. Call the function with two strings, and print the result.
4. Call the function with a number and a string, and print the result.

## The stack

Python stores information about functions which have been called in a *call stack*. Whenever a function is called, a new *stack frame* is added to the stack – all of the function's parameters are added to it, and as the body of the function is executed, local variables will be created there. When the function finishes executing, its stack frame is discarded, and the flow of control returns to wherever you were before you called the function, at the previous level of the stack.

If you recall the section about variable scope from the beginning of the course, this explains a little more about the way that variable names are resolved. When you use an identifier, Python will first look for it on the current level of the stack, and if it doesn't find it it will check the previous level, and so on – until either the variable is found or it isn't found anywhere and you get an error. This is why a local variable will always take precedence over a global variable with the same name.

Python also searches the stack whenever it handles an exception: first it checks if the exception can be handled in the current function, and if it cannot, it terminates the function and tries the next one down – until either the exception is handled on some level or the program itself has to terminate. The traceback you see when an exception is printed shows the path that Python took through the stack.

### Recursion

We can make a function call itself. This is known as *recursion*. A common example is a function which calculates numbers in the Fibonacci sequence: the zeroth number is `0`, the first number is `1`, and each subsequent number is the sum of the previous two numbers:

```python
def fibonacci(n):
    if n == 0:
        return 0

    if n == 1:
        return 1

    return fibonacci(n - 1) + fibonacci(n - 2)
```

Whenever we write a recursive function, we need to include some kind of condition which will allow it to *stop* recursing – an end case in which the function *doesn't* call itself. In this example, that happens at the beginning of the sequence: the first two numbers are *not* calculated from any previous numbers – they are constants.

What would happen if we omitted that condition from our function? When we got to *n = 2*, we would keep calling the function, trying to calculate `fibonacci(0)`, `fibonacci(-1)`, and so on. In theory, the function would end up recursing forever and never terminate, but in practice the program will crash with a `RuntimeError` and a message that we have exceeded the maximum recursion depth. This is because Python's stack has a finite size – if we keep placing instances of the function on the stack we will eventually fill it up and cause a *stack overflow*. Python protects itself from stack overflows by setting a limit on the number of times that a function is allowed to recurse.

Writing fail-safe recursive functions is difficult. What if we called the function above with a parameter of `-1`? We haven't included any error checking which guards against this, so we would skip over the end cases and try to calculate `fibonacci(-2)`, `fibonacci(-3)`, and keep going.

Any recursive function can be re-written in an *iterative* way which avoids recursion. For example:

```python
def fibonacci(n):
    current, next = 0, 1

    for i in range(n):
        current, next = next, current + next

    return current
```

This function uses *iteration* to count up to the desired value of *n*, updating variables to keep track of the calculation. All the iteration happens within a single instance of the function. Note that we assign new values to both variables at the same time, so that we can use both old values to calculate both new values on the right-hand side.

### Exercise 4

1. Write a recursive function which calculates the factorial of a given number. Use exception handling to raise an appropriate exception if the input parameter is not a positive integer, but allow the user to enter floats as long as they are whole numbers.

## Default parameters

The combination of the function name and the number of parameters that it takes is called the *function signature*. In statically typed languages, there can be multiple functions with the same name in the same scope as long as they have different numbers or types of parameters (in these languages, parameter types and return types are also part of the signature).

In Python, there can only be one function with a particular name defined in the scope – if you define another function with the same name, you will overwrite the first function. You must call this function with the correct number of parameters, otherwise you will get an error.

Sometimes there is a good reason to want to have two versions of the same function with different sets of parameters. You can achieve something similar to this by making some parameters *optional*. To make a parameter optional, we need to supply a default value for it. Optional parameters must come after all the required parameters in the function definition:

```
def make_greeting(title, name, surname, formal=True):
    if formal:
        return "Hello, %s %s!" % (title, surname)

    return "Hello, %s!" % name

print(make_greeting("Mr", "John", "Smith"))
print(make_greeting("Mr", "John", "Smith", False))
```

When we call the function, we can leave the optional parameter out – if we do, the default value will be used. If we include the parameter, our value will override the default value.

We can define multiple optional parameters:

```
def make_greeting(title, name, surname, formal=True, time=None):
    if formal:
        fullname =  "%s %s" % (title, surname)
    else:
        fullname = name

    if time is None:
        greeting = "Hello"
    else:
        greeting = "Good %s" % time

    return "%s, %s!" % (greeting, fullname)

print(make_greeting("Mr", "John", "Smith"))
print(make_greeting("Mr", "John", "Smith", False))
print(make_greeting("Mr", "John", "Smith", False, "evening"))
```

What if we want to pass in the *second* optional parameter, but not the *first*? So far we have been passing *positional* parameters to all these functions – a tuple of values which are matched up with parameters in the function signature based on their *positions*. We can also, however, pass these values in as *keyword* parameters – we can explicitly specify the parameter names along with the values:

```
print(make_greeting(title="Mr", name="John", surname="Smith"))
print(make_greeting(title="Mr", name="John", surname="Smith", formal=False, time="evening"))
```

We can mix positional and keyword parameters, but the keyword parameters must come *after* any positional parameters:

```
# this is OK
print(make_greeting("Mr", "John", surname="Smith"))
# this will give you an error
print(make_greeting(title="Mr", "John", "Smith"))
```

We can specify keyword parameters in any order – they don't have to match the order in the function definition:

```
print(make_greeting(surname="Smith", name="John", title="Mr"))
```

Now we can easily pass in the second optional parameter and not the first:

```
print(make_greeting("Mr", "John", "Smith", time="evening"))
```

## Mutable types and default parameters

We should be careful when using mutable types as default parameter values in function definitions if we intend to modify them in-place:

```
def add_pet_to_list(pet, pets=[]):
    pets.append(pet)
    return pets

list_with_cat = add_pet_to_list("cat")
list_with_dog = add_pet_to_list("dog")

print(list_with_cat)
print(list_with_dog) # oops
```

Remember that although we can execute a function *body* many times, a function *definition* is executed only once – that means that the empty list which is created in this function definition will be the same list for all instances of the function. What we really want to do in this case is to create an empty list inside the function body:

```
def add_pet_to_list(pet, pets=None):
    if pets is None:
        pets = []
    pets.append(pet)
    return pets
```

## Exercise 4

1. Write a function called `calculator`. It should take the following parameters: two numbers, an arithmetic operation (which can be addition, subtraction, multiplication or division and is addition by default), and an output format (which can be integer or floating point, and is floating point by default). Division should be floating-point division.

   The function should perform the requested operation on the two input numbers, and return a result in the requested format (if the format is integer, the result should be rounded and not just truncated). Raise exceptions as appropriate if any of the parameters passed to the function are invalid.

2. Call the function with the following sets of parameters, and check that the answer is what you expect:

   (a) `2`, `3.0`

   (b) `2`, `3.0`, output format is integer

   (c) `2`, `3.0`, operation is division

   (d) `2`, `3.0`, operation is division, output format is integer

## *args and **kwargs

Sometimes we may want to pass a variable-length list of positional or keyword parameters into a function. We can put * before a parameter name to indicate that it is a variable-length tuple of positional parameters, and we can use ** to indicate that a parameter is a variable-length dictionary of keyword parameters. By convention, the parameter name we use for the tuple is `args` and the name we use for the dictionary is `kwargs`:

```
def print_args(*args):
    for arg in args:
        print(arg)

def print_kwargs(**kwargs):
    for k, v in kwargs.items():
        print("%s: %s" % (k, v))
```

Inside the function, we can access `args` as a normal tuple, but the * means that `args` isn't passed into the function as a single parameter which is a tuple: instead, it is passed in as a series of individual parameters. Similarly, ** means that `kwargs` is passed in as a series of individual keyword parameters, rather than a single parameter which is a dictionary:

```
print_args("one", "two", "three")
print_args("one", "two", "three", "four")

print_kwargs(name="Jane", surname="Doe")
print_kwargs(age=10)
```

We can use * or ** when we are *calling* a function to *unpack* a sequence or a dictionary into a series of individual parameters:

```
my_list = ["one", "two", "three"]
print_args(*my_list)

my_dict = {"name": "Jane", "surname": "Doe"}
print_kwargs(**my_dict)
```

This makes it easier to build lists of parameters programmatically. Note that we can use this for *any* function, not just one which uses `*args` or `**kwargs`:

```
my_dict = {
    "title": "Mr",
    "name": "John",
    "surname": "Smith",
    "formal": False,
    "time": "evening",
}

print(make_greeting(**my_dict))
```

We can mix ordinary parameters, `*args` and `**kwargs` in the same function definition. `*args` and `**kwargs` must come after all the other parameters, and `**kwargs` must come after `*args`. You cannot have more than one variable-length list parameter or more than one variable dict parameter (recall that you can call them whatever you like):

```
def print_everything(name, time="morning", *args, **kwargs):
    print("Good %s, %s." % (time, name))

    for arg in args:
        print(arg)
    for k, v in kwargs.items():
        print("%s: %s" % (k, v))
```

If we use a * expression when you call a function, it must come after all the positional parameters, and if we use a ** expression it must come right at the end:

```
def print_everything(*args, **kwargs):
    for arg in args:
        print(arg)

    for k, v in kwargs.items():
        print("%s: %s" % (k, v))

# we can write all the parameters individually
print_everything("cat", "dog", day="Tuesday")

t = ("cat", "dog")
d = {"day": "Tuesday"}

# we can unpack a tuple and a dictionary
print_everything(*t, **d)
# or just one of them
print_everything(*t, day="Tuesday")
print_everything("cat", "dog", **d)

# we can mix * and ** with explicit parameters
print_everything("Jane", *t, **d)
print_everything("Jane", *t, time="evening", **d)
print_everything(time="evening", *t, **d)

# none of these are allowed:
print_everything(*t, "Jane", **d)
print_everything(*t, **d, time="evening")
```

If a function takes only `*args` and `**kwargs` as its parameters, it can be called with *any set of parameters*. One or both of `args` and `kwargs` can be empty, so the function will accept any combination of positional and keyword parameters, including no parameters at all. This can be useful if we are writing a very generic function, like `print_everything` in the example above.

### Exercise 5

1. Rewrite the calculator function from exercise 4 so that it takes any number of number parameters as well as the same optional keyword parameters. The function should apply the operation to the first two numbers, and then apply it again to the result and the next number, and so on. For example, if the numbers are `6`, `4`, `9` and `1` and the operation is subtraction the function should return `6 - 4 - 9 - 1`. If only one number is entered, it should be returned unmodified. If no numbers are entered, raise an exception.

## Decorators

Sometimes we may need to modify several functions in the same way – for example, we may want to perform a particular action before and after executing each of the functions, or pass in an extra parameter, or convert the output to another format.

We may also have good reasons not to write the modification into all the functions – maybe it would make the function definitions very verbose and unwieldy, and maybe we would like the option to apply the modification quickly and easily to any function (and remove it just as easily).

To solve this problem, we can write a function which modifies functions. We call a function like this a *decorator*. Our function will take a function object as a parameter, and will return a new function object – we can then assign the new function value to the old function's name to replace the old function with the new function. For example, here is a decorator which logs the function name and its arguments to a log file whenever the function is used:

```
# we define a decorator
def log(original_function):
    def new_function(*args, **kwargs):
        with open("log.txt", "w") as logfile:
            logfile.write("Function '%s' called with positional arguments %s and keyword arguments %

        return original_function(*args, **kwargs)

    return new_function

# here is a function to decorate
def my_function(message):
    print(message)

# and here is how we decorate it
my_function = log(my_function)
```

Inside our decorator (the outer function) we define a replacement function and return it. The replacement function (the inner function) writes a log message and then simply calls the original function and returns its value.

Note that the decorator function is only called once, when we replace the original function with the decorated function, but that the inner function will be called every time we use `my_function`. The inner function can access both variables in its own scope (like `args` and `kwargs`) and variables in the decorator's scope (like `original_function`).

Because the inner function takes `*args` and `**kwargs` as its parameters, we can use this decorator to decorate any function, no matter what its parameter list is. The inner function accepts any parameters, and simply passes them to the original function. We will still get an error inside the original function if we pass in the wrong parameters.

There is a shorthand syntax for applying decorators to functions: we can use the `@` symbol together with the decorator name before the definition of each function that we want to decorate:

```
@log
def my_function(message):
    print(message)
```

`@log` before the function definition means exactly the same thing as `my_function = log(my_function)` after the function definition.

We can pass additional parameters to our decorator. For example, we may want to specify a custom log file to use in our logging decorator:

```
def log(original_function, logfilename="log.txt"):
    def new_function(*args, **kwargs):
        with open(logfilename, "w") as logfile:
            logfile.write("Function '%s' called with positional arguments %s and keyword arguments %

        return original_function(*args, **kwargs)

    return new_function

@log("someotherfilename.txt")
```

```
def my_function(message):
    print(message)
```

Python has several built-in decorators which are commonly used to decorate class methods. We will learn about them in the next chapter.

**Note:** A decorator doesn't have to be a function – it can be any callable object. Some people prefer to write decorators as classes.

### Exercise 6

1. Rewrite the `log` decorator example so that the decorator logs both the function name and parameters and the returned result.
2. Test the decorator by applying it to a function which takes two arguments and returns their sum. Print the result of the function, and what was logged to the file.

## Lambdas

We have already seen that when we want to use a number or a string in our program we can either write it as a *literal* in the place where we want to use it or use a *variable* that we have already defined in our code. For example, `print("Hello!")` prints the literal string `"Hello!"`, which we haven't stored in a variable anywhere, but `print(message)` prints whatever string is stored in the variable `message`.

We have also seen that we can store a function in a variable, just like any other object, by referring to it by its name (but not calling it). Is there such a thing as a function literal? Can we define a function on the fly when we want to pass it as a parameter or assign it to a variable, just like we did with the string `"Hello!"`?

The answer is *yes*, but only for very simple functions. We can use the `lambda` keyword to define anonymous, one-line functions *inline* in our code:

```
a = lambda: 3

# is the same as

def a():
    return 3
```

Lambdas can take parameters – they are written between the `lambda` keyword and the colon, without brackets. A lambda function may only contain a single expression, and the result of evaluating this expression is implicitly returned from the function (we don't use the `return` keyword):

```
b = lambda x, y: x + y

# is the same as

def b(x, y):
    return x + y
```

Lambdas should only be used for very simple functions. If your lambda starts looking too complicated to be readable, you should rather write it out in full as a normal, named function.

### Exercise 7

1. Define the following functions as lambdas, and assign them to variables:
   (a) Take one parameter; return its square
   (b) Take two parameters; return the square root of the sums of their squares
   (c) Take any number of parameters; return their average
   (d) Take a string parameter; return a string which contains the unique letters in the input string (in any order)
2. Rewrite all these functions as named functions.

## Generator functions and `yield`

We have already encountered generators – sequences in which new elements are generated as they are needed, instead of all being generated up-front. We can create our own generators by writing functions which make use of the `yield` statement.

Consider this simple function which returns a range of numbers as a list:

```
def my_list(n):
    i = 0
    l = []

    while i < n:
        l.append(i)
        i += 1

    return l
```

This function builds the full list of numbers and returns it. We can change this function into a generator function while preserving a very similar syntax, like this:

```
def my_gen(n):
    i = 0

    while i < n:
        yield i
        i += 1
```

The first important thing to know about the `yield` statement is that if we use it in a function, that function will return a generator. We can test this by using the `type` function on the return value of `my_gen`. We can also try using it in a `for` loop, like we would use any other generator, to see what sequence the generator represents:

```
g = my_gen(3)

print(type(g))

for x in g:
    print(x)
```

What does the `yield` statement do? Whenever a new value is requested from the generator, for example by our `for` loop in the example above, the generator begins to execute the function until it reaches the `yield` statement. The `yield` statement causes the generator to return a single value.

After the `yield` statement is executed, execution of the function does not end – when the *next* value is requested from the generator, it will go back to the beginning of the function and execute it *again*.

If the generator executes the entire function without encountering a yield statement, it will raise a StopIteration exception to indicate that there are no more values. A for loop automatically handles this exception for us. In our my_gen function this will happen when i becomes equal to n – when this happens, the yield statement inside the while loop will no longer be executed.

### Exercise 8

1. Write a generator function which takes an integer n as a parameter. The function should return a generator which counts *down* from n to 0. Test your function using a for loop.

## Answers to exercises

### Answer to exercise 1

Here is an example program:

```
def func_a():
    print("This is my awesome function.")

func_a()

b = func_a

b()
```

### Answer to exercise 2

Here is an example program:

```
import math

def hypotenuse(x, y):
    print(math.sqrt(x**2 + y**2))

hypotenuse(12.3, 45.6)
hypotenuse(12, 34)
hypotenuse(12, 34.5)
```

### Answer to exercise 3

Here is an example program:

```
import math

def hypotenuse(x, y):
    try:
        return math.sqrt(x**2 + y**2)
    except TypeError:
        return None

print(hypotenuse(12, 34))
print(hypotenuse("12", "34"))
print(hypotenuse(12, "34"))
```

## Answer to exercise 3

1. Here is an example program:

```
def factorial(n):
    ni = int(n)

    if ni != n or ni <= 0:
        raise ValueError("%s is not a positive integer." % n)

    if ni == 1:
        return 1

    return ni * factorial(ni - 1)
```

## Answer to exercise 4

1. Here is an example program:

```
import math

ADD, SUB, MUL, DIV = range(4)

def calculator(a, b, operation=ADD, output_format=float):
    if operation == ADD:
        result = a + b
    elif operation == SUB:
        result = a - b
    elif operation == MUL:
        result = a * b
    elif operation == DIV:
        result = a / b
    else:
        raise ValueError("Operation must be ADD, SUB, MUL or DIV.")

    if output_format == float:
        result = float(result)
    elif output_format == int:
        result = math.round(result)
    else:
        raise ValueError("Format must be float or int.")

    return result
```

2. You should get the following results:

   (a) `5.0`

   (b) `5`

   (c) `0.6666666666666666`

   (d) `1`

## Answer to exercise 5

1. Here is an example program:

```python
import math

ADD, SUB, MUL, DIV = range(4)

def calculator(operation=ADD, output_format=float, *args):
    if not args:
        raise ValueError("At least one number must be entered.")

    result = args[0]

    for n in args[1:]:
        if operation == ADD:
            result += n
        elif operation == SUB:
            result -= n
        elif operation == MUL:
            result *= n
        elif operation == DIV:
            result /= n
        else:
            raise ValueError("Operation must be ADD, SUB, MUL or DIV.")

    if output_format == float:
        result = float(result)
    elif output_format == int:
        result = math.round(result)
    else:
        raise ValueError("Format must be float or int.")

    return result
```

## Answer to exercise 6

1. Here is an example program:

```python
def log(original_function, logfilename="log.txt"):
    def new_function(*args, **kwargs):
        result = original_function(*args, **kwargs)

        with open(logfilename, "w") as logfile:
            logfile.write("Function '%s' called with positional arguments %s and keyword argumen

        return result

    return new_function
```

2. Here is an example program:

```python
@log
def add(x, y):
    return x + y

print(add(3.5, 7))
```

```
with open("log.txt", "r") as logfile:
    print(logfile.read())
```

## Answer to exercise 7

1. Here is an example program:

```
import math

a = lambda x: x**2
b = lambda x, y: math.sqrt(x**2 + y**2)
c = lambda *args: sum(args)/len(args)
d = lambda s: "".join(set(s))
```

2. Here is an example program:

```
import math

def a(x):
    return x**2

def b(x, y):
    return math.sqrt(x**2 + y**2)

def c(*args):
    return sum(args)/len(args)

def d(s):
    return "".join(set(s))
```

## Answer to exercise 8

1. Here is an example program:

```
def my_gen(n):
    i = n

    while i >= 0:
        yield i
        i -= 1

for x in my_gen(3):
    print(x)
```

CHAPTER 9

# Classes

We have already seen how we can use a dictionary to group related data together, and how we can use functions to create shortcuts for commonly used groups of statements. A function performs an action using some set of input parameters. Not all functions are applicable to all kinds of data. *Classes* are a way of grouping together related data *and* functions which act upon that data.

A class is a kind of data type, just like a string, integer or list. When we create an object of that data type, we call it an *instance* of a class.

As we have already mentioned, in some other languages some entities are objects and some are not. In Python, everything is an object – everything is an instance of some class. In earlier versions of Python a distinction was made between built-in types and user-defined classes, but these are now completely indistinguishable. Classes and types are themselves objects, and they are of type `type`. You can find out the type of any object using the `type` function:

```
type(any_object)
```

The data values which we store inside an object are called *attributes*, and the functions which are associated with the object are called *methods*. We have already used the methods of some built-in objects, like strings and lists.

When we design our own objects, we have to decide how we are going to group things together, and what our objects are going to represent.

Sometimes we write objects which map very intuitively onto things in the real world. For example, if we are writing code to simulate chemical reactions, we might have `Atom` objects which we can combine to make a `Molecule` object. However, it isn't always necessary, desirable or even possible to make all code objects perfectly analogous to their real-world counterparts.

Sometimes we may create objects which don't have any kind of real-world equivalent, just because it's useful to group certain functions together.

