The Coder's Apprentice

Learning Programming with Python 3

Pieter Spronck

The Coder's Apprentice

Learning Programming with Python 3

Pieter Spronck

Version 1.0.16

November 11, 2017

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book will (at some point in time) be available from `http://www.spronck.net/pythonbook`

The latest version of this book will always be available from `http://www.spronck.net/pythonbook`**.**

Preface

Computational technology causes the world to change rapidly.

Almost 30 years ago I got my first job as a computer programmer. At the time, only larger companies with a big administrative overload used computers. Or rather, "a computer," because it was rare for a company to have more than one. There were no personal computers, no Internet, no mobile phones. People still used typewriters.

In those 30 years, the way people work and live has undergone huge changes. That is exceptionally clear when looking at the kind of work that people do. Mailmen, for instance, delivered the mail twice per day when I was a kid – now they deliver mail twice per week, which means that the contingent of professional mailmen has been decimated. Bank offices are closed because banking can be done much easier online. Information desks can be manned by digital avatars or be replaced by online information systems. Large department stores go out of business because people make their purchases online, leading to an enormous decline in the need for having salespeople. And though this has currently caused a small increase in the demand for people who work in transportation, we can see self-driving cars on the horizon, replacing the need to have any chauffeurs at all.

These are all "low profile" jobs, but "high profile" jobs aren't safe either. I have taught programming to professional journalists, who told me that computers are taking over large parts of their jobs, writing basic articles and doing automated background research – they wanted to take my courses because they realized that without skills in digital technology, they would be out of a job in a few years time. Programs have been developed that take over a menial but oh-so time consuming part of lawyers' jobs, namely researching case histories. Computers can write music, produce paintings, and even sculpt – why would you have someone hammer away at a block of granite for six months when a 3D-printer can produce a sculpture with a few hours of work? Even designing and running scientific experiments has been offloaded to computers in some research domains.

In the 30 years in which I have been a professional worker, I have seen the job market change from hardly incorporating computers at all, to a situation in which the need for human employees has been reduced considerably – regardless the job. And that change has not come to an end yet.

This does not mean that there is no place for humans in the job market. It does mean, however, that only humans who can make contributions that a computer has a hard time making on its own, can be assured of a job. In the near future, employability will be invariably linked to the ability to integrate the power of humans and computers in a way that enhances both of them.

The problem is that to be able to use computers to improve the quality of one's work, it does not suffice to be able to use a word processor or spreadsheet. One should actually be able to

expand the capabilities of computers from the perspective of one's chosen profession. For example, a journalist who can only run a fact-finding computer program that someone else wrote, is not needed. However, a journalist who is able to expand a fact-finding program so that it can come up with facts from new sources, is an asset.

To be able to employ computers in such a way, one needs the skills to think and solve problems like a computer programmer. Having taught students computer programming for many years, I know that this does not come naturally to most. To acquire the necessary skills, students need to spend several intensive courses on the topic.

Considering the fact that universities and colleges are supposed to prepare students for the job market in which they have to function for 40 or more years, and considering the fact that in the very near future (if not right now already) the ability to incorporate the power of computers in any job is a necessity to being a valued worker, one would expect that "computer programming" is one of the basic courses that any student needs to take. Unfortunately, it is not. Typically, basic required courses are "scientific writing," "philosophy of science," and "statistics," but "computer programming" is still seen, by most education programs, as an optional skill. It is not.

In my view, any course program that does not make "computer programming" a required course, is doing its students a disservice, as it is not preparing them for the job market. Actually, I would prefer it if secondary, or even primary schools would incorporate such courses, as programming skills tend to be easier to learn at a younger age. The reason is that they need a particular way of creative thinking, which is harder to acquire when one is already used to solving problems in the reproductive ways that are normally taught at schools.

All students, regardless of their chosen topic, need to learn how to program. Not because we should raise a generation of computer programmers – professional programming is a specialization that only a few people need to be able to do. But the ability to create programs provides students with the skills to think and solve problems like a computer programmer, to gain insight in the possibilities and limitations of computers, and to leverage the power of computers in a particular domain in a uniquely human way.