## Defining and using a class

Here is an example of a simple custom class which stores information about a person:

```
import datetime # we will use this for date objects

class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate
        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age

person = Person(
    "Jane",
    "Doe",
    datetime.date(1992, 3, 12), # year, month, day
    "No. 12 Short Street, Greenville",
    "555 456 0987",
    "jane.doe@example.com"
)

print(person.name)
print(person.email)
print(person.age())
```

We start the class definition with the `class` keyword, followed by the class name and a colon. We would list any parent classes in between round brackets before the colon, but this class doesn't have any, so we can leave them out.

Inside the class body, we define two functions – these are our object's methods. The first is called `__init__`, which is a special method. When we call the class object, a new instance of the class is created, and the `__init__` method on this new object is immediately executed with all the parameters that we passed to the class object. The purpose of this method is thus to set up a new object using data that we have provided.

The second method is a custom method which calculates the age of our person using the birthdate and the current date.

**Note:** `__init__` is sometimes called the object's *constructor*, because it is used similarly to the way that constructors are used in other languages, but that is not technically correct – it's better to call it the *initialiser*. There is a different method called `__new__` which is more analogous to a constructor, but it is hardly ever used.

You may have noticed that both of these method definitions have `self` as the first parameter, and we use this variable inside the method bodies – but we don't appear to pass this parameter in. This is because whenever we call a method on an object, *the object itself* is automatically passed in as the first parameter. This gives us a way to access the object's properties from inside the object's methods.

In some languages this parameter is *implicit* – that is, it is not visible in the function signature – and we access it with a special keyword. In Python it is explicitly exposed. It doesn't have to be called `self`, but this is a very strongly followed convention.

Now you should be able to see that our `__init__` function creates attributes on the object and sets them to the values we have passed in as parameters. We use the same names for the attributes and the parameters, but this is not compulsory.

The `age` function doesn't take any parameters except `self` – it only uses information stored in the object's attributes, and the current date (which it retrieves using the `datetime` module).

Note that the `birthdate` attribute is itself an object. The `date` class is defined in the `datetime` module, and we create a new instance of this class to use as the birthdate parameter when we create an instance of the `Person` class.

We don't have to assign it to an intermediate variable before using it as a parameter to `Person`; we can just create it when we call `Person`, just like we create the string literals for the other parameters.

Remember that defining a function doesn't make the function run. Defining a class also doesn't make anything run – it just tells Python about the class. The class will not be defined until Python has executed the entirety of the definition, so you can be sure that you can reference any method from any other method on the same class, or even reference the class inside a method of the class. By the time you call that method, the entire class will definitely be defined.

### Exercise 1

1. Explain what the following variables refer to, and their scope:
   (a) `Person`
   (b) `person`
   (c) `surname`
   (d) `self`
   (e) `age` (the function name)
   (f) `age` (the variable used inside the function)
   (g) `self.email`
   (h) `person.email`

## Instance attributes

It is important to note that the attributes set on the object in the `__init__` function do not form an exhaustive list of all the attributes that our object is ever allowed to have.

In some languages you must provide a list of the object's attributes in the class definition, placeholders are created for these allowed attributes when the object is created, and you may not add new attributes to the object later. In Python, you can add new attributes, and even new methods, to an object on the fly. In fact, there is nothing special about the `__init__` function when it comes to setting attributes. We could store a cached age value on the object from inside the `age` function:

```python
def age(self):
    if hasattr(self, "_age"):
        return self._age

    today = datetime.date.today()

    age = today.year - self.birthdate.year

    if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
        age -= 1

    self._age = age
    return age
```

**Note:** Starting an attribute or method name with an underscore (_) is a convention which we use to indicate that it is a "private" internal property and should not be accessed directly. In a more realistic example, our cached value would sometimes expire and need to be recalculated – so we should always use the `age` method to make sure that we get the right value.

We could even add a completely unrelated attribute from outside the object:

```
person.pets = ['cat', 'cat', 'dog']
```

It is very common for an object's methods to *update* the values of the object's attributes, but it is considered bad practice to *create* new attributes in a method without initialising them in the `__init__` method. Setting arbitrary properties from outside the object is frowned upon even more, since it breaks the object-oriented paradigm (which we will discuss in the next chapter).

The `__init__` method will definitely be executed before anything else when we create the object – so it's a good place to do all of our initialisation of the object's data. If we create a new attribute outside the `__init__` method, we run the risk that we will try to use it before it has been initialised.

In the `age` example above we have to check if an `_age` attribute exists on the object before we try to use it, because if we haven't run the `age` method before it will not have been created yet. It would be much tidier if we called this method at least once from `__init__`, to make sure that `_age` is created as soon as we create the object.

Initialising all our attributes in `__init__`, even if we just set them to empty values, makes our code less error-prone. It also makes it easier to read and understand – we can see at a glance what attributes our object has.

An `__init__` method doesn't have to take any parameters (except `self`) and it can be completely absent.

## `getattr`, `setattr` and `hasattr`

What if we want to get or set the value of an attribute of an object without hard-coding its name? We may sometimes want to loop over several attribute names and perform the same operation on all of them, as we do in this example which uses a dictionary:

```
for key in ["a", "b", "c"]:
    print(mydict[key])
```

How can we do something similar with an object? We can't use the `.` operator, because it must be followed by the attribute name as a bare word. If our attribute name is stored as a string value in a variable, we have to use the `getattr` function to retrieve the attribute value from an object:

```
for key in ["a", "b", "c"]:
    print(getattr(myobject, key, None))
```

Note that `getattr` is a built-in function, not a method on the object: it takes the object as its first parameter. The second parameter is the name of the variable as a string, and the optional third parameter is the default value to be returned if the attribute does not exist. If we do not specify a default value, `getattr` will raise an exception if the attribute does not exist.

Similarly, `setattr` allows us to set the value of an attribute. In this example, we copy data from a dictionary to an object:

```
for key in ["a", "b", "c"]:
    setattr(myobject, key, mydict[key])
```

The first parameter of `setattr` is the object, the second is the name of the function, and the third is the new value for the attribute.

As we saw in the previous `age` function example, `hasattr` detects whether an attribute exists.

There's nothing preventing us from using `getattr` on attributes even if the name can be hard-coded, but this is not recommended: it's an unnecessarily verbose and round-about way of accessing attributes:

```
getattr(myobject, "a")

# means the same thing as

myobject.a
```

You should only use these functions if you have a good reason to do so.

### Exercise 2

1. Rewrite the `Person` class so that a person's age is calculated for the first time when a new person instance is created, and recalculated (when it is requested) if the day has changed since the last time that it was calculated.

## Class attributes

All the attributes which are defined on a `Person` instance are *instance attributes* – they are added to the instance when the `__init__` method is executed. We can, however, also define attributes which are set on the *class*. These attributes will be shared by all instances of that class. In many ways they behave just like instance attributes, but there are some caveats that you should be aware of.

We define class attributes in the body of a class, at the same indentation level as method definitions (one level up from the insides of methods):

```
class Person:

    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, title, name, surname):
        if title not in self.TITLES:
            raise ValueError("%s is not a valid title." % title)

        self.title = title
        self.name = name
        self.surname = surname
```

As you can see, we access the class attribute `TITLES` just like we would access an instance attribute – it is made available as a property on the instance object, which we access inside the method through the `self` variable.

All the `Person` objects we create will share the same `TITLES` class attribute.

Class attributes are often used to define constants which are closely associated with a particular class. Although we can use class attributes from class instances, we can also use them from class objects, without creating an instance:

```
# we can access a class attribute from an instance
person.TITLES

# but we can also access it from the class
Person.TITLES
```

Note that the class object doesn't have access to any *instance* attributes – those are only created when an instance is created!

```
# This will give us an error
Person.name
Person.surname
```

Class attributes can also sometimes be used to provide default attribute values:

```
class Person:
    deceased = False

    def mark_as_deceased(self):
        self.deceased = True
```

When we set an attribute on an instance which has the same name as a class attribute, we are *overriding* the class attribute with an instance attribute, which will take precedence over it. If we create two `Person` objects and call the `mark_as_deceased` method on one of them, we will not affect the other one. We should, however, be careful when a class attribute is of a mutable type – because if we modify it in-place, we *will* affect all objects of that class at the same time. Remember that all instances share the same class attributes:

```
class Person:
    pets = []

    def add_pet(self, pet):
        self.pets.append(pet)

jane = Person()
bob = Person()

jane.add_pet("cat")
print(jane.pets)
print(bob.pets) # oops!
```

What we *should* do in cases like this is initialise the mutable attribute *as an instance attribute*, inside `__init__`. Then every instance will have its own separate copy:

```
class Person:

    def __init__(self):
        self.pets = []

    def add_pet(self, pet):
        self.pets.append(pet)

jane = Person()
bob = Person()

jane.add_pet("cat")
print(jane.pets)
print(bob.pets)
```

Note that method definitions are in the same scope as class attribute definitions, so we can use class attribute names as variables in method definitions (without `self`, which is only defined *inside* the methods):

```
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, title, name, surname, allowed_titles=TITLES):
        if title not in allowed_titles:
            raise ValueError("%s is not a valid title." % title)

        self.title = title
        self.name = name
        self.surname = surname
```

Can we have class *methods*? Yes, we can. In the next section we will see how to define them using a decorator.

### Exercise 3

1. Explain the differences between the attributes `name`, `surname` and `profession`, and what values they can have in different instances of this class:

```
class Smith:
    surname = "Smith"
    profession = "smith"

    def __init__(self, name, profession=None):
        self.name = name
        if profession is not None:
            self.profession = profession
```

## Class decorators

In the previous chapter we learned about decorators – functions which are used to modify the behaviour of other functions. There are some built-in decorators which are often used in class definitions.

### `@classmethod`

Just like we can define class *attributes*, which are shared between all instances of a class, we can define class *methods*. We do this by using the `@classmethod` decorator to decorate an ordinary method.

A class method still has its calling object as the first parameter, but by convention we rename this parameter from `self` to `cls`. If we call the class method from an instance, this parameter will contain the instance object, but if we call it from the class it will contain the class object. By calling the parameter `cls` we remind ourselves that it is not guaranteed to have any *instance* attributes.

What are class methods good for? Sometimes there are tasks associated with a class which we can perform using constants and other class attributes, without needing to create any class instances. If we had to use instance methods for these tasks, we would need to create an instance for no reason, which would be wasteful. Sometimes we write classes purely to group related constants together with functions which act on them – we may never instantiate these classes at all.

Sometimes it is useful to write a class method which creates an instance of the class after processing the input so that it is in the right format to be passed to the class constructor. This allows the constructor to be straightforward and not have to implement any complicated parsing or clean-up code:

```
class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        # (...)

    @classmethod
    def from_text_file(cls, filename):
        # extract all the parameters from the text file
        return cls(*params) # this is the same as calling Person(*params)
```

### `@staticmethod`

A static method doesn't have the calling object passed into it as the first parameter. This means that it doesn't have access to the rest of the class or instance at all. We can call them from an instance or a class object, but they are most commonly called from class objects, like class methods.

If we are using a class to group together related methods which don't need to access each other or any other data on the class, we may want to use this technique. The advantage of using static methods is that we eliminate unnecessary `cls` or `self` parameters from our method definitions. The disadvantage is that if we do occasionally want to refer to another class method or attribute inside a static method we have to write the class name out in full, which can be much more verbose than using the `cls` variable which is available to us inside a class method.

Here is a brief example comparing the three method types:

```python
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

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

    def fullname(self): # instance method
        # instance object accessible through self
        return "%s %s" % (self.name, self.surname)

    @classmethod
    def allowed_titles_starting_with(cls, startswith): # class method
        # class or instance object accessible through cls
        return [t for t in cls.TITLES if t.startswith(startswith)]

    @staticmethod
    def allowed_titles_ending_with(endswith): # static method
        # no parameter for class or instance object
        # we have to use Person directly
        return [t for t in Person.TITLES if t.endswith(endswith)]


jane = Person("Jane", "Smith")

print(jane.fullname())

print(jane.allowed_titles_starting_with("M"))
print(Person.allowed_titles_starting_with("M"))

print(jane.allowed_titles_ending_with("s"))
print(Person.allowed_titles_ending_with("s"))
```

## `@property`

Sometimes we use a method to generate a property of an object dynamically, calculating it from the object's other properties. Sometimes you can simply use a method to access a single attribute and return it. You can also use a different method to update the value of the attribute instead of accessing it directly. Methods like this are called *getters* and *setters*, because they "get" and "set" the values of attributes, respectively.

In some languages you are encouraged to use getters and setters for all attributes, and never to access their values directly – and there are language features which can make attributes inaccessible except through setters and getters. In Python, accessing simple attributes directly is perfectly acceptable, and writing getters and setters for all of them is considered unnecessarily verbose. Setters can be inconvenient because they don't allow use of compound assignment operators:

```python
class Person:
    def __init__(self, height):
        self.height = height

    def get_height(self):
        return self.height

    def set_height(self, height):
        self.height = height

jane = Person(153) # Jane is 153cm tall

jane.height += 1 # Jane grows by a centimetre
jane.set_height(jane.height + 1) # Jane grows again
```

As we can see, incrementing the height attribute through a setter is much more verbose. Of course we could write a *second* setter which increments the attribute by the given parameter – but we would have to do something similar for every attribute and every kind of modification that we want to perform. We would have a similar issue with in-place modifications, like adding values to lists.

Something which is often considered an *advantage* of setters and getters is that we can change the way that an attribute is generated inside the object without affecting any code which uses the object. For example, suppose that we initially created a `Person` class which has a `fullname` attribute, but later we want to change the class to have separate `name` and `surname` attributes which we combine to create a full name. If we always access the `fullname` attribute through a setter, we can just rewrite the setter – none of the code which calls the setter will have to be changed.

But what if our code accesses the `fullname` attribute directly? We can write a `fullname` method which returns the right value, but a method has to be *called*. Fortunately, the `@property` decorator lets us make a method behave like an attribute:

```python
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    @property
    def fullname(self):
        return "%s %s" % (self.name, self.surname)

jane = Person("Jane", "Smith")
print(jane.fullname) # no brackets!
```

There are also decorators which we can use to define a setter and a deleter for our attribute (a deleter will delete the attribute from our object). The getter, setter and deleter methods must all have the same name:

```python
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    @property
    def fullname(self):
        return "%s %s" % (self.name, self.surname)

    @fullname.setter
    def fullname(self, value):
        # this is much more complicated in real life
        name, surname = value.split(" ", 1)
        self.name = name
        self.surname = surname

    @fullname.deleter
    def fullname(self):
        del self.name
        del self.surname

jane = Person("Jane", "Smith")
print(jane.fullname)

jane.fullname = "Jane Doe"
print(jane.fullname)
print(jane.name)
print(jane.surname)
```

### Exercise 4

1. Create a class called `Numbers`, which has a single class attribute called `MULTIPLIER`, and a constructor which takes the parameters `x` and `y` (these should all be numbers).
   (a) Write a method called `add` which returns the sum of the attributes `x` and `y`.
   (b) Write a class method called `multiply`, which takes a single number parameter `a` and returns the product of `a` and `MULTIPLIER`.
   (c) Write a static method called `subtract`, which takes two number parameters, `b` and `c`, and returns `b - c`.
   (d) Write a method called `value` which returns a tuple containing the values of `x` and `y`. Make this method into a property, and write a setter and a deleter for manipulating the values of `x` and `y`.

## Inspecting an object

We can check what properties are defined on an object using the `dir` function:

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def fullname(self):
        return "%s %s" % (self.name, self.surname)

jane = Person("Jane", "Smith")

print(dir(jane))
```

Now we can see our attributes and our method – but what's all that other stuff? We will discuss *inheritance* in the next chapter, but for now all you need to know is that any class that you define has `object` as its parent class even if you don't explicitly say so – so your class will have a lot of default attributes and methods that any Python object has.

**Note:** in Python 2 we have to inherit from `object` explicitly, otherwise our class will be almost completely empty except for our own custom properties. Classes which don't inherit from `object` are called "old-style classes", and using them is not recommended. If we were to write the person class in Python 2 we would write the first line as `class Person(object):`.

This is why you can just leave out the `__init__` method out of your class if you don't have any initialisation to do – the default that you inherited from `object` (which does nothing) will be used instead. If you do write your own `__init__` method, it will *override* the default method. Sometimes we also call this *overloading*.

Many default methods and attributes that are found in built-in Python objects have names which begin and end in double underscores, like `__init__` or `__str__`. These names indicate that these properties have a special meaning – you shouldn't create your own methods or attributes with the same names unless you mean to overload them. These properties are usually methods, and they are sometimes called *magic methods*.

We can use `dir` on any object. You can try to use it on all kinds of objects which we have already seen before, like numbers, lists, strings and functions, to see what built-in properties these objects have in common.

Here are some examples of special object properties:

- `__init__`: the initialisation method of an object, which is called when the object is created.
- `__str__`: the string representation method of an object, which is called when you use the `str` function to convert that object to a string.
- `__class__`: an attribute which stores the the class (or type) of an object – this is what is returned when you use the `type` function on the object.
- `__eq__`: a method which determines whether this object is equal to another. There are also other methods for determining if it's not equal, less than, etc.. These methods are used in object comparisons, for example when we use the equality operator == to check if two objects are equal.
- `__add__` is a method which allows this object to be added to another object. There are equivalent methods for all the other arithmetic operators. Not all objects support all arithemtic operations – numbers have all of these methods defined, but other objects may only have a subset.
- `__iter__`: a method which returns an iterator over the object – we will find it on strings, lists and other iterables. It is executed when we use the `iter` function on the object.
- `__len__`: a method which calculates the length of an object – we will find it on sequences. It is executed when we use the `len` function of an object.
- `__dict__`: a dictionary which contains all the instance attributes of an object, with their names as keys. It can be useful if we want to iterate over all the attributes of an object. `__dict__` does not include any methods, class attributes or special default attributes like `__class__`.

### Exercise 5

1. Create an instance of the `Person` class from example 2. Use the `dir` function on the instance. Then use the `dir` function on the class.
   (a) What happens if you call the `__str__` method on the instance? Verify that you get the same result if you call the `str` function with the instance as a parameter.
   (b) What is the type of the instance?
   (c) What is the type of the class?
   (d) Write a function which prints out the names and values of all the custom attributes of any object that is passed in as a parameter.

## Overriding magic methods

We have already seen how to overload the `__init__` method so that we can customise it to initialise our class. We can also overload other special methods. For example, the purpose of the `__str__` method is to output a useful string representation of our object. but by default if we use the `str` function on a person object (which will call the `__str__` method), all that we will get is the class name and an ID. That's not very useful! Let's write a custom `__str__` method which shows the values of all of the object's properties:

```python
import datetime

class Person:
    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def __str__(self):
        return "%s %s, born %s\nAddress: %s\nTelephone: %s\nEmail:%s" % (self.name, self.surname, sel

jane = Person(
    "Jane",
    "Doe",
    datetime.date(1992, 3, 12), # year, month, day
    "No. 12 Short Street, Greenville",
    "555 456 0987",
    "jane.doe@example.com"
)

print(jane)
```

Note that when we insert the birthdate object into the output string with `%s` it will itself be converted to a string, so we don't need to do it ourselves (unless we want to change the format).

It is also often useful to overload the comparison methods, so that we can use comparison operators on our person objects. By default, our person objects are only equal if they are the same object, and you can't test whether one person object is greater than another because person objects have no default order.

Suppose that we want our person objects to be equal if all their attributes have the same values, and we want to be able to order them alphabetically by surname and then by first name. All of the magic comparison methods are independent of each other, so we will need to overload all of them if we want all of them to work – but fortunately once we have defined equality and one of the basic order methods the rest are easy to do. Each of these methods takes two parameters – `self` for the current object, and `other` for the other object:

```python
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def __eq__(self, other): # does self == other?
        return self.name == other.name and self.surname == other.surname

    def __gt__(self, other): # is self > other?
        if self.surname == other.surname:
            return self.name > other.name
        return self.surname > other.surname