The goal of this book is to teach anyone how to create useful programs in Python. It should be usable by secondary school students, and university and college students for whom computer programming is not naturally incorporated in their course program. Its aim is to give anyone the means to become proficient in programming, and as such get prepared to perform well in the 21st century job market.

Pieter Spronck
May 2, 2016
Maastricht, The Netherlands

Pieter Spronck is a Professor of Computer Science at Tilburg University, The Netherlands.

Acknowledgments

Many thanks to Allen B. Downey, who wrote the excellent Python 2 book *Think Python: How to Think Like a Computer Scientist*. I myself learned Python programming from his book, and used the LaTeX template that he graciously provided as the basis for this book. Downey recently released a Python 3 version of his book. If you are already familiar with

programming in general and just want to get to know Python, his book might be the way to go.

I am grateful to Peter Wentworth, who produced a Python 3 version of Downey's book. Peter uses a particular style of teaching that I find does not work too well with the students I have had, but I definitely got a lot of information from his book.

Many thanks also to Guido van Rossum, the original creator of Python. I love the concept of programming, but very few programming languages are actually a joy to use. Python is one of them, and for that I am grateful.

Thanks also to Ákos Kádár, Nanne van Noord, and Sander Wubben, who worked with me on an early version of a Python course, on which I later based this book.

Thanks to the members of Monty Python, whose television shows and audio recordings taught me English in a highly enjoyable way. Their show gave Python its name, and I have used quotes of their shows in some of the demonstrations and exercises in this book.

Many thanks to Myrthe Spronck, for creating the website for this book, found at `http://www.spronck.net/pythonbook`.

Thanks to all the contributors (listed below) who sent in corrections and suggestions.

If you have a suggestion or correction, please send email to `pythonbook@spronck.net` (not to be used for assistance with programming problems, of course – there are plenty of places on the Internet where you can get such help), or leave a message at the forum `http://www.spronck.net/forum`. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

Contributor list

- "oajns" indicated some spelling mistakes in Chapter 9 and in Appendices C and D (fixed in version 1.0.4).
- Larry Cali pointed out an error in the code for Exercise 4.3, which could give problems with floating-point values which Python cannot store exactly. I fixed the exercise and made a remark on this in Chapter 3 (fixed in version 1.0.5).
- Isaac Kramer noted a problem in Exercise 9.5, which made the issue in the code unnoticable. I fixed this to make the error actually occur as I explain in the Answers section (fixed in version 1.0.6).
- Ruud van Cruchten indicated that my discussion of providing multi-line commentary in Chapter 4 was incomplete and could lead to problems. I have extended the text in this respect (fixed in version 1.0.7).
- Nade Kang pointed out that the answer to Exercise 7.9 (second guessing game) could be confusing. I changed the code a little to compensate (fixed in version 1.0.7).
- Shiyu Zhang noticed that listing 8.16 contained useless parameters. I corrected this (fixed in version 1.0.8).
- Mustafa Amjed indicated several spelling and logical mistakes in the first 100 pages (fixed in version 1.0.8).
- Woodgirl Martyr indicated a spelling mistake in Chapter 1 (fixed in version 1.0.9).

- Claudia Dai pointed out a small mistake in the answer to Exercise 10.1 (counting vowels; fixed in version 1.0.9).
- Several of my students suggested adding flow charts to the chapters on conditions and iterations, as they would help understanding of how these concepts work. I followed that suggestion (added to version 1.0.9).
- Mauro Crociara pointed out multiple typos and gave many ideas for improvements (incorporated in version 1.0.11).
- Chris Spinks noticed some problems with the answer for Exercise 21.4, the extended fruit basket, and the regular expressions in the answers to Exercises 25.3 and 25.4, where one is supposed to extract names from texts (fixed in version 1.0.12).
- Patrick Vekemans noticed an error in the code in Subsection 7.3.2 (fixed in version 1.0.13).
- Jose Perez-Carballo pointed out to me that the list of reserved words that I presented was actually the Python 2 list, which has undergone a few changes in Python 3 (fixed in version 1.0.13). He also pointed out a typo (fixed in version 1.0.14).
- Jos Kaats indicated an erroneous detail in the calling of functions from functions (fixed in 1.0.14).
- Luis Mendo Tomas had quite a few remarks which all lead to changes in the book, in particular the inclusion of a section on default values for function parameters (version 1.0.14).
- Abdulkader Abdullah pointed out a mistake in the ranges used in the answer to exercise 14.2 (fixed in version 1.0.16).
- Corné Heeren provided many ideas for improvements. I particularly appreciate his simple way to determine the middle of three numbers in the chapter on functions (version 1.0.16).