    # now we can define all the other methods in terms of the first two

    def __ne__(self, other): # does self != other?
```

```
        return not self == other # this calls self.__eq__(other)

    def __le__(self, other): # is self <= other?
        return not self > other # this calls self.__gt__(other)

    def __lt__(self, other): # is self < other?
        return not (self > other or self == other)

    def __ge__(self, other): # is self >= other?
        return not self < other
```

Note that `other` is not guaranteed to be another person object, and we haven't put in any checks to make sure that it is. Our method will crash if the other object doesn't have a `name` or `surname` attribute, but if they are present the comparison will work. Whether that makes sense or not is something that we will need to think about if we create similar types of objects.

Sometimes it makes sense to exit with an error if the other object is not of the same type as our object, but sometimes we can compare two compatible objects even if they are not of the same type. For example, it makes sense to compare `1` and `2.5` because they are both numbers, even though one is an integer and the other is a float.

**Note:** Python 2 also has a `__cmp__` method which was introduced to the language before the individual comparison methods (called *rich comparisons*) described above. It is used if the rich comparisons are not defined. You should overload it in a way which is consistent with the rich comparison methods, otherwise you may encounter some very strange behaviour.

### Exercise 6

1. Write a class for creating completely generic objects: its `__init__` function should accept any number of keyword parameters, and set them on the object as attributes with the keys as names. Write a `__str__` method for the class – the string it returns should include the name of the class and the values of all the object's custom instance attributes.

## Answers to exercises

### Answer to exercise 1

1. (a) `Person` is a class defined in the global scope. It is a global variable.
   (b) `person` is an instance of the `Person` class. It is also a global variable.
   (c) `surname` is a parameter passed into the `__init__` method – it is a local variable in the scope if the `__init__` method.
   (d) `self` is a parameter passed into each instance method of the class – it will be replaced by the instance object when the method is called on the object with the `.` operator. It is a new local variable inside the scope of each of the methods – it just always has the same value, and by convention it is always given the same name to reflect this.
   (e) `age` is a method of the `Person` class. It is a local variable in the scope of the class.
   (f) `age` (the variable used inside the function) is a local variable inside the scope of the `age` method.
   (g) `self.email` isn't really a separate variable. It's an example of how we can refer to attributes and methods of an object using a variable which refers to the object, the `.` operator and the name of the

attribute or method. We use the `self` variable to refer to an object inside one of the object's own methods – wherever the variable `self` is defined, we can use `self.email`, `self.age()`, etc..

(h) `person.email` is another example of the same thing. In the global scope, our person instance is referred to by the variable name `person`. Wherever `person` is defined, we can use `person.email`, `person.age()`, etc..

## Answer to exercise 2

1. Here is an example program:

```
import datetime

class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

        # This isn't strictly necessary, but it clearly introduces these attributes
        self._age = None
        self._age_last_recalculated = None

        self._recalculate_age()

    def _recalculate_age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        self._age = age
        self._age_last_recalculated = today

    def age(self):
        if (datetime.date.today() > self._age_last_recalculated):
            self._recalculate_age()

        return self._age
```

## Answer to exercise 3

1. `name` is always an instance attribute which is set in the constructor, and each class instance can have a different name value. `surname` is always a class attribute, and cannot be overridden in the constructor – every instance will have a surname value of `Smith`. `profession` is a class attribute, but it can optionally be overridden by an instance attribute in the constructor. Each instance will have a profession value of `smith` unless the optional `surname` parameter is passed into the constructor with a different value.

## Answer to exercise 4

1. Here is an example program:

```
class Numbers:
    MULTIPLIER = 3.5

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self):
        return self.x + self.y

    @classmethod
    def multiply(cls, a):
        return cls.MULTIPLIER * a

    @staticmethod
    def subtract(b, c):
        return b - c

    @property
    def value(self):
        return (self.x, self.y)

    @value.setter
    def value(self, xy_tuple):
        self.x, self.y = xy_tuple

    @value.deleter
    def value(self):
        del self.x
        del self.y
```

## Answer to exercise 5

1. (a) You should see something like `'<__main__.Person object at 0x7fcb233301d0>'`.

   (b) `<class '__main__.Person'>` – `__main__` is Python's name for the program you are executing.

   (c) `<class 'type'>` – any class has the type `type`.

   (d) Here is an example program:

```
def print_object_attrs(any_object):
    for k, v in any_object.__dict__.items():
        print("%s: %s" % (k, v))
```

## Answer to exercise 6

1. Here is an example program:

```
class AnyClass:
    def __init__(self, **kwargs):
        for k, v in kwargs.items():
            setattr(self, k, v)
    def __str__(self):
        attrs = ["%s=%s" % (k, v) for (k, v) in self.__dict__.items()]
        classname = self.__class__.__name__
        return "%s: %s" % (classname, " ".join(attrs))
```

CHAPTER 10

# Object-oriented programming

## Introduction

As you have seen from the earliest code examples in this course, it is not compulsory to organise your code into classes when you program in Python. You can use functions by themselves, in what is called a *procedural* programming approach. However, while a procedural style can suffice for writing short, simple programs, an object-oriented programming (OOP) approach becomes more valuable the more your program grows in size and complexity.

The more data and functions comprise your code, the more important it is to arrange them into logical subgroups, making sure that data and functions which are related are grouped together and that data and functions which are not related don't interfere with each other. Modular code is easier to understand and modify, and lends itself more to reuse – and code reuse is valuable because it reduces development time.

As a worst-case scenario, imagine a program with a hundred functions and a hundred separate global variables all in the same file. This would be a very difficult program to maintain. All the variables could potentially be modified by all the functions even if they shouldn't be, and in order to pick unique names for all the variables, some of which might have a very similar purpose but be used by different functions, we would probably have to resort to poor naming practices. It would probably be easy to confuse these variables with each other, since it would be difficult to see which functions use which variables.

We could try to make this code more modular even without object orientation. We could group related variables together into aggregate data structures. In the past, some other languages, like C++, introduced a `struct` type which eventually became indistinguishable from a class, but which initially didn't have any methods – only attributes. This allowed programmers to construct compound variables out of many individual variables, and was the first step towards object orientation. In Python, we often use dictionaries for ad-hoc grouping of related data.

We could also split up the functions and data into separate *namespaces* instead of having them all defined inside the same *global namespace*. This often coincides with splitting the code physically into multiple files. In Python we do this by splitting code up into *modules*.

The main additional advantage of object orientation, as we saw in the previous chapter, is that it combines data with the functions which act upon that data in a single structure. This makes it easy for us to find related parts of our code, since they are physically defined in close proximity to one another, and also makes it easier for us to write our code in such a way that the data inside each object is accessed as much as possible only through that object's methods. We will discuss this principle, which we call *encapsulation*, in the next section.

Some people believe that OOP is a more intuitive programming style to learn, because people find it easy to reason about objects and relationships between them. OOP is thus sometimes considered to be a superior approach because it allows new programmers to become proficient more quickly.

### Basic OOP principles

The most important principle of object orientation is *encapsulation*: the idea that data inside the object should only be accessed through a public *interface* – that is, the object's methods.

The `age` function we saw in the previous chapter is a good example of this philosophy. If we want to use the data stored in an object to perform an action or calculate a derived value, we define a method associated with the object which does this. Then whenever we want to perform this action we call the method on the object. We consider it bad practice to retrieve the information from inside the object and write separate code to perform the action outside of the object.

Encapsulation is a good idea for several reasons:

- the functionality is defined *in one place* and not in multiple places.
- it is defined in a logical place – the place where the data is kept.
- data inside our object is not modified unexpectedly by external code in a completely different part of our program.
- when we use a method, we only need to know what result the method will produce – we don't need to know details about the object's internals in order to use it. We could switch to using another object which is completely different on the inside, and not have to change any code because both objects have the same interface.

We can say that the object "knows how" to do things with its own data, and it's a bad idea for us to access its internals and do things with the data ourselves. If an object doesn't have an interface method which does what we want to do, we should add a new method or update an existing one.

Some languages have features which allow us to enforce encapsulation strictly. In Java or C++, we can define access permissions on object attributes, and make it illegal for them to be accessed from outside the object's methods. In Java it is also considered good practice to write setters and getters for all attributes, even if the getter simply retrieves the attribute and the setter just assigns it the value of the parameter which you pass in.

In Python, encapsulation is not enforced by the language, but there is a convention that we can use to indicate that a property is intended to be private and is not part of the object's public interface: we begin its name with an underscore.

It is also customary to set and get simple attribute values directly, and only write setter and getter methods for values which require some kind of calculation. In the last chapter we learned how to use the property decorator to replace a simple attribute with a method without changing the object's interface.

### Relationships between objects

In the next section we will look at different ways that classes can be related to each other. In Python, there are two main types of relationships between classes: *composition* and *inheritance*.

## Composition

Composition is a way of *aggregating* objects together by making some objects attributes of other objects. We saw in the previous chapter how we can make a `datetime.date` object an attribute of our `Person` object, and use it to store a person's birthdate. We can say that a person *has a* birthdate – if we can express a relationship between two classes using the phrase *has-a*, it is a composition relationship.

Relationships like this can be one-to-one, one-to-many or many-to-many, and they can be unidirectional or bidirectional, depending on the specifics of the the roles which the objects fulfil.

According to some formal definitions the term *composition* implies that the two objects are quite strongly linked – one object can be thought of as *belonging* exclusively to the other object. If the owner object ceases to exist, the owned object will probably cease to exist as well. If the link between two objects is weaker, and neither object has exclusive ownership of the other, it can also be called *aggregation*.

Here are four classes which show several examples of aggregation and composition:

```
class Student:
    def __init__(self, name, student_number):
        self.name = name
        self.student_number = student_number
        self.classes = []

    def enrol(self, course_running):
        self.classes.append(course_running)
        course_running.add_student(self)


class Department:
    def __init__(self, name, department_code):
        self.name = name
        self.department_code = department_code
        self.courses = {}

    def add_course(self, description, course_code, credits):
        self.courses[course_code] = Course(description, course_code, credits, self)
        return self.courses[course_code]


class Course:
    def __init__(self, description, course_code, credits, department):
        self.description = description
        self.course_code = course_code
        self.credits = credits
        self.department = department
        self.department.add_course(self)

        self.runnings = []

    def add_running(self, year):
        self.runnings.append(CourseRunning(self, year))
        return self.runnings[-1]


class CourseRunning:
    def __init__(self, course, year):
        self.course = course
        self.year = year
        self.students = []

    def add_student(self, student):
        self.students.append(student)


maths_dept = Department("Mathematics and Applied Mathematics", "MAM")
mam1000w = maths_dept.add_course("Mathematics 1000", "MAM1000W", 1)
mam1000w_2013 = mam1000w.add_running(2013)

bob = Student("Bob", "Smith")
bob.enrol(mam1000w_2013)
```

Why are there two classes which both describe a course? This is an example of the way that translation of real-life concepts into objects in your code may not always be as straightforward as it appears. Would it have made sense to have a single course object which has both description, code and department attributes and a list of students?

There are two distinct concepts, both of which can be called a "course", that we need to represent: one is the theoretical *idea* of a course, which is offered by a department every year and always has the same name and code, and the other is the course as it is run *in a particular year*, each time with a different group of enrolled students. We have represented these two concepts by two separate classes which are linked to each other. `Course` is the theoretical description of a course, and `CourseRunning` is the concrete instance of a course.

We have defined several relationships between these classes:

- A student can be enrolled in several courses (`CourseRunning` objects), and a course (`CourseRunning`) can have multiple students enrolled in it in a particular year, so this is a many-to-many relationship. A student knows about all his or her courses, and a course has a record of all enrolled students, so this is a bidirectional relationship. These objects aren't very strongly coupled – a student can exist independently of a course, and a course can exist independently of a student.
- A department offers multiple courses (`Course` objects), but in our implementation a course can only have a single department – this is a one-to-many relationship. It is also bidirectional. Furthermore, these objects are more strongly coupled – you can say that a department *owns* a course. The course cannot exist without the department.
- A similar relationship exists between a course and its "runnings": it is also bidirectional, one-to-many and strongly coupled – it wouldn't make sense for "MAM1000W run in 2013" to exist on its own in the absence of "MAM1000W".

What words like "exist" and "owns" actually mean for our code can vary. An object which "owns" another object could be responsible for creating that object when it requires it and destroying it when it is no longer needed – but these words can also be used to describe a logical relationship between concepts which is not necessarily literally implemented in that way in the code.

## Exercise 1

1. Briefly describe a possible collection of classes which can be used to represent a music collection (for example, inside a music player), focusing on how they would be related by composition. You should include classes for songs, artists, albums and playlists. Hint: write down the four class names, draw a line between each pair of classes which you think should have a relationship, and decide what kind of relationship would be the most appropriate.

   For simplicity you can assume that any song or album has a single "artist" value (which could represent more than one person), but you should include compilation albums (which contain songs by a selection of different artists). The "artist" of a compilation album can be a special value like "Various Artists". You can also assume that each song is associated with a single album, but that multiple copies of the same song (which are included in different albums) can exist.

2. Write a simple implementation of this model which clearly shows how the different classes are composed. Write some example code to show how you would use your classes to create an album and add all its songs to a playlist. Hint: if two objects are related to each other bidirectionally, you will have to decide how this link should be formed – one of the objects will have to be created before the other, so you can't link them to each other in both directions simultaneously!

## Inheritance

*Inheritance* is a way of arranging objects in a hierarchy from the most general to the most specific. An object which *inherits* from another object is considered to be a *subtype* of that object. As we saw in the previous chapter, all objects in Python inherit from `object`. We can say that a string, an integer or a `Person` instance *is an* `object` instance. When we can describe the relationship between two objects using the phrase *is-a*, that relationship is inheritance.

We also often say that a class is a *subclass* or *child class* of a class from which it inherits, or that the other class is its *superclass* or *parent class*. We can refer to the most generic class at the base of a hierarchy as a *base class*.

Inheritance can help us to represent objects which have some differences and some similarities in the way they work. We can put all the functionality that the objects have in common in a base class, and then define one or more subclasses with their own custom functionality.

Inheritance is also a way of reusing existing code easily. If we already have a class which does *almost* what we want, we can create a subclass in which we partially override some of its behaviour, or perhaps add some new functionality.

Here is a simple example of inheritance:

```python
class Person:
    def __init__(self, name, surname, number):
        self.name = name
        self.surname = surname
        self.number = number


class Student(Person):
    UNDERGRADUATE, POSTGRADUATE = range(2)

    def __init__(self, student_type, *args, **kwargs):
        self.student_type = student_type
        self.classes = []
        super(Student, self).__init__(*args, **kwargs)

    def enrol(self, course):
        self.classes.append(course)


class StaffMember(Person):
    PERMANENT, TEMPORARY = range(2)

    def __init__(self, employment_type, *args, **kwargs):
        self.employment_type = employment_type
        super(StaffMember, self).__init__(*args, **kwargs)


class Lecturer(StaffMember):
    def __init__(self, *args, **kwargs):
        self.courses_taught = []
        super(Lecturer, self).__init__(*args, **kwargs)

    def assign_teaching(self, course):
        self.courses_taught.append(course)


jane = Student(Student.POSTGRADUATE, "Jane", "Smith", "SMTJNX045")
jane.enrol(a_postgrad_course)
```

```
bob = Lecturer(StaffMember.PERMANENT, "Bob", "Jones", "123456789")
bob.assign_teaching(an_undergrad_course)
```

Our base class is `Person`, which represents any person associated with a university. We create a subclass to represent students and one to represent staff members, and then a subclass of `StaffMember` for people who teach courses (as opposed to staff members who have administrative positions.)

We represent both student numbers and staff numbers by a single attribute, `number`, which we define in the base class, because it makes sense for us to treat them as a unified form of identification for any person. We use different attributes for the kind of student (undergraduate or postgraduate) that someone is and whether a staff member is a permanent or a temporary employee, because these are different sets of options.

We have also added a method to `Student` for enrolling a student in a course, and a method to `Lecturer` for assigning a course to be taught by a lecturer.

The `__init__` method of the base class initialises all the instance variables that are common to all subclasses. In each subclass we *override* the `__init__` method so that we can use it to initialise that class's attributes – but we want the parent class's attributes to be initialised as well, so we need to call the parent's `__init__` method from ours. To find the right method, we use the `super` function – when we pass in the current class and object as parameters, it will return a proxy object with the correct `__init__` method, which we can then call.

In each of our overridden `__init__` methods we use those of the method's parameters which are specific to our class inside the method, and then pass the remaining parameters to the parent class's `__init__` method. A common convention is to add the specific parameters for each successive subclass to the *beginning* of the parameter list, and define all the other parameters using `*args` and `**kwargs` – then the subclass doesn't need to know the details about the parent class's parameters. Because of this, if we add a new parameter to the superclass's `__init__`, we will only need to add it to all the places where we create that class or one of its subclasses – we won't also have to update all the child class definitions to include the new parameter.

## Exercise 2

1. A very common use case for inheritance is the creation of a custom exception hierarchy. Because we use the class of an exception to determine whether it should be caught by a particular `except` block, it is useful for us to define custom classes for exceptions which we want to raise in our code. Using inheritance in our classes is useful because if an `except` block catches a particular exception class, it will also catch its child classes (because a child class *is* its parent class). That means that we can efficiently write `except` blocks which handle groups of related exceptions, just by arranging them in a logical hierarchy. Our exception classes should inherit from Python's built-in exception classes. They often won't need to contain any additional attributes or methods.

   Write a simple program which loops over a list of user data (tuples containing a username, email and age) and adds each user to a directory if the user is at least 16 years old. You do not need to store the age. Write a simple exception hierarchy which defines a different exception for each of these error conditions:

   (a) the username is not unique

   (b) the age is not a positive integer

   (c) the user is under 16

   (d) the email address is not valid (a simple check for a username, the `@` symbol and a domain name is sufficient)

   Raise these exceptions in your program where appropriate. Whenever an exception occurs, your program should move onto the next set of data in the list. Print a different error message for each different kind of exception.

   Think about where else it would be a good idea to use a custom class, and what kind of collection type would be most appropriate for your directory.

You can consider an email address to be valid if it contains one @ symbol and has a non-empty username and domain name – you don't need to check for valid characters. You can assume that the age is already an integer value.

## More about inheritance

### Multiple inheritance

The previous example might seem like a good way to represent students and staff members at first glance, but if we started to extend this system we would soon encounter some complications. At a real university, the divisions between staff and students and administrative and teaching staff are not always clear-cut. A student who tutors a course is also a kind of temporary staff member. A staff member can enrol in a course. A staff member can have *both* an administrative role in the department *and* a teaching position.

In Python it is possible for a class to inherit from multiple other classes. We could, for example, create a class called `Tutor`, which inherits from both `Student` and `StaffMember`. Multiple inheritance isn't too difficult to understand if a class inherits from multiple classes which have completely different properties, but things get complicated if two parent classes implement the same method or attribute.

If classes `B` and `C` inherit from `A` and class `D` inherits from `B` and `C`, and both `B` and `C` have a method `do_something`, which `do_something` will `D` inherit? This ambiguity is known as the *diamond problem*, and different languages resolve it in different ways. In our `Tutor` class we would encounter this problem with the `__init__` method.

Fortunately the `super` function knows how to deal gracefully with multiple inheritance. If we use it inside the `Tutor` class's `__init__` method, all of the parent classes' `__init__` methods should be called in a sensible order. We would then end up with a class which has all the attributes and methods found in both `Student` and `StaffMember`.

### Mix-ins

If we use multiple inheritance, it is often a good idea for us to design our classes in a way which avoids the kind of ambiguity described above. One way of doing this is to split up optional functionality into *mix-ins*. A Mix-in is a class which is not intended to stand on its own – it exists to add extra functionality to another class through multiple inheritance. For example, let us try to rewrite the example above so that each set of related things that a person can do at a university is written as a mix-in:

```python
class Person:
    def __init__(self, name, surname, number):
        self.name = name
        self.surname = surname
        self.number = number


class LearnerMixin:
    def __init__(self):
        self.classes = []

    def enrol(self, course):
        self.classes.append(course)


class TeacherMixin:
    def __init__(self):
        self.courses_taught = []
    def assign_teaching(self, course):
        self.courses_taught.append(course)


class Tutor(Person, LearnerMixin, TeacherMixin):
    def __init__(self, *args, **kwargs):
        super(Tutor, self).__init__(*args, **kwargs)

jane = Tutor("Jane", "Smith", "SMTJNX045")
jane.enrol(a_postgrad_course)
jane.assign_teaching(an_undergrad_course)
```

Now Tutor inherits from one "main" class, `Person`, and two mix-ins which are not related to `Person`. Each mix-in is responsible for providing a specific piece of optional functionality. Our mix-ins still have `__init__` methods, because each one has to initialise a list of courses (we saw in the previous chapter that we can't do this with a class attribute). Many mix-ins just provide additional methods and don't initialise anything. This sometimes means that they depend on other properties which already exist in the class which inherits from them.

We could extend this example with more mix-ins which represent the ability to pay fees, the ability to get paid for services, and so on – we could then create a relatively flat hierarchy of classes for different kinds of people which inherit from `Person` and some number of mix-ins.

## Abstract classes and interfaces

In some languages it is possible to create a class which can't be instantiated. That means that we can't use this class directly to create an object – we can only inherit from the class, and use the subclasses to create objects.

Why would we want to do this? Sometimes we want to specify a set of properties that an object needs to have in order to be suitable for some task – for example, we may have written a function which expects one of its parameters to be an object with certain methods that our function will need to use. We can create a class which serves as a *template* for suitable objects by defining a list of methods that these objects must implement. This class is not intended to be instantiated because all our method definitions are empty – all the *insides* of the methods must be implemented in a subclass.

The abstract class is thus an *interface* definition – some languages also have a type of structure called an interface, which is very similar. We say that a class *implements* an interface if it inherits from the class which specifies that interface.

In Python we can't prevent anyone from instantiating a class, but we can create something similar to an abstract class by using `NotImplementedError` inside our method definitions. For example, here are some "abstract" classes which can be used as templates for shapes:

```
class Shape2D:
    def area(self):
        raise NotImplementedError()

class Shape3D:
    def volume(self):
        raise NotImplementedError()
```

Any two-dimensional shape has an area, and any three-dimensional shape has a volume. The formulae for working out area and volume differ depending on what shape we have, and objects for different shapes may have completely different attributes.

If an object inherits from `2DShape`, it will gain that class's default `area` method – but the default method raises an error which makes it clear to the user that a custom method must be defined in the child object:

```
class Square(Shape2D):
    def __init__(self, width):
        self.width = width

    def area(self):
        return self.width ** 2
```

### Exercise 3

1. Write an "abstract" class, `Box`, and use it to define some methods which any box object should have: `add`, for adding any number of items to the box, `empty`, for taking all the items out of the box and returning them as a list, and `count`, for counting the items which are currently in the box. Write a simple `Item` class which has a `name` attribute and a `value` attribute – you can assume that all the items you will use will be `Item` objects. Now write two subclasses of `Box` which use different underlying collections to store items: `ListBox` should use a list, and `DictBox` should use a dict.
2. Write a function, `repack_boxes`, which takes any number of boxes as parameters, gathers up all the items they contain, and redistributes them as evenly as possible over all the boxes. Order is unimportant. There are multiple ways of doing this. Test your code with a `ListBox` with 20 items, a `ListBox` with 9 items and a `DictBox` with 5 items. You should end up with two boxes with 11 items each, and one box with 12 items.

## Avoiding inheritance

Inheritance can be a useful technique, but it can also be an unnecessary complication. As we have already discussed, multiple inheritance can cause a lot of ambiguity and confusion, and hierarchies which use multiple inheritance should be designed carefully to minimise this.

A deep hierarchy with many layers of subclasses may be difficult to read and understand. In our first inheritance example, to understand how the `Lecturer` class works we have to read through *three* different classes instead of one. If our classes are long and split into several different files, it can be hard to figure out which subclass is responsible for a particular piece of behaviour. You should avoid creating hierarchies which are more than one or two classes deep.

In some statically typed languages inheritance is very popular because it allows the programmer to work around some of the restrictions of static typing. If a lecturer and a student are both a kind of person, we can write a function which accepts a parameter of type `Person` and have it work on both lecturer and student objects because they both inherit from `Person`. This is known as *polymorphism*.

In Python inheritance is not compulsory for polymorphism, because Python is not statically typed. A function can work on both lecturer and student objects if they both have the appropriate attributes and methods even if these objects *don't* share a parent class, and are completely unrelated. When you check parameters yourself, you are encouraged not to check an object's type directly, but instead to check for the presence of the methods and attributes that your function needs to use – that way you are not forcing the parameter objects into an inheritance hierarchy when this is unnecessary.

### Replacing inheritance with composition

Sometimes we can replace inheritance with composition and achieve a similar result – this approach is sometimes considered preferable. In the mix-in example, we split up the possible behaviours of a person into logical groups. Instead of implementing these sets of behaviours as mix-ins and having our class inherit from them, we can add them as *attributes* to the `Person` class:

```
class Learner:
    def __init__(self):
        self.classes = []

    def enrol(self, course):
        self.classes.append(course)


class Teacher:
    def __init__(self):
        self.courses_taught = []

    def assign_teaching(self, course):
        self.courses_taught.append(course)


class Person:
    def __init__(self, name, surname, number, learner=None, teacher=None):
        self.name = name
        self.surname = surname
        self.number = number

        self.learner = learner
        self.teacher = teacher

jane = Person("Jane", "Smith", "SMTJNX045", Learner(), Teacher())
jane.learner.enrol(a_postgrad_course)
jane.teacher.assign_teaching(an_undergrad_course)
```

Now instead of calling the `enrol` and `assign_teaching` methods on our person object directly, we *delegate* to the object's `learner` and `teacher` attributes.

### Exercise 4

1. Rewrite the `Person` class in the last example, implementing additional methods called `enrol` and `assign_teaching` which hide the delegation. These methods should raise an appropriate error message if the delegation cannot be performed because the corresponding attribute has not been set.

## Answers to exercises

### Answer to exercise 1

1. The following relationships should exist between the four classes:
   - a one-to-many relationship between albums and songs – this is likely to be bidirectional, since songs and albums are quite closely coupled.
   - a one-to-many relationship between artists and songs. This can be unidirectional or bidirectional. We don't really need to store links to all of an artist's songs on an artist object, since a reference to the artist from each song is enough for us to search our songs by artist, but if the music collection is very large it may be a good idea to cache this list.
   - a one-to-many relationship between artists and albums, which can be unidirectional or bidirectional for the same reasons.

  - a one-to-many relationship between playlists and songs – this is likely to be unidirectional, since it's uncommon to keep track of all the playlists on which a particular song appears.

2. Here is an example program:

```python
class Song:

    def __init__(self, title, artist, album, track_number):
        self.title = title
        self.artist = artist
        self.album = album
        self.track_number = track_number

        artist.add_song(self)


class Album:

    def __init__(self, title, artist, year):
        self.title = title
        self.artist = artist
        self.year = year

        self.tracks = []

        artist.add_album(self)

    def add_track(self, title, artist=None):
        if artist is None:
            artist = self.artist

        track_number = len(self.tracks)

        song = Song(title, artist, self, track_number)

        self.tracks.append(song)


class Artist:
    def __init__(self, name):
        self.name = name

        self.albums = []
        self.songs = []

    def add_album(self, album):
        self.albums.append(album)

    def add_song(self, song):
        self.songs.append(song)


class Playlist:
    def __init__(self, name):
        self.name = name
        self.songs = []

    def add_song(self, song):
        self.songs.append(song)
```

```
band = Artist("Bob's Awesome Band")
album = Album("Bob's First Single", band, 2013)
album.add_track("A Ballad about Cheese")
album.add_track("A Ballad about Cheese (dance remix)")
album.add_track("A Third Song to Use Up the Rest of the Space")

playlist = Playlist("My Favourite Songs")

for song in album.tracks:
    playlist.add_song(song)
```

## Answer to exercise 2

1. Here is an example program:

```
# Exceptions

class DuplicateUsernameError(Exception):
    pass

class InvalidAgeError(Exception):
    pass

class UnderageError(Exception):
    pass

class InvalidEmailError(Exception):
    pass

# A class for a user's data

class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email

example_list = [
    ("jane", "jane@example.com", 21),
    ("bob", "bob@example", 19),
    ("jane", "jane2@example.com", 25),
    ("steve", "steve@somewhere", 15),
    ("joe", "joe", 23),
    ("anna", "anna@example.com", -3),
]

directory = {}

for username, email, age in example_list:
    try:
        if username in directory:
            raise DuplicateUsernameError()
        if age < 0:
            raise InvalidAgeError()
        if age < 16:
            raise UnderageError()
        email_parts = email.split('@')
        if len(email_parts) != 2 or not email_parts[0] or not email_parts[1]:
            raise InvalidEmailError()

    except DuplicateUsernameError:
        print("Username '%s' is in use." % username)
    except InvalidAgeError:
        print("Invalid age: %d" % age)
    except UnderageError:
        print("User %s is underage." % username)
    except InvalidEmailError:
        print("'%s' is not a valid email address." % email)

    else:
        directory[username] = User(username, email)
```

## Answer to exercise 3

1. Here is an example program:

```
    class Box:
        def add(self, *items):
            raise NotImplementedError()

        def empty(self):
            raise NotImplementedError()

        def count(self):
            raise NotImplementedError()


    class Item:
        def __init__(self, name, value):
            self.name = name
            self.value = value


    class ListBox(Box):
        def __init__(self):
            self._items = []

        def add(self, *items):
            self._items.extend(items)

        def empty(self):
            items = self._items
            self._items = []
            return items

        def count(self):
            return len(self._items)


    class DictBox(Box):
        def __init__(self):
            self._items = {}
        def add(self, *items):
            self._items.update(dict((i.name, i) for i in items))

        def empty(self):
            items = list(self._items.values())
            self._items = {}
            return items

        def count(self):
            return len(self._items)
```

2. Here is an example program:

```
    def repack_boxes(*boxes):
        items = []

        for box in boxes:
            items.extend(box.empty())

        while items:
            for box in boxes:
                try:
                    box.add(items.pop())
                except IndexError:
                    break

    box1 = ListBox()
    box1.add(Item(str(i), i) for i in range(20))

    box2 = ListBox()
    box2.add(Item(str(i), i) for i in range(9))

    box1 = DictBox()
    box1.add(Item(str(i), i) for i in range(5))

    repack_boxes(box1, box2, box3)

    print(box1.count())
    print(box2.count())
    print(box3.count())
```

## Answer to exercise 4

1. Here is an example program:

```
    class Person:
        def __init__(self, name, surname, number, learner=None, teacher=None):
            self.name = name
            self.surname = surname
            self.number = number

            self.learner = learner
            self.teacher = teacher

        def enrol(self, course):
            if not hasattr(self, "learner"):
                raise NotImplementedError()
            self.learner.enrol(course)

        def assign_teaching(self, course):
            if not hasattr(self, "teacher"):
                raise NotImplementedError()

            self.teacher.assign_teaching(course)
```

CHAPTER 11

# Packaging and testing

## Modules

All software projects start out small, and you are likely to start off by writing all of your program's code in a single file. As your project grows, it will become increasingly inconvenient to do this – it's difficult to find anything in a single file of code, and eventually you are likely to encounter a problem if you want to call two different classes by the same name. At some point it will be a good idea to tidy up the project by splitting it up into several files, putting related classes or functions together in the same file.

Sometimes there will be a natural way to split things up from the start of the project – for example, if your program has a database backend, a business logic layer and a graphical user interface, it is a good idea to put these three things into three separate files from the start instead of mixing them up.

How do we access code in one file from another file? Python provides a mechanism for creating a *module* from each file of source code. You can use code which is defined inside a module by *importing* the module using the `import` keyword – we have already done this with some built-in modules in previous examples.

Each module has its own namespace, so it's OK to have two classes which have the same name, as long as they are in different modules. If we import the whole module, we can access properties defined inside that module with the `.` operator:

```
import datetime
today = datetime.date.today()
```

We can also import specific classes or functions from the module using the `from` keyword, and use their names independently:

```
from datetime import date
today = date.today()
```

Creating a module is as simple as writing code into a Python file. A module which has the same name as the file (without the `.py` suffix) will automatically be created – we will be able to import it if we run Python from the directory where the file is stored, or a script which is in the same directory as the other Python files. If you want to be able to import your modules no matter where you run Python, you should package your code and install it – we will look at this in the next chapter.

We can use the `as` keyword to give an imported name an alias in our code – we can use this to shorten a frequently used module name, or to import a class which has the same name as a class which is already in our namespace, without overriding it:

```
import datetime as dt
today = dt.date.today()
```

```
from mymodule import MyClass as FirstClass
from myothermodule import MyClass as OtherClass
```

We can also import everything from a module using `*`, but this is not recommended, since we might accidentally import things which redefine names in our namespace and not realise it:

```
from mymodule import *
```

## Packages

Just as a module is a collection of classes or functions, a package is a collection of modules. We can organise several module files into a directory structure. There are various tools which can then convert the directory into a special format (also called a “package”) which we can use to install the code cleanly on our computer (or other people’s computers). This is called *packaging*.

### Packaging a program with Distribute

A library called Distribute is currently the most popular tool for creating Python packages, and is recommended for use with Python 3. It isn’t a built-in library, but can be installed with a tool like `pip` or `easy_install`. Distribute is a more modern version of an older packaging library called Setuptools, and has been designed to replace it. The original Setuptools doesn’t support Python 3, but you may encounter it if you work on Python 2 code. The two libraries have almost identical interfaces, and Distribute’s module name, which you import in your code, is also `setuptools`.

Let’s say that we have split up our large program, which we will call “ourprog”, into three files: `db.py` for the database backend, `rules.py` for the business logic, and `gui.py` for the graphical user interface. First, we should arrange our files into the typical directory structure which packaging tools expect:

```
ourprog/
    ourprog/
        __init__.py
        db.py
        gui.py
        rules.py
    setup.py
```

We have created two new files. `__init__.py` is a special file which marks the inner `ourprog` directory as a package, and also allows us to import all of `ourprog` as a module. We can use this file to import classes or functions from our modules (`db`, `gui` and `rules`) into the package’s namespace, so that they can be imported directly from `ourprog` instead of from `ourprog.db`, and so on – but for now we will leave this file blank.

The other file, `setup.py`, is the specification for our package. Here is a minimal example:

```
from setuptools import setup

setup(name='ourprog',
    version='0.1',
    description='Our first program',
    url='http://example.com',
    author='Jane Smith',
    author_email='jane.smith@example.com',
    license='GPL',
    packages=['ourprog'],
    zip_safe=False,
)
```

We create the package with a single call of the `setup` function, which we import from the `setuptools` module. We pass in several parameters which describe our package.

### Installing and importing our modules

Now that we have written a `setup.py` file, we can run it in order to install our package on our system. Although this isn't obvious, `setup.py` is a script which takes various command-line parameters – we got all this functionality when we imported `setuptools`. We have to pass an `install` parameter to the script to install the code. We need to input this command on the commandline, while we are in the same directory as `setup.py`:

```
python3 setup.py install
```

If everything has gone well, we should now be able to import `ourprog` from anywhere on our system.

## Documentation

Code documentation is often treated as an afterthought. While we are writing a program, it can seem to us that what our functions and classes do is obvious, and that writing down a lengthy explanation for each one is a waste of time. We may feel very differently when we look at our code again after a break of several months, or when we are forced to read and understand somebody else's undocumented code!

We have already seen how we can insert comments into Python code using the `#` symbol. Comments like this are useful for annotating individual lines, but they are not well-suited to longer explanations, or systematic documentation of all structures in our code. For that, we use *docstrings*.

### Docstrings

A docstring is just an ordinary string – it is usually written between triple quotes, because triple quotes are good for defining multiline string literals. What makes a docstring special is its position in the code. There are many tools which can parse Python code for strings which appear immediately after the definition of a module, class, function or method and aggregate them into an automatically generated body of documentation.

Documentation written like this can be easier to maintain than a completely separate document which is written by hand. The docstring for each individual class or function is defined next to the function in our code, where we are likely to see it and notice if it is out of sync and needs to be updated. Docstrings can also function as comments – other people will be able to see them while reading our source code. Interactive shells which use Python can also display docstrings when the user queries the usage of a function or class.

There are several different tools which parse docstrings – the one which is currently used the most is called Sphinx. In this course we won't go into detail about how to use Sphinx to generate documents, but we will see how to write docstrings in a format which is compatible with Sphinx.

### Docstring examples

The Sphinx markup language is a variant of reStructuredText (reST) with some extra keywords defined. There is no set, compulsory Sphinx docstring format – we can put any kind of Sphinx syntax inside the docstrings. A docstring should at the very least contain a basic description of the structure being documented.

If the structure is a function, it is helpful to describe all the parameters and the return value, and also mention if the function can raise any exceptions. Because Python isn't statically typed, it is important to provide information about the parameters that a function accepts.

We can also provide a much longer explanation after summarising all the basic information – we can go into as much detail as we like; there is no length limit.

Here are some examples of docstrings form various objects:

```python
"""This is a module for our Person class.
.. moduleauthor: Jane Smith <jane.smith@example.com>
"""

import datetime

class Person:
    """This is a class which represents a person. It is a bit of a silly class.
    It stores some personal information, and can calculate a person's age.
    """

    def __init__(self, name, surname, birthdate, address, telephone, email):
        """This method creates a new person.

        :param name: first name
        :type name: str
        :param surname: surname
        :type surname: str
        :param birthdate: date of birth
        :type birthdate: datetime.date
        :param address: physical address
        :type address: str
        :param telephone: telephone number
        :type telephone: str
        :param email: email address
        :type email: str
        """

        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        """This method calculates the person's age from the birthdate and the current date.

        :returns: int -- the person's age in years
        """
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age
```

## Testing

Automated tests are a beneficial addition to any program. They not only help us to discover errors, but also make it easier for us to modify code – we can run the tests after making a change to make sure that we haven't broken anything. This is vital in any large project, especially if there are many people working on the same code. Without tests, it can be very difficult for anyone to find out what other parts of the system a change could affect, and introducing any modification is thus a potential risk. This makes development on the project move very slowly, and changes often introduce bugs.

Adding automated tests can seem like a waste of time in a small project, but they can prove invaluable if the project becomes larger or if we have to return to it to make a small change after a long absence. They can also serve as a form of documentation – by reading through test cases we can get an idea of how our program is supposed to behave. Some people even advocate writing tests *first*, thereby creating a specification for what the program is supposed to do, and filling in the actual program code afterwards.

We may find this approach a little extreme, but we shouldn't go too far in the opposite direction – if we wait until we have written the entire program before writing any tests, we probably won't bother writing them at all. It is a good idea to write portions of our code and the tests for them at approximately the same time – then we can test our code while we are developing it. Most programmers write at least temporary tests during development to make sure that a new function is working correctly – we saw in a previous chapter how we can use print statements as a quick, but impermanent form of debugging. It is better practice to write a permanent test instead – once we have set up the testing framework, it really doesn't require a lot more effort.

In order for our code to be suitable for automated testing, we need to organise it in logical subunits which are easy to import and use independently from outside the program. We should already be doing this by using functions and classes, and avoiding reliance on global variables. If a function relies only on its input parameters to produce some kind of result, we can easily import this function into a separate testing module, and check that various examples of input produce the expected results. Each matching set of input and expected output is called a *test case*.

Tests which are applied to individual components in our code are known as *unit tests* – they verify that each of the components is working correctly. Testing the interaction between different components in a system is known as *integration testing*. A test can be called a *functional test* if it tests a particular feature, or *function* of the code – this is usually a relatively high-level specification of a requirement, not an actual single function.

In this section we will mostly look at unit testing, but we can apply similar techniques at any level of automated tests. When we are writing unit tests, as a rule of thumb, we should have a test for every function in our code (including each method of each class).

It is also good practice to write a new test whenever we fix a bug – the test should specifically check for the bug which we have just fixed. If the bug was caused by something which is a common mistake, it's possible that someone will make the same mistake again in the future – our test will help to prevent that. This is a form of *regression testing*, which aims to ensure that our code doesn't break when we add changes.

### Selecting test cases

How do we select test cases? There are two major approaches that we can follow: *black-box* or *glass-box* testing. We can also use a combination of the two.

In black-box testing, we treat our function like an opaque "black box". We don't use our knowledge of how the function is written to pick test cases – we only think about what the function is supposed to do. A strategy commonly used in black-box testing is is *equivalence testing* and *boundary value analysis*.

An *equivalence class* is a set of input values which should all produce similar output, and there are *boundaries* between neighbouring equivalence classes. Input values which lie near these boundaries are the most likely to produce incorrect output, because it's easy for a programmer to use < instead of <= or start counting from `1` instead of `0`, both of which could cause an *off-by-one* error. If we test an input value from inside each equivalence class, and additionally test values just before, just after and on each boundary, we can be reasonably sure that we have covered all the bases.

For example, consider a simple function which calculates a grade from a percentage mark. If we were to use equivalence testing and boundary analysis on this function, we would pick the test cases like this:

| Equivalence class | sample | lower boundary | just above boundary | just below boundary |
|---|---|---|---|---|
| mark > 100 | 150 | 100 | 101 | 99 |
| 80 <= mark <= 100 | 90 | 80 | 81 | 79 |
| 70 <= mark < 80 | 75 | 70 | 71 | 69 |
| 60 <= mark < 70 | 65 | 60 | 61 | 59 |
| 50 <= mark < 60 | 55 | 50 | 51 | 49 |
| 0 <= mark < 50 | 25 | 0 | 1 | -1 |
| mark < 0 | -50 | | | |

In glass-box testing, we pick our test cases by analysing the code inside our function. The most extensive form of this strategy is *path coverage*, which aims to test every possible path through the function.

A function without any selection or loop statements has only one path. Testing such a function is relatively easy – if it runs correctly once, it will probably run correctly every time. If the function contains a selection or loop statement, there will be more than one possible path passing through it: something different will happen if an *if* condition is true or if it is false, and a loop will execute a variable number of times. For a function like this, a single test case might not execute every statement in the code.

We could construct a separate test case for every possible path, but this rapidly becomes impractical. Each *if* statement doubles the number of paths – if our function had 10 *if* statements, we would need more than a thousand test cases, and if it had 20, we would need over a million! A more viable alternative is the *statement coverage* strategy, which only requires us to pick enough test cases to ensure that each *statement* inside our function is executed at least once.

## Writing unit tests

We can write unit tests in Python using the built-in `unittest` module. We typically put all our tests in a file hierarchy which is separate from our main program. For example, if we were to add tests to our packaging example above, we would probably create a test module for each of our three program modules, and put them all in a separate test directory:

```
ourprog/
    ourprog/
        __init__.py
        db.py
        gui.py
        rules.py
        test/
            __init__.py
            test_db.py
            test_gui.py
            test_rules.py
    setup.py
```

Suppose that our `rules.py` file contains a single class:

```python
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
    def fullname(self, title):
        if title not in self.TITLES:
            raise ValueError("Unrecognised title: '%s'" % title)

        return "%s %s %s" % (title, self.name, self.surname)
```

Our `test_rules.py` file should look something like this:

```
import unittest
from ourprog.rules import Person

class TestPerson(unittest.TestCase):

    def setUp(self):
        self.person = Person("Jane", "Smith")

    def test_init(self):
        self.assertEqual(self.person.name, "Jane")
        self.assertEqual(self.person.surname, "Smith")

    def test_fullname(self):
        self.assertEqual(self.person.fullname("Ms"), "Ms Jane Smith")
        self.assertEqual(self.person.fullname("Mrs"), "Mrs Jane Smith")
        self.assertRaises(ValueError, self.person.fullname, "HRH")
```

We import the `unittest` module, and also the class which we are going to test. This example assumes that we have packaged our code and installed it on our system, so that Python can find `ourprog.rules`.

In the `unittest` package, the `TestCase` class serves as a container for tests which need to share some data. For each collection of tests that we want to write, we define a class which inherits from `TestCase` and define all our tests as methods on that class.

In this example, all the tests in this `TestCase` test the same class, and there is one test per method (including the initialisation method) – but there is no compulsory mapping. You can use multiple `TestCase` classes to test each of your own classes, or perhaps have one `TestCase` for each set of related functionality.

We set up the class which we are going to test in the `setUp` method – this special method will be executed before each test is run. There is also a `tearDown` method, which we can use if we need to do something *after* each test.

Inside each test, we use the *assertion* methods of `TestCase` to check if certain things are true about our program's behaviour. As soon as one assertion statement fails, the whole test fails. We will often use `assertEqual`, but there are many other assertion methods like `assertNotEqual`, `assertTrue` or `assertIn`. `assertRaises` lets us check that a function raises an exception. Note that when we use this assertion method we don't call the function (because it would raise an exception!) – we just pass in the function name and its parameters.

There are many ways of running the tests once we have written them. Here is a simple way of running all the tests from a single file: at the bottom of `test_rules.py`, we can add:

```
if __name__ == '__main__':
    unittest.main()
```

Now if we execute `test_rules.py` with Python, `unittest` will run the `TestCase` which we have defined. The condition in the `if` statement detects whether we are running the file as a script, and prevents the `main` function from being executed if we import this module from another file. We will learn more about writing scripts in the next chapter.

We can also execute the unittest module on the commandline and use it to import and run some or all of our tests. By default the module will try to *discover* all the tests that can be imported from the current directory, but we can also specify one or more module, class or test method:

```
# these commands will try to find all our tests
python -m unittest
python -m unittest discover

# but we can be more specific
python -m unittest ourprog.test.test_rules
python -m unittest ourprog.test.test_rules.TestPerson
python -m unittest ourprog.test.test_rules.TestPerson.test_fullname

# we can also turn on verbose output with -v
python -m unittest -v test_rules
```

The `unittest` package also allows us to group some or all of our tests into *suites*, so that we can run many related tests at once. One way to add all the tests from our `TestPerson` class to a suite is to add this function to the `test_rules.py` file:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(TestPerson)
    return suite
```

We could define a suite in `ourprog/test/__init__py` which contains all the tests from all our modules, either by combining suites from all the modules or just adding all the tests directly. The `TestSuite` class and the `TestLoader` class, which we can use to build suites, are both very flexible. They allow us to construct test suites in many different ways.