Contents

Chapter 1

Introduction

Computers are wonderful machines. While most machines (cars, television sets, microwaves) have a specific purpose which they excel at accomplishing, computers are purposeless machines that can be taught to accomplish anything. The power to make a computer do your bidding is called "programming."

Nowadays, in any scientific and professional endeavor, people have to deal with large volumes of data. Those who are able to leverage the power of computers to make use of such data, i.e., those who can program, are far better able to do their jobs than those who are not. In fact, it can be argued that in the very near future, those who do not possess programming skills will become unemployable. Therefore, I feel that it is necessary for anyone to acquire basic skills in this area during their education.

Being able to write computer programs not only entails knowing what specific code statements mean and do; it also entails having the ability think like a programmer, and to analyze problems from the perspective of solving them with a computer program. Such skills cannot be learned from a book, they can only be learned by actually creating programs.

This book has been designed to teach the basics of the Python 3 computer language. Students will not only learn to use the language, but also do their first practical exercises with it.

The book is not only catering towards people who are naturally inclined towards programming. It is meant to also be used by those who have no particular aptitude for programming. This is exemplified by texts which try to be extensive and foresee problems that might arise when trying to understand certain concepts.

1.1 How to use this book

This book is meant to be used as a course. It is not meant as a Python language reference. You do not need a book as a language reference, as an excellent language reference can be found on the Internet (`http://docs.python.org`).

The chapters of this book are written to be studied in sequential order. For a brief course on the basics of the Python language, using it for "imperative programming," you should

study variables and expressions, conditions and loops, functions, string handling, lists and dictionaries, and files. I.e., you can limit yourself to Chapters 1 to 19, whereby the Chapters 9 (Recursion), 14 (Sets), 17 (Exceptions), 18 (Binary files), and 19 (Bitwise operators) may be considered advanced material, which you can skip until you need them (though I highly recommend that you at least try to understand recursion, as it helps solving some of the exercises in later chapters).

For an advanced course on the basics of the Python language, you will have to delve into object orientation, meaning that you also have to study Chapters 20 to 23, whereby Chapter 23 (Iterators and Generators) can be considered optional material.

The remainder of the chapters are all useful, but optional material, from which you can pick and choose, though I recommend that you at least read through them to understand the topics that they cover. Future editions of this book may have extra optional material added to the end.

When studying this book, you should have a computer with Python installed at hand (Chapter 2 explains how to get Python for your computer). The book contains many small and larger exercises, and you should do all of those while studying. There is no way that you will learn how to program if you skip the exercises. More on the exercises follows later in this chapter (Section 1.9).

Many of the code snippets in this book – in particular all the answers to the exercises and all the slightly longer pieces of code – have a file name listed as a caption. This means that this code is available under that particular file name from the website associated with this book (`http://www.spronck.net/pythonbook`). You can download this code and load it immediately in the editor that you are using if you so wish.

Note that copying and pasting code from a PDF file to an editor will, in general, not work. Text in a PDF file is not stored in such a way that spaces are inserted in the correct places when you copy code. So you must either manually type in code, or use the listings that are provided as separate files.

1.2 Prerequisites and assumptions

This book assumes that you have no programming skills at all, but are willing to learn. You should also have the ability to think in abstractions.

You should realize that learning how to program might take a significant time investment. It does not suffice to just read the material and do the occasional small exercises. You will have to practice with the material and also do larger exercises, if you really want to gain the ability to create programs. If you stick to the basic chapters (everything up to dealing with text files), if you have no programming knowledge at all, you should count on having to invest between 100 and 200 hours to get to the finish, depending on aptitude. Learning everything that the book has to offer will take between 200 and 400 hours.

Note that this book will not try to teach you to be a professional programmer. It teaches the initial skills that any professional programmer also acquired during his or her education. After teaching those initial skills, the book ends. For most people, this is enough to deal with programming tasks they encounter, and provides a sufficient basis to learn more if there is a need.