We can integrate our tests with our packaging code by adding a `test_suite` parameter to our `setup` call in `setup.py`. Despite its name, this parameter doesn't have to be a suite – we can just specify the full name of our `test` module to include all our tests:

```
setup(name='ourprog',
    # (...)
    test_suite='ourprog.test',
    # (...)
)
```

Now we can build our package and run all our tests by passing the `test` parameter to `setup.py`:

```
python setup.py test

# We can override what to run using -s
# For example, we can run a single module
python setup.py test -s ourprog.test.test_rules
```

In previous versions of Python, we would have needed to define a test suite just to run all our tests at once, but in newer versions it is no longer necessary to define our own suites for simple test organisation. We can now easily run all the tests, or a single module, class or method just by using `unittest` on the commandline or `setup.py test`. We may still find it useful to write custom suites for more complex tasks – we may wish to group tests which are spread across multiple modules or classes, but which are all related to the same feature.

## Checking for test coverage

How do we know if our test cases cover all the statements in our code? There are many third-party unit testing libraries which include functionality for calculating coverage, but we can perform a very simple check by using `unittest` together with the built-in `trace` module. We can modify our test module like this:

```
import trace, sys

# all our test code

if __name__ == "__main__":
    t = trace.Trace(ignoredirs=[sys.prefix, sys.exec_prefix], count=1, trace=0)
    t.runfunc(unittest.main)
    r = t.results()
    r.write_results(show_missing=True)
```

The first line in the `if` block creates a `Trace` object which is going to trace the execution of our program – across all the source files in which code is found. We use the `ignoredirs` parameter to ignore any code in Python's installed modules – now we should only see results from our program file and our test file. Setting the `count` parameter to `1` makes the `Trace` object count the number of times that each line is executed, and setting `trace` to `0` prevents it from printing out lines as they are executed.

The second line specifies that we should run our test suite's main function. The third line retrieves the results from the object – the results are another kind of object, which is based on a dictionary. This object has a convenient `write_results` method which we use in the fourth line to output a file of line counts for each of our source files. They will be written to the current directory by default, but we could also specify another directory with an optional parameter. The `show_missing` parameter ensures that lines which were *never* executed are included in the files and clearly marked.

We need to run the test file directly to make sure that the code inside the `if` block is executed. Afterwards, we should find two files which end in `.cover` in the current directory – one for our program file and one for our test file. Each line should be annotated with the number of times that it was executed when we ran our test code.

## Exercise 1

In this exercise you will write a program which estimates the cost of a telephone call, and design and implement unit tests for the program.

The phone company applies the following rules to a phone call to calculate the charge:

- The minimum before-tax charge of 59.400 cents applies to all calls to any destination up to 50km away and 89.000 cents for any destination further than 50km away.
- Calls are charged on a per-second basis at 0.759 cents per second (<= 50km) and 1.761 cents per second (> 50km)
- Off-peak seconds (from 19:00:00 to 06:59:59 the next day) are given a discount of 40% off (<= 50km) and 50% off (> 50km) off the above rate
- If the type of call was share-call AND the destination is more than 50km away, there is a discount of 50% off after any off-peak discount (minimum charge still applies). However, share-calls over shorter distances are not discounted.
- Finally, VAT of 14% is added to give the final cost.

Your program should ask for the following input:

- The starting time of the call (to be split up into hours, minutes and seconds)
- The duration of the call (to be split up into minutes and seconds)
- Whether the duration was more than 50km away
- Whether the call was share-call

Hint: you can prompt the user to input hours, minutes and seconds at once by asking for a format like *HH:MM:SS* and splitting the resulting string by the delimiter. You may assume that the user will enter valid input, and that no call will exceed 59 minutes and 59 seconds.

Your program should output the following information:

- The basic cost
- The off-peak discount
- The share-call discount
- The net cost
- The VAT
- The total cost

1. Before you write the program, identify the equivalence classes and boundaries that you will need to use in equivalence testing and boundary analysis when writing black-box tests. This may help you to design the program itself, and not just the tests!
2. Write the program. Remember that you will need to write unit tests for this program, and design it accordingly – the calculation that you need to test should be placed in some kind of unit, like a function, which can be imported from outside of the program and used independently of the rest of the code (like the user input)!
3. Now implement the black-box tests which you have designed by writing a unit test module for your program. Run all the tests, and make sure that they pass! Then use the `trace` module to check how well your tests cover your function code.

## Answers to exercises

### Answer to exercise 1

1. Peak and off-peak times provide an obvious source of equivalence classes for the start and duration of the call. A call could start during peak or off-peak hours, and it could end in peak or off-peak hours (because the maximum duration of a call is just under an hour, a call can cross the peak/off-peak boundary once, but not twice). A call could also cross over the boundary between days, and this wrapping must be handled correctly.

   A good set of boundaries for the start of the call would be: 00:00, 06:00, 07:00, 18:00 and 19:00. A good set of boundaries for the duration of the call would be the minimum and maximum durations – 00:00 and 59:59. We don't need to test every combination of start time and duration – the duration of the call is only really important if the call starts within an hour of the peak/off-peak switch. We can test the remaining start times with a single duration.

   The other input values entered by the user are boolean, so only a true value and a false value needs to be tested for each. Again, we don't need to test each boolean option with every possible combination of the previous options – one or two cases should be sufficient.

2. Here is an example program:

```
import datetime

# The first value in each tuple is for distances <= 50km
# The second value is for distances > 50km
MIN_CHARGE = (59.400, 89.000)
CHARGE_PER_SEC = (0.759, 1.761)
OFFPEAK_DISCOUNT = (0.4, 0.5)
SHARECALL_DISCOUNT = (0.0, 0.5)
```

```
NEAR, FAR = 0, 1

OFF_PEAK_START = datetime.time(19, 0, 0)
HOUR_BEFORE_OFF_PEAK_START = datetime.time(18, 0, 0)
OFF_PEAK_END = datetime.time(7, 0, 0)
HOUR_BEFORE_OFF_PEAK_END = datetime.time(6, 0, 0)

VAT_RATE = 0.14

def price_estimate(start_str, duration_str, destination_str, share_call_str):
    start = datetime.datetime.strptime(start_str, "%H:%M:%S").time()
    d_m, d_s = [int(p) for p in duration_str.split(":")]
    duration = datetime.timedelta(minutes=d_m, seconds=d_s).total_seconds()
    # We set the destination to an index value we can use with the tuple constants
    destination = FAR if destination_str.lower() == 'y' else NEAR
    share_call = True if share_call_str.lower() == 'y' else False

    peak_seconds = 0
    off_peak_seconds = 0

    if start >= OFF_PEAK_END and start <= HOUR_BEFORE_OFF_PEAK_START:
        # whole call fits in peak time
        peak_seconds = duration
    elif start >= OFF_PEAK_START or start <= HOUR_BEFORE_OFF_PEAK_END:
        # whole call fits in off-peak time
        off_peak_seconds = duration
    else:
        # call starts within hour of peak/off-peak boundary
        secs_left_in_hour = 3600 - start.minute * 60 + start.second

        if start < OFF_PEAK_END:
            # call starts in off-peak time
            if duration > secs_left_in_hour:
                peak_seconds = duration - secs_left_in_hour
            off_peak_seconds = duration - peak_seconds
        else:
            # call starts in peak time
            if duration > secs_left_in_hour:
                off_peak_seconds = duration - secs_left_in_hour
            peak_seconds = duration - off_peak_seconds

    basic = CHARGE_PER_SEC[destination] * duration
    offpeak_discount = OFFPEAK_DISCOUNT[destination] * CHARGE_PER_SEC[destination] * off_peak_se
    if share_call:
        share_call_discount =  SHARECALL_DISCOUNT[destination] * (basic - offpeak_discount)
    else:
        share_call_discount = 0
    net = basic - offpeak_discount - share_call_discount

    if net < MIN_CHARGE[destination]:
        net = MIN_CHARGE[destination]

    vat = VAT_RATE * net
    total = net + vat

    return basic, offpeak_discount, share_call_discount, net, vat, total

if __name__ == "__main__":
    start_str = input("Please enter the starting time of the call (HH:MM:SS): ")
    duration_str = input("Please enter the duration of the call (MM:SS): ")
    destination_str = input("Was the destination more than 50km away? (Y/N): ")
    share_call_str = input("Was the call a share-call? (Y/N): ")

    results = price_estimate(start_str, duration_str, destination_str, share_call_str)

    print("""Basic cost: %g
    Off-peak discount: %g
    Share-call discount: %g
    Net cost: %g
    VAT: %g
    Total cost: %g
    """ % results)
```

3. Here is an example program, including a coverage test:

```
    import unittest
    import trace, sys

    from estimate import price_estimate

    class TestEstimate(unittest.TestCase):
        def test_off_peak(self):
            # all these cases should fall within off-peak hours and have the same result
            test_cases = [
                ("23:59:59", "10:00", "N", "N"),
                ("00:00:00", "10:00", "N", "N"),
                ("00:00:01", "10:00", "N", "N"),
                ("05:59:59", "10:00", "N", "N"),
                ("06:00:00", "10:00", "N", "N"),
                ("06:00:01", "10:00", "N", "N"),
                ("19:00:00", "10:00", "N", "N"),
                ("19:00:01", "10:00", "N", "N"),
            ]

            for start, duration, far_away, share_call in test_cases:
                basic, op_discount, sc_discount, net, vat, total = price_estimate(start, duration, f
                self.assertAlmostEqual(basic, 455.4)
                self.assertAlmostEqual(op_discount, 182.16)
                self.assertAlmostEqual(sc_discount, 0)
                self.assertAlmostEqual(net, 273.24)
                self.assertAlmostEqual(vat, 38.2536)
                self.assertAlmostEqual(total, 311.4936)

        def test_peak(self):
            # all these cases should fall within peak hours and have the same result
            test_cases = [
                ("07:00:00", "10:00", "N", "N"),
                ("07:00:01", "10:00", "N", "N"),
                ("17:59:59", "10:00", "N", "N"),
                ("18:00:00", "10:00", "N", "N"),
                ("18:00:01", "10:00", "N", "N"),
            ]

            for start, duration, far_away, share_call in test_cases:
                basic, op_discount, sc_discount, net, vat, total = price_estimate(start, duration, f
                self.assertAlmostEqual(basic, 455.4)
                self.assertAlmostEqual(op_discount, 0)
                self.assertAlmostEqual(sc_discount, 0)
                self.assertAlmostEqual(net, 455.4)
                self.assertAlmostEqual(vat, 63.756)
                self.assertAlmostEqual(total, 519.156)

        def test_peak_and_off_peak(self):
            # these test cases cross the peak / off-peak boundary, and all have different results.
            test_cases = [
                ("06:59:59", "59:59", "N", "N"),
                ("07:00:00", "59:59", "N", "N"),
                ("07:00:01", "59:59", "N", "N"),

                ("18:59:59", "59:59", "N", "N"),
                ("19:00:00", "59:59", "N", "N"),
                ("19:00:01", "59:59", "N", "N"),

                ("06:30:00", "00:00", "N", "N"),
                ("06:30:00", "00:01", "N", "N"),
                ("06:30:00", "59:58", "N", "N"),
                ("06:30:00", "59:59", "N", "N"),
            ]

            expected_results = [
                (2731.641, 36.128400000000006, 0, 2695.5126, 377.371764, 3072.884364),
                (2731.641, 0.0, 0, 2731.641, 382.42974, 3114.07074),
                (2731.641, 0.0, 0, 2731.641, 382.42974, 3114.07074),

                (2731.641, 1056.528, 0, 1675.113, 234.51582, 1909.62882),
                (2731.641, 1092.6564, 0, 1638.9846, 229.457844, 1868.442444),
                (2731.641, 1092.6564, 0, 1638.9846, 229.457844, 1868.442444),

                (0.0, 0.0, 0, 59.4, 8.316, 67.716), # minimum charge
                (0.759, 0.3036, 0, 59.4, 8.316, 67.716), # minimum charge
                (2730.882, 546.48, 0, 2184.402, 305.81628, 2490.21828),
                (2731.641, 546.48, 0, 2185.161, 305.92254, 2491.08354),
            ]

            for parameters, results in zip(test_cases, expected_results):
                basic, op_discount, sc_discount, net, vat, total = price_estimate(*parameters)
                exp_basic, exp_op_discount, exp_sc_discount, exp_net, exp_vat, exp_total = results
                self.assertAlmostEqual(basic, exp_basic)
                self.assertAlmostEqual(op_discount, exp_op_discount)
                self.assertAlmostEqual(sc_discount, exp_sc_discount)
                self.assertAlmostEqual(net, exp_net)
                self.assertAlmostEqual(vat, exp_vat)
                self.assertAlmostEqual(total, exp_total)

        def test_far_destination_share_call(self):
            # now we repeat some basic test cases with a far destination and/or share-call

            test_cases = [
                # off-peak
                ("23:59:59", "10:00", "Y", "N"),
                ("23:59:59", "10:00", "Y", "Y"),
                ("23:59:59", "10:00", "N", "Y"),
                # peak
                ("07:00:00", "10:00", "Y", "N"),
                ("07:00:00", "10:00", "Y", "Y"),
                ("07:00:00", "10:00", "N", "Y"),
            ]

            expected_results = [
                (1056.6, 528.3, 0, 528.3, 73.962, 602.262),
                (1056.6, 528.3, 264.15, 264.15, 36.981, 301.131),
                (455.4, 182.16, 0.0, 273.24, 38.2536, 311.4936),

                (1056.6, 0.0, 0, 1056.6, 147.924, 1204.524),
                (1056.6, 0.0, 528.3, 528.3, 73.962, 602.262),
                (455.4, 0.0, 0.0, 455.4, 63.756, 519.156),
            ]

            for parameters, results in zip(test_cases, expected_results):
                basic, op_discount, sc_discount, net, vat, total = price_estimate(*parameters)
                exp_basic, exp_op_discount, exp_sc_discount, exp_net, exp_vat, exp_total = results
                self.assertAlmostEqual(basic, exp_basic)
                self.assertAlmostEqual(op_discount, exp_op_discount)
                self.assertAlmostEqual(sc_discount, exp_sc_discount)
                self.assertAlmostEqual(net, exp_net)
                self.assertAlmostEqual(vat, exp_vat)
                self.assertAlmostEqual(total, exp_total)

    if __name__ == "__main__":
        t = trace.Trace(ignoredirs=[sys.prefix, sys.exec_prefix], count=1, trace=0)
        t.runfunc(unittest.main)

        r = t.results()
        r.write_results(show_missing=True)
```

CHAPTER 12

# Useful modules in the Standard Library

Python comes with a built-in selection of modules which provide commonly used functionality. We have encountered some of these modules in previous chapters – for example, `itertools`, `logging`, `pdb` and `unittest`. We will look at a few more examples in this chapter. This is only a brief overview of a small subset of the available modules – you can see the full list, and find out more details about each one, by reading the Python Standard Library documentation.

## Date and time: `datetime`

The `datetime` module provides us with objects which we can use to store information about dates and times:

- `datetime.date` is used to create dates which are not associated with a time.
- `datetime.time` is used for times which are independent of a date.
- `datetime.datetime` is used for objects which have both a date and a time.
- `datetime.timedelta` objects store *differences* between dates or datetimes – if we subtract one datetime from another, the result will be a timedelta.
- `datetime.timezone` objects represent time zone adjustments as offsets from UTC. This class is a subclass of `datetime.tzinfo`, which is not meant to be used directly.

We can query these objects for a particular component (like the year, month, hour or minute), perform arithmetic on them, and extract printable string versions from them if we need to display them. Here are a few examples:

```
import datetime

# this class method creates a datetime object with the current date and time
now = datetime.datetime.today()

print(now.year)
print(now.hour)
print(now.minute)

print(now.weekday())

print(now.strftime("%a, %d %B %Y"))

long_ago = datetime.datetime(1999, 3, 14, 12, 30, 58)

print(long_ago) # remember that this calls str automatically
print(long_ago < now)
```

```
difference = now - long_ago
print(type(difference))
print(difference) # remember that this calls str automatically
```

### Exercise 1

1. Print ten dates, each two a week apart, starting from today, in the form *YYYY-MM-DD*.

## Mathematical functions: `math`

The `math` module is a collection of mathematical functions. They can be used on floats or integers, but are mostly intended to be used on floats, and usually return floats. Here are a few examples:

```
import math

# These are constant attributes, not functions
math.pi
math.e

# round a float up or down
math.ceil(3.3)
math.floor(3.3)

# natural logarithm
math.log(5)
# logarithm with base 10
math.log(5, 10)
math.log10(5) # this function is slightly more accurate

# square root
math.sqrt(10)

# trigonometric functions
math.sin(math.pi/2)
math.cos(0)

# convert between radians and degrees
math.degrees(math.pi/2)
math.radians(90)
```

If you need mathematical functions to use on complex numbers, you should use the `cmath` module instead.

### Exercise 2

1. Write an object which represents a sphere of a given radius. Write a method which calculates the sphere's volume, and one which calculates its surface area.

## Pseudo-random numbers: `random`

We call a sequence of numbers *pseudo-random* when it appears in some sense to be random, but actually isn't. Pseudo-random number sequences are generated by some kind of predictable algorithm, but they possess enough of the properties of truly random sequences that they can be used in many applications that call for random numbers.

It is difficult for a computer to generate numbers which are genuinely random. It is possible to gather truly random input using hardware, from sources such as the user's keystrokes or tiny fluctuations in voltage measurements, and use that input to generate random numbers, but this process is more complicated and expensive than pseudo-random number generation, which can be done purely in software.

Because pseudo-random sequences aren't actually random, it is also possible to reproduce the exact same sequence twice. That isn't something we would want to do by accident, but it is a useful thing to be able to deliberately while debugging software, or in an automated test.

In Python can we use the `random` module to generate pseudo-random numbers, and do a few more things which depend on randomness. The core function of the module generates a random float between 0 and 1, and most of the other functions are derived from it. Here are a few examples:

```python
import random

# a random float from 0 to 1 (excluding 1)
random.random()

pets = ["cat", "dog", "fish"]
# a random element from a sequence
random.choice(pets)
# shuffle a list (in place)
random.shuffle(pets)

# a random integer from 1 to 10 (inclusive)
random.randint(1, 10)
```

When we load the `random` module we can *seed* it before we start generating values. We can think of this as picking a place in the pseudo-random sequence where we want to start. We normally want to start in a different place every time – by default, the module is seeded with a value taken from the system clock. If we want to reproduce the same random sequence multiple times – for example, inside a unit test – we need to pass the same integer or string as parameter to `seed` each time:

```python
# set a predictable seed
random.seed(3)
random.random()
random.random()
random.random()

# now try it again
random.seed(3)
random.random()
random.random()
random.random()

# and now try a different seed
random.seed("something completely different")
random.random()
random.random()
random.random()
```

## Exercise 3

1. Write a program which randomly picks an integer from 1 to 100. Your program should prompt the user for guesses – if the user guesses incorrectly, it should print whether the guess is too high or too low. If the user

guesses correctly, the program should print how many guesses the user took to guess the right answer. You can assume that the user will enter valid input.

## Matching string patterns: `re`

The `re` module allows us to write *regular expressions*. Regular expressions are a mini-language for matching strings, and can be used to find and possibly replace text. If you learn how to use regular expressions in Python, you will find that they are quite similar to use in other languages.

The full range of capabilities of regular expressions is quite extensive, and they are often criticised for their potential complexity, but with the knowledge of only a few basic concepts we can perform some very powerful string manipulation easily.

**Note:** Regular expressions are good for use on plain text, but a bad fit for parsing more structured text formats like XML – you should always use a more specialised parsing library for those.

The Python documentation for the `re` module not only explains how to use the module, but also contains a reference for the complete regular expression syntax which Python supports.

### A regular expression primer

A regular expression is a string which describes a pattern. This pattern is compared to other strings, which may or may not match it. A regular expression can contain normal characters (which are treated literally as specific letters, numbers or other symbols) as well as special symbols which have different meanings within the expression.

Because many special symbols use the backslash (\) character, we often use *raw strings* to represent regular expressions in Python. This eliminates the need to use extra backslashes to escape backslashes, which would make complicated regular expressions much more difficult to read. If a regular expression doesn't contain any backslashes, it doesn't matter whether we use a raw string or a normal string.

Here are some very simple examples:

```
# this regular expression contains no special symbols
# it won't match anything except 'cat'
"cat"

# a . stands for any single character (except the newline, by default)
# this will match 'cat', 'cbt', 'c3t', 'c!t' ...
"c.t"

# a * repeats the previous character 0 or more times
# it can be used after a normal character, or a special symbol like .
# this will match 'ct', 'cat', 'caat', 'caaaaaaaaat' ...
"ca*t"
# this will match 'sc', 'sac', 'sic', 'supercalifragilistic' ...
"s.*c"

# + is like *, but the character must occur at least once
# there must be at least one 'a'
"ca+t"

# more generally, we can use curly brackets {} to specify any number of repeats
# or a minimum and maximum
# this will match any five-letter word which starts with 'c' and ends with 't'
```

```
"c.{3}t"
# this will match any five-, six-, or seven-letter word ...
"c.{3,5}t"

# One of the uses for ? is matching the previous character zero or one times
# this will match 'http' or 'https'
"https?"

# square brackets [] define a set of allowed values for a character
# they can contain normal characters, or ranges
# if ^ is the first character in the brackets, it *negates* the contents
# the character between 'c' and 't' must be a vowel
"c[aeiou]t"
# this matches any character that *isn't* a vowel, three times
"[^aeiou]{3}"
# This matches an uppercase UCT student number
"[B-DF-HJ-NP-TV-Z]{3}[A-Z]{3}[0-9]{3}"

# we use \ to escape any special regular expression character
# this would match 'c*t'
r"c\*t"
# note that we have used a raw string, so that we can write a literal backslash

# there are also some shorthand symbols for certain allowed subsets of characters:
# \d matches any digit
# \s matches any whitespace character, like space, tab or newline
# \w matches alphanumeric characters -- letters, digits or the underscore
# \D, \S and \W are the opposites of \d, \s and \w

# we can use round brackets () to *capture* portions of the pattern
# this is useful if we want to search and replace
# we can retrieve the contents of the capture in the replace step
# this will capture whatever would be matched by .*
"c(.*)t"

# ^ and $ denote the beginning or end of a string
# this will match a string which starts with 'c' and ends in 't'
"^c.*t$"

# | means "or" -- it lets us choose between multiple options.
"cat|dog"
```

## Using the `re` module

Now that we have seen how to construct regular expression strings, we can start using them. The `re` module provides us with several functions which allow us to use regular expressions in different ways:

- `search` searches for the regular expression inside a string – the regular expression will match if any subset of the string matches.
- `match` matches a regular expression against the entire string – the regular expression will only match if the *whole string* matches. `re.match('something', some_string)` is equivalent to `re.search('^something$', some_string)`.
- `sub` searches for the regular expression and replaces it with the provided replacement expression.
- `findall` searches for all matches of the regular expression within the string.

- `split` splits a string using any regular expression as a delimiter.
- `compile` allows us to convert our regular expression string to a pre-compiled regular expression *object*, which has methods analogous to the `re` module. Using this object is slightly more efficient.

As you can see, this module provides more powerful versions of some simple string operations: for example, we can also split a string or replace a substring using the built-in `split` and `replace` methods – but we can only use them with *fixed* delimiters or search patterns and replacements. With `re.sub` and `re.split` we can specify variable patterns instead of fixed strings.

All of the functions take a regular expression as the first parameter. `match`, `search`, `findall` and `split` also take the string to be searched as the second parameter – but in the `sub` function this is the third parameter, the second being the replacement string. All the functions also take an keyword parameter which specifies optional *flags*, which we will discuss shortly.

`match` and `search` both return match objects which store information such as the contents of captured groups. `sub` returns a modified copy of the original string. `findall` and `split` return a list of strings. `compile` returns a compiled regular expression object.

The methods of a regular expression object are very similar to the functions of the module, but the first parameter (the regular expression string) of each method is dropped – because it has already been compiled into the object.

Here are some usage examples:

```python
import re

# match and search are quite similar
print(re.match("c.*t", "cravat")) # this will match
print(re.match("c.*t", "I have a cravat")) # this won't
print(re.search("c.*t", "I have a cravat")) # this will

# We can use a static string as a replacement...
print(re.sub("lamb", "squirrel", "Mary had a little lamb."))
# Or we can capture groups, and substitute their contents back in.
print(re.sub("(.*) (BITES) (.*)", r"\3 \2 \1", "DOG BITES MAN"))
# count is a keyword parameter which we can use to limit replacements
print(re.sub("a", "b", "aaaaaaaaaa"))
print(re.sub("a", "b", "aaaaaaaaaa", count=1))

# Here's a closer look at a match object.
my_match = re.match("(.*) (BITES) (.*)", "DOG BITES MAN")
print(my_match.groups())
print(my_match.group(1))

# We can name groups.
my_match = re.match("(?P<subject>.*) (?P<verb>BITES) (?P<object>.*)", "DOG BITES MAN")
print(my_match.group("subject"))
print(my_match.groupdict())
# We can still access named groups by their positions.
print(my_match.group(1))

# Sometimes we want to find all the matches in a string.
print(re.findall("[^ ]+@[^ ]+", "Bob <bob@example.com>, Jane <jane.doe@example.com>"))

# Sometimes we want to split a string.
print(re.split(", *", "one,two,  three, four"))

# We can compile a regular expression to an object
my_regex = re.compile("(.*) (BITES) (.*)")
# now we can use it in a very similar way to the module
```

```
print(my_regex.sub(r"\3 \2 \1", "DOG BITES MAN"))
```

### Greed

Regular expressions are *greedy* by default – this means that if a part of a regular expression can match a variable number of characters, it will always try to match as many characters as possible. That means that we sometimes need to take special care to make sure that a regular expression doesn't match too much. For example:

```
# this is going to match everything between the first and last '"'
# but that's not what we want!
print(re.findall('".*"', '"one" "two" "three" "four"'))

# This is a common trick
print(re.findall('"[^"]*"', '"one" "two" "three" "four"'))

# We can also use ? after * or other expressions to make them *not greedy*
print(re.findall('".*?"', '"one" "two" "three" "four"'))
```

### Functions as replacements

We can also use `re.sub` to apply a *function* to a match instead of a string replacement. The function must take a match object as a parameter, and return a string. We can use this functionality to perform modifications which may be difficult or impossible to express as a replacement string:

```
def swap(m):
    subject = m.group("object").title()
    verb = m.group("verb")
    object = m.group("subject").lower()
    return "%s %s %s!" % (subject, verb, object)

print(re.sub("(?P<subject>.*) (?P<verb>.*) (?P<object>.*)!", swap, "Dog bites man!"))
```

### Flags

Regular expressions have historically tended to be applied to text line by line – newlines have usually required special handling. In Python, the text is treated as a single unit by default, but we can change this and a few other options using *flags*. These are the most commonly used:

- `re.IGNORECASE` – make the regular expression case-insensitive. It is case-sensitive by default.
- `re.MULTILINE` – make `^` and `$` match the beginning and end of each line (excluding the newline at the end), as well as the beginning and end of the whole string (which is the default).
- `re.DOTALL` – make `.` match any character (by default it does not match newlines).

Here are a few examples:

```
print(re.match("cat", "Cat")) # this won't match
print(re.match("cat", "Cat", re.IGNORECASE)) # this will

text = """numbers = 'one,
two,
three'
numbers = 'four,
five,
six'
not_numbers = 'cat,
dog'"""

print(re.findall("^numbers = '.*?'", text)) # this won't find anything
# we need both DOTALL and MULTILINE
print(re.findall("^numbers = '.*?'", text, re.DOTALL | re.MULTILINE))
```

**Note:** `re` functions only have a single keyword parameter for flags, but we can combine multiple flags into one using the `|` operator (bitwise *or*) – this is because the values of these constants are actually integer powers of two.

### Exercise 4

1. Write a function which takes a string parameter and returns `True` if the string is a valid Python variable name or `False` if it isn't. You don't have to check whether the string is a reserved keyword in Python – just whether it is otherwise syntactically valid. Test it using all the examples of valid and invalid variable names described in the first chapter.
2. Write a function which takes a string which contains two words separated by any amount and type of whitespace, and returns a string in which the words are swapped around and the whitespace is preserved.

## Parsing CSV files: `csv`

CSV stands for *comma-separated values* – it's a very simple file format for storing tabular data. Most spreadsheets can easily be converted to and from CSV format.

In a typical CSV file, each line represents a row of values in the table, with the columns separated by commas. Field values are often enclosed in double quotes, so that any literal commas or newlines inside them can be escaped:

```
"one","two","three"
"four, five","six","seven"
```

Python's `csv` module takes care of all this in the background, and allows us to manipulate the data in a CSV file in a simple way, using the `reader` class:

```
import csv

with open("numbers.csv") as f:
    r = csv.reader(f)
    for row in r:
        print row
```

There is no single CSV standard – the comma may be replaced with a different delimiter (such as a tab), and a different quote character may be used. Both of these can be specified as optional keyword parameters to `reader`.

Similarly, we can *write* to a CSV file using the `writer` class:

```
with open('pets.csv', 'w') as f:
    w = csv.writer(f)
    w.writerow(['Fluffy', 'cat'])
    w.writerow(['Max', 'dog'])
```

We can use optional parameters to `writer` to specify the delimiter and quote character, and also whether to quote all fields or only fields with characters which need to be escaped.

### Exercise 5

1. Open a CSV file which contains three columns of numbers. Write out the data to a new CSV file, swapping around the second and third columns and adding a fourth column which contains the sum of the first three.

## Writing scripts: `sys` and `argparse`

We have already seen a few scripts. Technically speaking, any Python file can be considered a script, since it can be executed without compilation. When we call a Python program a script, however, we usually mean that it contains statements other than function and class definitions – scripts *do something* other than define structures to be reused.

### Scripts vs libraries

We can combine class and function definitions with statements that use them in the same file, but in a large project it is considered good practice to keep them separate: to define all our classes in *library* files, and import them into the main program. If we do put both classes and main program in one file, we can ensure that the program is only executed when the file is run as a script and not if it is imported from another file – we saw an example of this earlier:

```python
class MyClass:
    pass

class MyOtherClass:
    pass

if __name__ == '__main__':
    my_object = MyClass()
    # do more things
```

If our file is written purely for use as a script, and will never be imported, including this conditional statement is considered unnecessary.

### Simple command-line parameters

When we run a program on the commandline, we often want to pass in parameters, or *arguments*, just as we would pass parameters to a function inside our code. For example, when we use the Python interpreter to run a file, we pass the filename in as an argument. Unlike parameters passed to a function in Python, arguments passed to an application on the commandline are separated by spaces and listed after the program name without any brackets.

The simplest way to access commandline arguments inside a script is through the `sys` module. All the arguments in order are stored in the module's `argv` attribute. We must remember that the first argument is always the name of the script file, and that all the arguments will be provided in string format. Try saving this simple script and calling it with various arguments after the script name:

```python
import sys

print sys.argv
```

### Complex command-line parameters

The `sys` module is good enough when we only have a few simple arguments – perhaps the name of a file to open, or a number which tells us how many times to execute a loop. When we want to provide a variety of complicated arguments, some of them optional, we need a better solution.

The `argparse` module allows us to define a wide range of compulsory and optional arguments. A commonly used type of argument is the *flag*, which we can think of as equivalent to a keyword argument in Python. A flag is optional, it has a name (sometimes both a long name and a short name) and it may have a value. In Linux and OSX programs, flag names often start with a dash (long names usually start with two), and this convention is sometimes followed by Windows programs too.

Here is a simple example of a program which uses `argparse` to define two positional arguments which must be integers, a flag which specifies an operation to be performed on the two numbers, and a flag to turn on verbose output:

```python
import argparse
import logging

parser = argparse.ArgumentParser()
# two integers
parser.add_argument("num1", help="the first number", type=int)
parser.add_argument("num2", help="the second number", type=int)
# a string, limited to a list of options
parser.add_argument("op", help="the desired arithmetic operation", choices=['add', 'sub', 'mul', 'di
# an optional flag, true by default, with a short and a long name
parser.add_argument("-v", "--verbose", help="turn on verbose output", action="store_true")

opts = parser.parse_args()

if opts.verbose:
    logging.basicConfig(level=logging.DEBUG)

logging.debug("First number: %d" % opts.num1)
logging.debug("Second number: %d" % opts.num2)
logging.debug("Operation: %s" % opts.op)

if opts.op == "add":
    result = opts.num1 + opts.num2
elif opts.op == "sub":
    result = opts.num1 - opts.num2
elif opts.op == "mul":
    result = opts.num1 * opts.num2
elif opts.op == "div":
    result = opts.num1 / opts.num2

print(result)
```

`argparse` automatically defines a `help` parameter, which causes the program's usage instructions to be printed when we pass `-h` or `--help` to the script. These instructions are automatically generated from the descriptions we supply in all the argument definitions. We will also see informative error output if we don't pass in the correct arguments. Try calling the script above with different arguments!

**Note:** if we are using Linux or OSX, we can turn our scripts into *executable files*. Then we can execute them directly instead of passing them as parameters to Python. To make our script executable we must mark it as executable using a system tool (`chmod`). We must also add a line to the beginning of the file to let the operating system know that it should use Python to execute it. This is typically `#!/usr/bin/env python`.

## Exercise 6

1. Write a script which reorders the columns in a CSV file. It should take as parameters the path of the original CSV file, a string listing the indices of the columns in the order that they should appear, and optionally a path

to the destination file (by default it should have the same name as the original file, but with a suffix). The script should return an error if the list of indices cannot be parsed or if any of the indices are not valid (too low or too high). You may allow indices to be negative or repeated. You should include usage instructions.

## Answers to exercises

### Answer to exercise 1

1. Here is an example program:

```
import datetime

today = datetime.datetime.today()

for w in range(10):
    day = today + datetime.timedelta(weeks=w)
    print(day.strftime("%Y-%m-%d"))
```

### Answer to exercise 2

1. Here is an example program:

```
import math

class Sphere:
    def __init__(self, radius):
        self.radius = radius

    def volume(self):
        return (4/3) * math.pi * math.pow(self.radius, 3)

    def surface_area(self):
        return 4 * math.pi * self.radius ** 2
```

### Answer to exercise 3

1. Here is an example program:

```
import random

secret_number = random.randint(1, 100)
guess = None
num_guesses = 0

while not guess == secret_number:
    guess = int(input("Guess a number from 1 to 100: "))
    num_guesses += 1

    if guess == secret_number:
        suffix = '' if num_guesses == 1 else 'es'
        print("Congratulations! You guessed the number after %d guess%s." % (num_guesses, suffix
        break
    if guess < secret_number:
        print("Too low!")
    else:
        print("Too high!")
```

## Answer to exercise 4

1.
   ```
   import re

   VALID_VARIABLE = re.compile('[a-zA-Z_][a-zA-Z0-9_]*')

   def validate_variable_name(name):
       return bool(VALID_VARIABLE.match(name))
   ```

2.
   ```
   import re

   WORDS = re.compile('(\S+)(\s+)(\S+)')

   def swap_words(s):
       return WORDS.sub(r'\3\2\1', s)
   ```

## Answer to exercise 5

1. Here is an example program:

```
import csv

with open("numbers.csv") as f_in:
    with open("numbers_new.csv", "w") as f_out:
        r = csv.reader(f_in)
        w = csv.writer(f_out)
        for row in r:
            w.writerow([row[0], row[2], row[1], sum(float(c) for c in row)])
```

## Answer to exercise 6

1. Here is an example program:

```
import sys
import argparse
import csv
import re

parser = argparse.ArgumentParser()
parser.add_argument("input", help="the input CSV file")
parser.add_argument("order", help="the desired column order; comma-separated; starting from zer
parser.add_argument("-o", "--output", help="the destination CSV file")

opts = parser.parse_args()

output_file = opts.output
if not output_file:
    output_file = re.sub("\.csv", "_reordered.csv", opts.input, re.IGNORECASE)
```

```
    try:
        new_row_indices = [int(i) for i in opts.order.split(',')]
    except ValueError:
        sys.exit("Unable to parse column list.")

    with open(opts.input) as f_in:
        with open(output_file, "w") as f_out:
            r = csv.reader(f_in)
            w = csv.writer(f_out)
            for row in r:
                new_row = []
                for i in new_row_indices:
                    try:
                        new_row.append(row[i])
                    except IndexError:
                        sys.exit("Invalid column: %d" % i)
                w.writerow(new_row)
```

CHAPTER 13

# Introduction to GUI programming with `tkinter`

We have previously seen how to write text-only programs which have a *command-line interface*, or CLI. Now we will briefly look at creating a program with a *graphical user interface*, or GUI. In this chapter we will use `tkinter`, a module in the Python standard library which serves as an interface to Tk, a simple *toolkit*. There are many other toolkits available, but they often vary across platforms. If you learn the basics of `tkinter`, you should see many similarities should you try to use a different toolkit.

We will see how to make a simple GUI which handles user input and output. GUIs often use a form of OO programming which we call *event-driven*: the program responds to *events*, which are actions that a user takes.

**Note:** in some Linux distributions, like Ubuntu and Debian, the `tkinter` module is packaged separately to the rest of Python, and must be installed separately.

## Event-driven programming

Anything that happens in a user interface is an *event*. We say that an event is *fired* whenever the user does something – for example, clicks on a button or types a keyboard shortcut. Some events could also be triggered by occurrences which are not controlled by the user – for example, a background task might complete, or a network connection might be established or lost.

Our application needs to monitor, or *listen* for, all the events that we find interesting, and respond to them in some way if they occur. To do this, we usually associate certain functions with particular events. We call a function which performs an action in response to an event an *event handler* – we *bind* handlers to events.

## `tkinter` basics

`tkinter` provides us with a variety of common GUI elements which we can use to build our interface – such as buttons, menus and various kinds of entry fields and display areas. We call these elements *widgets*. We are going to construct a *tree* of widgets for our GUI – each widget will have a parent widget, all the way up to the *root window* of our application. For example, a button or a text field needs to be *inside* some kind of containing window.

The widget classes provide us with a lot of default functionality. They have methods for configuring the GUI's appearance – for example, arranging the elements according to some kind of *layout* – and for handling various kinds of user-driven events. Once we have constructed the backbone of our GUI, we will need to customise it by integrating it with our internal application class.

Our first GUI will be a window with a label and two buttons:

```python
from tkinter import Tk, Label, Button

class MyFirstGUI:
    def __init__(self, master):
        self.master = master
        master.title("A simple GUI")

        self.label = Label(master, text="This is our first GUI!")
        self.label.pack()

        self.greet_button = Button(master, text="Greet", command=self.greet)
        self.greet_button.pack()

        self.close_button = Button(master, text="Close", command=master.quit)
        self.close_button.pack()

    def greet(self):
        print("Greetings!")

root = Tk()
my_gui = MyFirstGUI(root)
root.mainloop()
```

Try executing this code for yourself. You should be able to see a window with a title, a text label and two buttons – one which prints a message in the console, and one which closes the window. The window should have all the normal properties of any other window you encounter in your window manager – you are probably able to drag it around by the titlebar, resize it by dragging the frame, and maximise, minimise or close it using buttons on the titlebar.

**Note:** The *window manager* is the part of your operating system which handles windows. All the widgets inside a window, like buttons and other controls, may look different in every GUI toolkit, but the way that the window frames and title bars look and behave is determined by your window manager and should always stay the same.

We are using three widgets: `Tk` is the class which we use to create the *root* window – the main window of our application. Our application should only have one root, but it is possible for us to create other windows which are separate from the main window.

`Button` and `Label` should be self-explanatory. Each of them has a parent widget, which we pass in as the first parameter to the constructor – we have put the label and both buttons inside the main window, so they are the main window's children in the tree. We use the `pack` method on each widget to position it inside its parent – we will learn about different kinds of layout later.

All three of these widgets can display text (we could also make them display images). The label is a static element – it doesn't *do* anything by default; it just displays something. Buttons, however, are designed to cause something to happen when they are clicked. We have used the `command` keyword parameter when constructing each button to specify the function which should handle each button's click events – both of these functions are object methods.

We didn't have to write any code to make the buttons fire click events or to bind the methods to them explicitly. That functionality is already built into the button objects – we only had to provide the handlers. We also didn't have to write our own function for closing the window, because there is already one defined as a method on the window object. We did, however, write our own method for printing a message to the console.

There are many ways in which we could organise our application class. In this example, our class doesn't inherit from any `tkinter` objects – we use *composition* to associate our tree of widgets with our class. We could also use *inheritance* to extend one of the widgets in the tree with our custom functions.

`root.mainloop()` is a method on the main window which we execute when we want to run our application. This method will loop forever, waiting for events from the user, until the user exits the program – either by closing the window, or by terminating the program with a keyboard interrupt in the console.

### Widget classes

There are many different widget classes built into `tkinter` – they should be familiar to you from other GUIs:

- A `Frame` is a container widget which is placed inside a window, which can have its own border and background – it is used to group related widgets together in an application's layout.
- `Toplevel` is a container widget which is displayed as a separate window.
- `Canvas` is a widget for drawing graphics. In advanced usage, it can also be used to create custom widgets – because we can draw anything we like inside it, and make it interactive.
- `Text` displays formatted text, which can be editable and can have embedded images.
- A `Button` usually maps directly onto a user action – when the user clicks on a button, something should happen.
- A `Label` is a simple widget which displays a short piece of text or an image, but usually isn't interactive.
- A `Message` is similar to a `Label`, but is designed for longer bodies of text which need to be wrapped.
- A `Scrollbar` allows the user to scroll through content which is too large to be visible all at once.
- `Checkbutton`, `Radiobutton`, `Listbox`, `Entry` and `Scale` are different kinds of input widgets – they allow the user to enter information into the program.
- `Menu` and `Menubutton` are used to create pull-down menus.

## Layout options

The GUI in the previous example has a relatively simple layout: we arranged the three widgets in a single column inside the window. To do this, we used the `pack` method, which is one of the three different *geometry managers* available in `tkinter`. We have to use one of the available geometry managers to specify a position for each of our widgets, otherwise the widget will not appear in our window.

By default, `pack` arranges widgets vertically inside their parent container, from the top down, but we can change the alignment to the bottom, left or right by using the optional `side` parameter. We can mix different alignments in the same container, but this may not work very well for complex layouts. It should work reasonably well in our simple case, however:

```python
from tkinter import LEFT, RIGHT

# (...)

self.label.pack()
self.greet_button.pack(side=LEFT)
self.close_button.pack(side=RIGHT)
```

We can create quite complicated layouts with `pack` by grouping widgets together in frames and aligning the groups to our liking – but we can avoid a lot of this complexity by using the `grid` method instead. It allows us to position widgets in a more flexible way, using a *grid layout*. This is the geometry manager recommended for complex interfaces:

```python
from tkinter import W

# (...)
```

```
self.label.grid(columnspan=2, sticky=W)
self.greet_button.grid(row=1)
self.close_button.grid(row=1, column=1)
```

We place each widget in a cell inside a table by specifying a row and a column – the default row is the first available empty row, and the default column is `0`.

If a widget is smaller than its cell, we can customise how it is aligned using the `sticky` parameter – the possible values are the cardinal directions (`N`, `S`, `E` and `W`), which we can combine through addition. By default, the widget is centered both vertically and horizontally, but we can make it *stick* to a particular side by including it in the `sticky` parameter. For example, `sticky=W` will cause the widget to be left-aligned horizontally, and `sticky=W+E` will cause it to be stretched to fill the whole cell horizontally. We can also specify corners using `NE`, `SW`, etc..

To make a widget span multiple columns or rows, we can use the `columnspan` and `rowspan` options – in the example above, we have made the label span two columns so that it takes up the same space horizontally as both of the buttons underneath it.

**Note:** Never use both `pack` and `grid` inside the same window. The algorithms which they use to calculate widget positions are not compatible with each other, and your program will hang forever as `tkinter` tries unsuccessfully to create a widget layout which satisfies both of them.

The third geometry manager is `place`, which allows us to provide explicit sizes and positions for widgets. It is seldom a good idea to use this method for ordinary GUIs – it's far too inflexible and time consuming to specify an absolute position for every element. There are some specialised cases, however, in which it can come in useful.

## Custom events

So far we have only bound event handlers to events which are defined in `tkinter` by default – the `Button` class already knows about button clicks, since clicking is an expected part of normal button behaviour. We are not restricted to these particular events, however – we can make widgets listen for other events and bind handlers to them, using the `bind` method which we can find on every widget class.

Events are uniquely identified by a sequence name in string format – the format is described by a mini-language which is not specific to Python. Here are a few examples of common events:

- `"<Button-1>"`, `"<Button-2>"` and `"<Button-3>"` are events which signal that a particular mouse button has been pressed while the mouse cursor is positioned over the widget in question. *Button 1* is the left mouse button, *Button 3* is the right, and *Button 2* the middle button – but remember that not all mice have a middle button.
- `"<ButtonRelease-1>"` indicates that the left button has been released.
- `"<B1-Motion>"` indicates that the mouse was moved while the left button was pressed (we can use *B2* or *B3* for the other buttons).
- `"<Enter>"` and `"<Leave>"` tell us that the mouse curson has entered or left the widget.
- `"<Key>"` means that any key on the keyboard was pressed. We can also listen for specific key presses, for example `"<Return>"` (the *enter* key), or combinations like `"<Shift-Up>"` (*shift-up-arrow*). Key presses of most printable characters are expressed as the bare characters, without brackets – for example, the letter `a` is just `"a"`.
- `"<Configure>"` means that the widget has changed size.

We can now extend our simple example to make the label interactive – let us make the label text cycle through a sequence of messages whenever it is clicked:

```
from tkinter import Tk, Label, Button, StringVar

class MyFirstGUI:
    LABEL_TEXT = [
        "This is our first GUI!",
        "Actually, this is our second GUI.",
        "We made it more interesting...",
        "...by making this label interactive.",
        "Go on, click on it again.",
    ]
    def __init__(self, master):
        self.master = master
        master.title("A simple GUI")

        self.label_index = 0
        self.label_text = StringVar()
        self.label_text.set(self.LABEL_TEXT[self.label_index])
        self.label = Label(master, textvariable=self.label_text)
        self.label.bind("<Button-1>", self.cycle_label_text)
        self.label.pack()

        self.greet_button = Button(master, text="Greet", command=self.greet)
        self.greet_button.pack()

        self.close_button = Button(master, text="Close", command=master.quit)
        self.close_button.pack()

    def greet(self):
        print("Greetings!")

    def cycle_label_text(self, event):
        self.label_index += 1
        self.label_index %= len(self.LABEL_TEXT) # wrap around
        self.label_text.set(self.LABEL_TEXT[self.label_index])

root = Tk()
my_gui = MyFirstGUI(root)
root.mainloop()
```

Updating a label's text is a little convoluted – we can't simply update the text using a normal Python string. Instead, we have to provide the label with a special `tkinter` string variable object, and set a new value on the object whenever we want the text in the label to change.

We have defined a handler which cycles to the next text string in the sequence, and used the `bind` method of the label to bind our new handler to left clicks on the label. It is important to note that this handler takes an additional parameter – an event object, which contains some information about the event. We could use the same handler for many different events (for example, a few similar events which happen on different widgets), and use this parameter to distinguish between them. Since in this case we are only using our handler for one kind of event, we will simply ignore the event parameter.

## Putting it all together

Now we can use all this information to create a simple calculator. We will allow the user to enter a number in a text field, and either add it to or subtract it from a running total, which we will display. We will also allow the user to reset the total:

```
from tkinter import Tk, Label, Button, Entry, IntVar, END, W, E

class Calculator:

    def __init__(self, master):
        self.master = master
        master.title("Calculator")

        self.total = 0
        self.entered_number = 0

        self.total_label_text = IntVar()
        self.total_label_text.set(self.total)
        self.total_label = Label(master, textvariable=self.total_label_text)

        self.label = Label(master, text="Total:")

        vcmd = master.register(self.validate) # we have to wrap the command
        self.entry = Entry(master, validate="key", validatecommand=(vcmd, '%P'))

        self.add_button = Button(master, text="+", command=lambda: self.update("add"))
        self.subtract_button = Button(master, text="-", command=lambda: self.update("subtract"))
        self.reset_button = Button(master, text="Reset", command=lambda: self.update("reset"))

        # LAYOUT

        self.label.grid(row=0, column=0, sticky=W)
        self.total_label.grid(row=0, column=1, columnspan=2, sticky=E)

        self.entry.grid(row=1, column=0, columnspan=3, sticky=W+E)

        self.add_button.grid(row=2, column=0)
        self.subtract_button.grid(row=2, column=1)
        self.reset_button.grid(row=2, column=2, sticky=W+E)

    def validate(self, new_text):
        if not new_text: # the field is being cleared
            self.entered_number = 0
            return True

        try:
            self.entered_number = int(new_text)
            return True
        except ValueError:
            return False

    def update(self, method):
        if method == "add":
            self.total += self.entered_number
        elif method == "subtract":
            self.total -= self.entered_number
        else: # reset
            self.total = 0

        self.total_label_text.set(self.total)
        self.entry.delete(0, END)

root = Tk()
```

```
my_gui = Calculator(root)
root.mainloop()
```

We have defined two methods on our class: the first is used to validate the contents of the entry field, and the second is used to update our total.

## Validating text entry

Our `validate` method checks that the contents of the entry field are a valid integer: whenever the user types something inside the field, the contents will only change if the new value is a valid number. We have also added a special exception for when the value is nothing, so that the field can be cleared (by the user, or by us). Whenever the value of the field changes, we store the integer value of the contents in `self.entered_number`. We have to perform the conversion at this point anyway to see if it's a valid integer – if we store the value now, we won't have to do the conversion again when it's time to update the total.

How do we connect this validation function up to our entry field? We use the `validatecommand` parameter. The function we use for this command must return `True` if the entry's value is allowed to change and `False` otherwise, and it *must* be wrapped using a widget's `register` method (we have used this method on the window object).

We can also optionally specify arguments which must be passed to the function – to do this, we pass in a tuple containing the function and a series of strings which contain special codes. When the function is called, these codes will be replaced by different pieces of information about the change which is being made to the entry value. In our example, we only care about one piece of information: what the new value is going to be. The code string for this is `'%P'`, so we add it into the tuple.

Another optional parameter which is passed to `Entry` is `validate`, which specifies when validation should occur. the default value is `'none'` (a string value, not Python's `None`!), which means that no validation should be done. We have selected `'key'`, which will cause the entry to be validated whenever the user types something inside it – but it will also be triggered when we clear the entry from inside our `update` method.

## Updating the total

We have written a single handler for updating the total, because what we have to do in all three cases is very similar. However, the way that we update the value depends on which button was pressed – that's why our handler needs a parameter. This presents us with a problem – unfortunately, `tkinter` has no option for specifying parameters to be passed to button commands (or *callbacks*). We can solve the problem by wrapping the handler in three different functions, each of which calls the handler with a different parameter when it is called. We have used lambda functions to create these wrappers because they are so simple.

Inside the handler, we first update our running total using the integer value of the entry field (which is calculated and stored inside the `validate` method – note that we initialise it to zero in the `__init__` method, so it's safe for the user to press the buttons without typing anything). We know how to update the total because of the parameter which is passed into the handler.

Once we have updated the total, we need to update the text displayed by the label to show the new total – we do this by setting the new value on the `IntVar` linked to the label as its text variable. This works just like the `StringVar` in the previous example, except that an `IntVar` is used with integer values, not strings.

Finally, once we have used the number the user entered, we clear it from the entry field using the entry widget's delete method by deleting all the characters from the first index (zero) to the end (`END` is a constant defined by `tkinter`). We should also clear our internal value for the last number to be entered – fortunately, our deletion triggers the validation method, which already resets this number to zero if the entry is cleared.

### Exercise 1

1. Explain why we needed to use lambdas to wrap the function calls in the last example. Rewrite the button definitions to replace the lambdas with functions which have been written out in full.
2. Create a GUI for the guessing game from exercise 3 in the previous chapter.

## Answers to exercises

### Answer to exercise 1

1. The lambdas are necessary because we need to pass *functions* into the button constructors, which the button objects will be able to call later. If we used the bare function calls, we would be calling the functions and passing their return values (in this case, `None`) into the constructors. Here is an example of how we can rewrite this code fragment with full function definitions:

```
def update_add():
    self.update("add")

def update_subtract():
    self.update("subtract")

def update_reset():
    self.update("reset")

self.add_button = Button(master, text="+", command=update_add)
self.subtract_button = Button(master, text="-", command=update_subtract)
self.reset_button = Button(master, text="Reset", command=update_reset)
```

2. Here is an example program:

```
import random
from tkinter import Tk, Label, Button, Entry, StringVar, DISABLED, NORMAL, END, W, E

class GuessingGame:
    def __init__(self, master):
        self.master = master
        master.title("Guessing Game")

        self.secret_number = random.randint(1, 100)
        self.guess = None
        self.num_guesses = 0

        self.message = "Guess a number from 1 to 100"
        self.label_text = StringVar()
        self.label_text.set(self.message)
        self.label = Label(master, textvariable=self.label_text)

        vcmd = master.register(self.validate) # we have to wrap the command
        self.entry = Entry(master, validate="key", validatecommand=(vcmd, '%P'))

        self.guess_button = Button(master, text="Guess", command=self.guess_number)
        self.reset_button = Button(master, text="Play again", command=self.reset, state=DISABLED

        self.label.grid(row=0, column=0, columnspan=2, sticky=W+E)
        self.entry.grid(row=1, column=0, columnspan=2, sticky=W+E)
        self.guess_button.grid(row=2, column=0)
        self.reset_button.grid(row=2, column=1)

    def validate(self, new_text):
        if not new_text: # the field is being cleared
            self.guess = None
            return True

        try:
            guess = int(new_text)
            if 1 <= guess <= 100:
                self.guess = guess
                return True
            else:
                return False
        except ValueError:
            return False

    def guess_number(self):
        self.num_guesses += 1

        if self.guess is None:
            self.message = "Guess a number from 1 to 100"

        elif self.guess == self.secret_number:
            suffix = '' if self.num_guesses == 1 else 'es'
            self.message = "Congratulations! You guessed the number after %d guess%s." % (self.n
            self.guess_button.configure(state=DISABLED)
            self.reset_button.configure(state=NORMAL)

        elif self.guess < self.secret_number:
            self.message = "Too low! Guess again!"
        else:
            self.message = "Too high! Guess again!"

        self.label_text.set(self.message)

    def reset(self):
        self.entry.delete(0, END)
        self.secret_number = random.randint(1, 100)
        self.guess = 0
        self.num_guesses = 0

        self.message = "Guess a number from 1 to 100"
        self.label_text.set(self.message)

        self.guess_button.configure(state=NORMAL)
        self.reset_button.configure(state=DISABLED)

root = Tk()
my_gui = GuessingGame(root)
root.mainloop()
```

CHAPTER 14

# Sorting, searching and algorithm analysis

## Introduction

We have learned that in order to write a computer program which performs some task we must construct a suitable algorithm. However, whatever algorithm we construct is unlikely to be unique – there are likely to be many possible algorithms which can perform the same task. Are some of these algorithms in some sense better than others? Algorithm analysis is the study of this question.

In this chapter we will analyse four algorithms; two for each of the following common tasks:

- *sorting*: ordering a list of values
- *searching*: finding the position of a value within a list

Algorithm analysis should begin with a clear statement of the task to be performed. This allows us both to check that the algorithm is correct and to ensure that the algorithms we are comparing perform the same task.

Although there are many ways that algorithms can be compared, we will focus on two that are of primary importance to many data processing algorithms:

- *time complexity*: how the number of steps required depends on the size of the input
- *space complexity*: how the amount of extra memory or storage required depends on the size of the input

**Note:** Common sorting and searching algorithms are widely implemented and already available for most programming languages. You will seldom have to implement them yourself outside of the exercises in these notes. It is nevertheless important for you to understand these basic algorithms, because you are likely to use them within your own programs – their space and time complexity will thus affect that of your own algorithms. Should you need to select a specific sorting or searching algorithm to fit a particular task, you will require a good understanding of the available options.

## Sorting algorithms

The sorting of a list of values is a common computational task which has been studied extensively. The classic description of the task is as follows:

> Given a *list of values* and a function that *compares two values*, order the values in the list from smallest to largest.

The values might be integers, or strings or even other kinds of objects. We will examine two algorithms:

- *Selection sort*, which relies on repeated *selection* of the next smallest item
- *Merge sort*, which relies on repeated *merging* of sections of the list that are already sorted

Other well-known algorithms for sorting lists are *insertion sort*, *bubble sort*, *heap sort*, *quicksort* and *shell sort*.

There are also various algorithms which perform the sorting task for restricted kinds of values, for example:

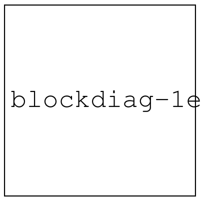
- *Counting sort*, which relies on the values belonging to a small set of items
- *Bucket sort*, which relies on the ability to map each value to one of a small set of items
- *Radix sort*, which relies on the values being sequences of digits

If we restrict the task, we can enlarge the set of algorithms that can perform it. Among these new algorithms may be ones that have desirable properties. For example, *Radix sort* uses fewer steps than any generic sorting algorithm.

## Selection sort

To order a given list using selection sort, we repeatedly select the smallest remaining element and move it to the end of a growing sorted list.

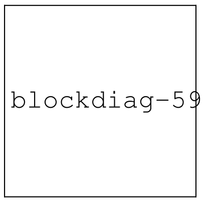
To illustrate selection sort, let us examine how it operates on a small list of four elements:


blockdiag-1e61cab6fdf1866858dd67cf79da0de4bbe32f0e.png

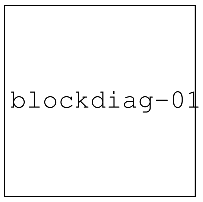
Initially the entire list is unsorted. We will use the front of the list to hold the sorted items – to avoid using extra storage space – but at the start this sorted list is empty.

First we must find the smallest element in the unsorted portion of the list. We take the first element of the unsorted list as a candidate and compare it to each of the following elements in turn, replacing our candidate with any element found to be smaller. This requires 3 comparisons and we find that element 1.5 at position 2 is smallest.

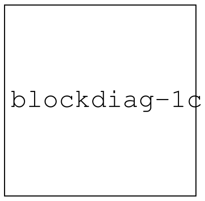
Now we will swap the first element of our unordered list with the smallest element. This becomes the start of our ordered list:


blockdiag-59cc273248938d32529988e9189d103c67c86bbc.png

We now repeat our previous steps, determining that 2.7 is the smallest remaining element and swapping it with 3.8 – the first element of the current unordered section – to get:


blockdiag-0116a491a5344b5d46426992b649ea3672194ba8.png

Finally, we determine that 3.8 is the smallest of the remaining unordered elements and swap it with 7.2:


blockdiag-1c7238b5b91f6a1f6c062e4baf72a740f748467d.png

The table below shows the number of operations of each type used in sorting our example list:

| Sorted List Length | Comparisons | Swaps | Assign smallest candidate |
|---|---|---|---|
| 0 -> 1 | 3 | 1 | 3 |
| 1 -> 2 | 2 | 1 | 2 |
| 2 -> 3 | 1 | 1 | 2 |
| **Total** | **6** | **3** | **7** |

Note that the number of *comparisons* and the number of *swaps* are independent of the contents of the list (this is true for selection sort but not necessarily for other sorting algorithms) while the number of times we have to assign a new value to the smallest candidate depends on the contents of the list.

More generally, the algorithm for selection sort is as follows:

1. Divide the list to be sorted into a sorted portion at the front (initially empty) and an unsorted portion at the end (initially the whole list).
2. Find the smallest element in the unsorted list:
1. Select the first element of the unsorted list as the initial candidate.
2. Compare the candidate to each element of the unsorted list in turn, replacing the candidate with the current element if the current element is smaller.
3. Once the end of the unsorted list is reached, the candidate is the smallest element.
3. Swap the smallest element found in the previous step with the first element in the unsorted list, thus extending the sorted list by one element.
4. Repeat the steps 2 and 3 above until only one element remains in the unsorted list.

**Note:** The *Selection sort* algorithm as described here has two properties which are often desirable in sorting algorithms.

The first is that the algorithm is *in-place*. This means that it uses essentially no extra storage beyond that required for the input (the unsorted list in this case). A little extra storage may be used (for example, a temporary variable to hold the candidate for the smallest element). The important property is that the extra storage required should not increase as the size of the input increases.

The second is that the sorting algorithm is *stable*. This means that two elements which are equal retain their initial relative ordering. This becomes important if there is additional information attached to the values being sorted (for example, if we are sorting a list of people using a comparison function that compares their dates of birth). Stable sorting algorithms ensure that sorting an already sorted list leaves the order of the list unchanged, even in the presence of elements that are treated as equal by the comparison.

## Exercise 1

Complete the following code which will perform a selection sort in Python. ”...” denotes missing code that should be filled in:

```
def selection_sort(items):
    """Sorts a list of items into ascending order using the
       selection sort algoright.
       """
    for step in range(len(items)):
        # Find the location of the smallest element in
        # items[step:].
        location_of_smallest = step
        for location in range(step, len(items)):
            # TODO: determine location of smallest
            ...
        # TODO: Exchange items[step] with items[location_of_smallest]
        ...
```

## Exercise 2

Earlier in this section we counted the number of *comparisons*, *swaps* and *assignments* used in our example.

1. How many swaps are performed when we apply selection sort to a list of N items?
2. How many comparisons are performed when we apply selection sort to a list of N items?
   (a) How many comparisons are performed to find the smallest element when the unsorted portion of the list has M items?
   (b) Sum over all the values of M encountered when sorting the list of length N to find the total number of comparisons.
3. The number of assignments (to the candidate smallest number) performed during the search for a smallest element is at most one more than the number of comparisons. Use this to find an upper limit on the total number of assignments performed while sorting a list of length N.
4. Use the results of the previous question to find an upper bound on the total number of operations (swaps, comparisons and assignments) performed. Which term in the number of operations will dominate for large lists?

## Merge sort

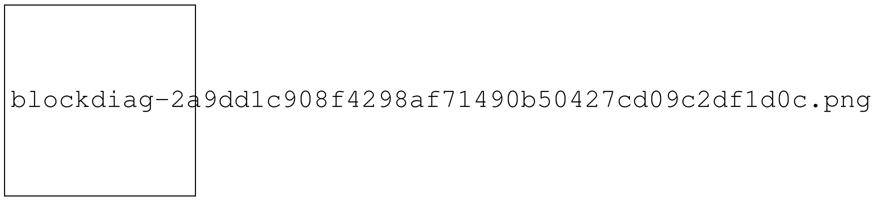
When we use merge sort to order a list, we repeatedly merge sorted sub-sections of the list – starting from sub-sections consisting of a single item each.

We will see shortly that merge sort requires significantly fewer operations than selection sort.

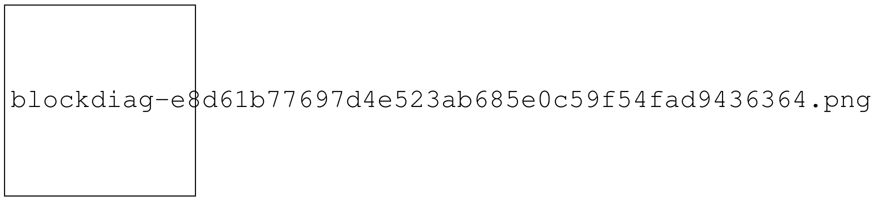
Let us start once more with our small list of four elements:

blockdiag-5876e48223e11f2794cd3eef4bdb15b7801257e5.png

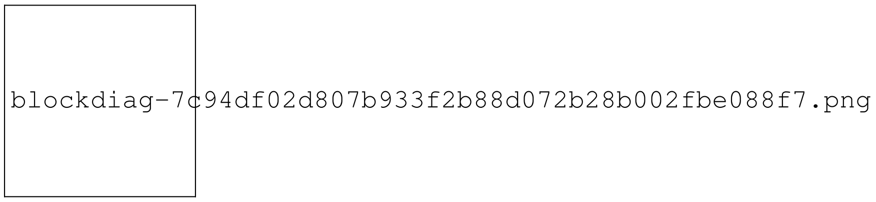
First we will merge the two sections on the left into the temporary storage. Imagine the two sections as two sorted piles of cards – we will merge the two piles by repeatedly taking the smaller of the top two cards and placing it at the end of the merged list in the temporary storage. Once one of the two piles is empty, the remaining items in the other pile can just be placed on the end of the merged list:



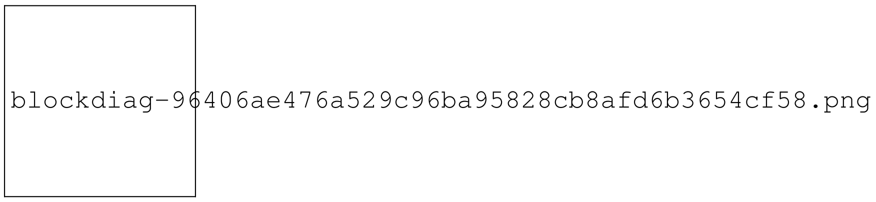
Next we copy the merged list from the temporary storage back into the portion of the list originally occupied by the merged subsections:



We repeat the procedure to merge the second pair of sorted sub-sections:



Having reached the end of the original list, we now return to the start of the list and begin to merge sorted sub-sections again. We repeat this until the entire list is a single sorted sub-section. In our example, this requires just one more merge:



Notice how the size of the sorted sections of the list doubles after every iteration of merges. After M steps the size of the sorted sections is $2^M$. Once $2^M$ is greater than N, the entire list is sorted. Thus, for a list of size N, we need M equals $\log_2 N$ interations to sort the list.

Each iteration of merges requires a complete pass through the list and each element is copied twice – once into the temporary storage and once back into the original list. As long as there are items left in both sub-sections in each pair, each copy into the temporary list also requires a comparison to pick which item to copy. Once one of the lists runs out, no comparisons are needed. Thus each pass requires 2N copies and roughly N comparisons (and certainly no more than N).

The total number of operations required for our merge sort algorithm is the product of the number of operations in each pass and the number of passes – i.e. $2N\log_2 N$ copies and roughly $N\log_2 N$ comparisons.

The algorithm for merge sort may be written as this list of steps:

1. Create a temporary storage list which is the same size as the list to be sorted.
2. Start by treating each element of the list as a sorted one-element sub-section of the original list.
3. Move through all the sorted sub-sections, merging adjacent pairs as follows:
   (a) Use two variables to point to the indices of the smallest uncopied items in the two sorted sub-sections, and a third variable to point to the index of the start of the temporary storage.

(b) Copy the smaller of the two indexed items into the indicated position in the temporary storage. Increment the index of the sub-section from which the item was copied, and the index into temporary storage.

(c) If all the items in one sub-section have been copied, copy the items remaining in the other sub-section to the back of the list in temporary storage. Otherwise return to step 3 ii.

(d) Copy the sorted list in temporary storage back over the section of the original list which was occupied by the two sub-sections that have just been merged.

4. If only a single sorted sub-section remains, the entire list is sorted and we are done. Otherwise return to the start of step 3.

### Exercise 3

Write a Python function that implements merge sort. It may help to write a separate function which performs merges and call it from within your merge sort implementation.

### Python's sorting algorithm

Python's default sorting algorithm, which is used by the built-in `sorted` function as well as the `sort` method of list objects, is called *Timsort*. It's an algorithm developed by Tim Peters in 2002 for use in Python. Timsort is a modified version of merge sort which uses insertion sort to arrange the list of items into conveniently mergeable sections.

**Note:** Tim Peters is also credited as the author of *The Zen of Python* – an attempt to summarise the early Python community's ethos in a short series of koans. You can read it by typing `import this` into the Python console.

## Searching algorithms

Searching is also a common and well-studied task. This task can be described formally as follows:

> Given a *list of values*, a function that *compares two values* and a *desired value*, find the position of the desired value in the list.

We will look at two algorithms that perform this task:

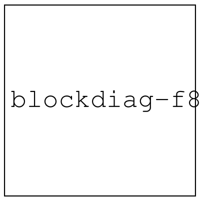
- *linear search*, which simply checks the values in sequence until the desired value is found
- *binary search*, which requires a sorted input list, and checks for the value in the middle of the list, repeatedly discarding the half of the list which contains values which are definitely either all larger or all smaller than the desired value

There are numerous other searching techniques. Often they rely on the construction of more complex data structures to facilitate repeated searching. Examples of such structures are *hash tables* (such as Python's dictionaries) and *prefix trees*. Inexact searches that find elements similar to the one being searched for are also an important topic.

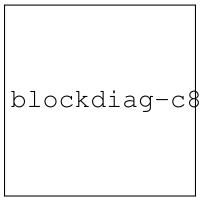
### Linear search

Linear search is the most basic kind of search method. It involves checking each element of the list in turn, until the desired element is found.

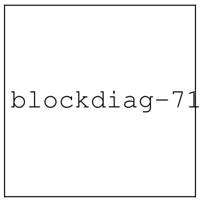
For example, suppose that we want to find the number 3.8 in the following list:


blockdiag-f89c0ad9fad40f3a5f55d040b2ea2ee966441749.png

We start with the first element, and perform a comparison to see if its value is the value that we want. In this case, 1.5 is not equal to 3.8, so we move onto the next element:


blockdiag-c89fc2a0f6ac9dc31c4733e1ba9612095141c63e.png

We perform another comparison, and see that 2.7 is also not equal to 3.8, so we move onto the next element:


blockdiag-711dbad019be24cc93ec7d4eafe67ce96c832ad0.png

We perform another comparison and determine that we have found the correct element. Now we can end the search and return the position of the element (index 2).

We had to use a total of 3 comparisons when searching through this list of 4 elements. How many comparisons we need to perform depends on the total length of the list, but also whether the element we are looking for is near the beginning or near the end of the list. In the worst-case scenario, if our element is the last element of the list, we will have to search through the entire list to find it.

If we search the same list many times, assuming that all elements are equally likely to be searched for, we will on average have to search through half of the list each time. The cost (in comparisons) of performing linear search thus scales linearly with the length of the list.

### Exercise 4

1. Write a function which implements linear search. It should take a list and an element as a parameter, and return the position of the element in the list. If the element is not in the list, the function should raise an exception. If the element is in the list multiple times, the function should return the first position.

## Binary search

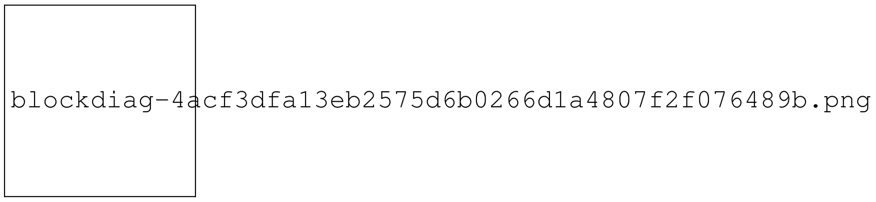
Binary search is a more efficient search algorithm which relies on the elements in the list being sorted. We apply the same search process to progressively smaller sub-lists of the original list, starting with the whole list and approximately halving the search area every time.

We first check the *middle* element in the list.

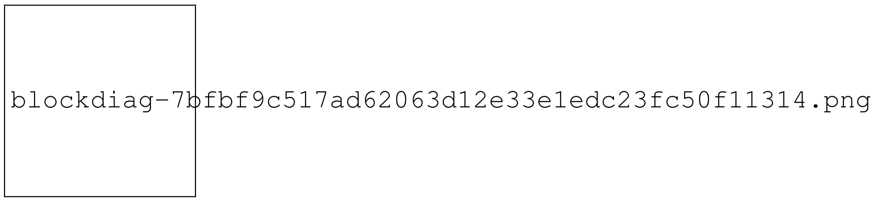
- If it is the value we want, we can stop.
- If it is *higher* than the value we want, we repeat the search process with the portion of the list *before* the middle element.

- If it is *lower* than the value we want, we repeat the search process with the portion of the list *after* the middle element.

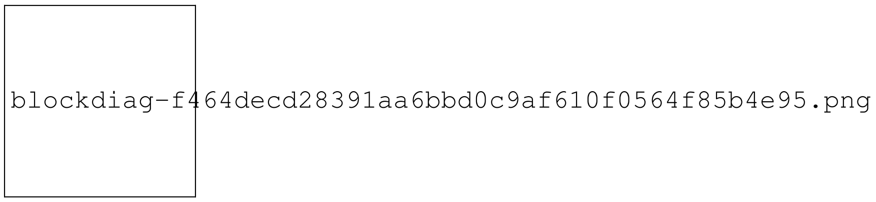
For example, suppose that we want to find the value 3.8 in the following list of 7 elements:



First we compare the element in the middle of the list to our value. 7.2 is *bigger* than 3.8, so we need to check the first half of the list next.



Now the first half of the list is our new list to search. We compare the element in the middle of this list to our value. 2.7 is *smaller* than 3.8, so we need to search the *second half* of this sublist next.



The second half of the last sub-list is just a single element, which is also the middle element. We compare this element to our value, and it is the element that we want.

We have performed 3 comparisons in total when searching this list of 7 items. The number of comparisons we need to perform scales with the size of the list, but much more slowly than for linear search – if we are searching a list of length N, the maximum number of comparisons that we will have to perform is $\log_2 N$.

### Exercise 5

1. Write a function which implements binary search. You may assume that the input list will be sorted. Hint: this function is often written recursively.

## Algorithm complexity and Big O notation

We commonly express the cost of an algorithm as a function of the number N of elements that the algorithm acts on. The function gives us an estimate of the number of operations we have to perform in order to use the algorithm on N elements – it thus allows us to predict how the number of required operations will increase as N increases. We use a function which is an *approximation* of the exact function – we simplify it as much as possible, so that only the most important information is preserved.

For example, we know that when we use linear search on a list of N elements, on average we will have to search through half of the list before we find our item – so the number of operations we will have to perform is N/2. However, the most important thing is that the algorithm scales *linearly* – as N increases, the cost of the algorithm increases in proportion to N, not $N^2$ or $N^3$. The constant factor of 1/2 is insignificant compared to the very large differences in cost between – for example – N and $N^2$, so we leave it out when we describe the cost of the algorithm.

We thus write the cost of the linear search algorithm as O(N) – we say that the cost is *on the order of N*, or just *order N*. We call this notation *big O notation*, because it uses the capital O symbol (for *order*).

We have dropped the constant factor 1/2. We would also drop any lower-order terms from an expression with multiple terms – for example, $O(N^3 + N^2)$ would be simplified to $O(N^3)$.

In the example above we calculated the *average* cost of the algorithm, which is also known as the *expected* cost, but it can also be useful to calculate the *best case* and *worst case* costs. Here are the best case, expected and worst case costs for the sorting and searching algorithms we have discussed so far:

| Algorithm | Best case | Expected | Worst case |
|---|---|---|---|
| Selection sort | $O(N^2)$ | $O(N^2)$ | $O(N^2)$ |
| Merge sort | O(N log N) | O(N log N) | O(N log N) |
| Linear search | O(1) | O(N) | O(N) |
| Binary search | O(1) | O(log N) | O(log N) |

What does O(1) mean? It means that the cost of an algorithm is *constant*, no matter what the value of N is. For both these search algorithms, the best case scenario happens when the first element to be tested is the correct element – then we only have to perform a single operation to find it.

In the previous table, big O notation has been used to describe the *time complexity* of algorithms. It can also be used to describe their *space complexity* – in which case the cost function represents the number of units of space required for storage rather than the required number of operations. Here are the space complexities of the algorithms above (for the worst case, and excluding the space required to store the input):

| Algorithm | Space complexity |
|---|---|
| Selection sort | O(1) |
| Merge sort | O(N) |
| Linear search | O(1) |
| Binary search | O(1) |

None of these algorithms require a significant amount of storage space in addition to that used by the input list, except for the merge sort – which, as we saw in a previous section, requires temporary storage which is the same size as the input (and thus scales linearly with the input size).

**Note:** The Python wiki has a summary of the time complexities of common operations on collections. You may also wish to investigate the `collections` module, which provides additional collection classes which are optimised for particular tasks.

**Note:** *Computational complexity theory* studies the inherent complexity of *tasks* themselves. Sometimes it is possible to prove that *any* algorithm that can perform a given task will require some minimum number of steps or amount of extra storage. For example, it can be shown that, given a list of arbitrary objects and only a comparison function with which to compare them, no sorting algorithm can use fewer than O(N log N) comparisons.

## Exercise 6

1. We can see from the comparison tables above that binary search is more efficient than linear search. Why would we ever use linear search? Hint: what property must a list have for us to be able to use a binary search on it?
2. Suppose that each of the following functions shows the average number of operations required to perform some algorithm on a list of length N. Give the big O notation for the time complexity of each algorithm:

(a) $4N^2 + 2N + 2$

(b) $N + \log N$

(c) $N \log N$

(d) 3

## Answers to exercises

### Answer to exercise 1

Completed selection sort implementation:

```
def selection_sort(items):
    """Sorts a list of items into ascending order using the
       selection sort algoright.
       """
    for step in range(len(items)):
        # Find the location of the smallest element in
        # items[step:].
        location_of_smallest = step
        for location in range(step, len(items)):
            # determine location of smallest
            if items[location] < items[location_of_smallest]:
                location_of_smallest = location
        # Exchange items[step] with items[location_of_smallest]
        temporary_item = items[step]
        items[step] = items[location_of_smallest]
        items[location_of_smallest] = temporary_item
```

### Answer to exercise 2

1. `N - 1` swaps are performed.
2. `(N - 1) * N / 2` comparisons are performed.

   (a) `M - 1` comparisons are performed finding the smallest element.

   (b) Summing `M - 1` from `2` to `N` gives:

```
        1 + 2 + 3 + ... + (N - 1)

        = (N - 1) * N / 2
```

3. At most `(N - 1) * N / 2 + (N - 1)` assignements are performed.
4. At most `N**2 + N - 2` operations are performed. For long lists the number of operations grows as `N**2`.

### Answer to exercise 3

1. Here is an example program:

```
    def merge(items, sections, temporary_storage):
        (start_1, end_1), (start_2, end_2) = sections
        i_1 = start_1
        i_2 = start_2
        i_t = 0

        while i_1 < end_1 or i_2 < end_2:
            if i_1 < end_1 and i_2 < end_2:
                if items[i_1] < items[i_2]:
                    temporary_storage[i_t] = items[i_1]
                    i_1 += 1
                else:  # the_list[i_2] >= items[i_1]
                    temporary_storage[i_t] = items[i_2]
                    i_2 += 1
                i_t += 1

            elif i_1 < end_1:
                for i in range(i_1, end_1):
                    temporary_storage[i_t] = items[i_1]
                    i_1 += 1
                    i_t += 1

            else:  # i_2 < end_2
                for i in range(i_2, end_2):
                    temporary_storage[i_t] = items[i_2]
                    i_2 += 1
                    i_t += 1

        for i in range(i_t):
            items[start_1 + i] = temporary_storage[i]


    def merge_sort(items):
        n = len(items)
        temporary_storage = [None] * n
        size_of_subsections = 1

        while size_of_subsections < n:
            for i in range(0, n, size_of_subsections * 2):
                i1_start, i1_end = i, min(i + size_of_subsections, n)
                i2_start, i2_end = i1_end, min(i1_end + size_of_subsections, n)
                sections = (i1_start, i1_end), (i2_start, i2_end)
                merge(items, sections, temporary_storage)
            size_of_subsections *= 2

        return items
```

## Answer to exercise 4

1. Here is an example program:

```
    def linear_search(items, desired_item):
        for position, item in enumerate(items):
            if item == desired_item:
                return position

        raise ValueError("%s was not found in the list." % desired_item)
```

## Answer to exercise 5

1. Here is an example program:

```
def binary_search(items, desired_item, start=0, end=None):
    if end == None:
        end = len(items)

    if start == end:
        raise ValueError("%s was not found in the list." % desired_item)

    pos = (end - start) // 2 + start

    if desired_item == items[pos]:
        return pos
    elif desired_item > items[pos]:
        return binary_search(items, desired_item, start=(pos + 1), end=end)
    else: # desired_item < items[pos]:
        return binary_search(items, desired_item, start=start, end=pos)
```

## Answer to exercise 6

1. The advantage of linear search is that it can be performed on an *unsorted* list – if we are going to examine all the values in turn, their order doesn't matter. It can be more efficient to perform a linear search than a binary search if we need to find a value *once* in a large unsorted list, because just sorting the list in preparation for performing a binary search could be more expensive. If, however, we need to find values in the same large list multiple times, sorting the list and using binary search becomes more worthwhile.

2. We drop all constant factors and less significant terms:

   (a) O($N^2$)

   (b) O(N)

   (c) O(N log N)

   (d) O(1)

CHAPTER 15

# Indices and tables

- genindex
- modindex
- search