1.3 Why Python?

Python has become a language of choice for teaching people how to program. It is a powerful language, that is easy to use, and that offers all the possibilities that any other computer language offers. It is easily portable between different operating systems. It is freely available. For beginning programmers, it has the advantage that it enforces writing readable code. Python is also a language that is used for many practical applications, either as a basis for complete programs, or as an extension to programs written in a different language.

The main advantage of using Python is that it allows you to focus on "thinking like a programmer," rather than learning all the arcane intricacies of a language. Here is an example of the difference between using Python, and using some other popular programming languages: The first program that anybody writes in any language, is *Hello World*. This is a program that displays the text "Hello, world!" on the screen. In the highly popular computer language C++, *Hello World* is coded as follows:

```
#include <iostream>
int main() {
    std::cout << "Hello, world!";
}
```

In C#, Microsoft's popular variant of C++, it is:

```
using System;
namespace HelloWorld {
    class Hello {
        static void Main() {
            Console.WriteLine( "Hello, world!" );
            Console.ReadKey();
        }
    }
}
```

In Objective-C, Apple's C++ variation, the code becomes even worse:

```
#import <Foundation/Foundation.h>
int main ( int argc, const char * argv[] ) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSLog (@"Hello, world!");
    [pool drain];
    return 0;
}
```

In Java, which is taught as the first language to many computer science students, it is:

```
class Hello {
    public static void main( String[] args ) {
        System.out.println( "Hello, world!" );
    }
}
```

Now compare this to writing *Hello World* in Python:

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

I think we can agree that the Python version of this program is much more readable and understandable – even without knowing the language – than any of the other languages.

1.4 Python's limitations as a programming language

Python is a general-purpose programming language. This means that it can be used for anything and everything you would ever want to do with computer languages. Can you therefore conclude that once you have learned Python, you will never need to learn another language?

The answer is that it depends on what you need to do with computer programming. While Python can be used for anything, it is not the most suitable language for everything. For instance, most game programmers use C++ or C# to create their programs, because these languages produce very fast executables, and speed is of utmost importance for games. People who create complex statistical models have their own computer languages too. Sometimes you need to write programs that interact with other programs which require the use of a specific language. And for some problems languages with a different philosophy to writing programs are better suitable.

All in all, while Python basically has no limitations as a programming language, for specific problems specific other languages might be better suitable. Still, for many people Python suffices to do everything they need to do for their studies and job. Moreover, Python is a great language to learn programming with, and once you understand and can use Python, you have a very strong basis to learn any other programming language. That is why I believe it is the language of choice to teach programming to beginners.

1.5 What does it mean "to think like a programmer?"

This book is not only meant to teach you to use Python, but, more importantly, to teach you how to think like a programmer, because thinking like a programmer is a necessity to understand what you can use computers for and how you should use them. But what does "thinking like a programmer" entail? I will answer this question by illustrating it with performing a specific task:

Suppose that you have a deck of cards, each card with a different number on it. You have to sort these cards from low to high, lowest card on top. Most people are able to do that. Also, most people, when you ask them how they do it, will look at you mystified, and answer: "Well,... I just sort them low to high... what do you mean with how do I do it?" Other people may say: "I first seek the highest card, and put it down. Then I seek the next highest card, and put it on top of the highest card. Etcetera." While this more or less explains how they sort cards, if you then ask them: "But how do you seek the highest card?," most of them, again, will look at you mystified.

The problem is that if you need to explain to a computer how to sort a deck of cards, you cannot assume that the computer can infer anything from vague statements, even if such statements would be completely clear to a human. You cannot tell the computer: "Seek the highest card," because even if the computer would understand English, it would ask you how it should seek the highest card. You will have to be very explicit about it. You have to say something like "Take the top card from the deck and hold it in you left hand. Then do the following until the deck runs out: take the top card from the deck in your right hand. If the value of the card in your right hand is higher than the value of the card in your left hand, put the left-hand card in the discard pile and put the right-hand card in your left hand. Otherwise, put the right-hand card in the discard pile. Once the deck has run out and your right hand is empty, the card in your left hand is the highest card."

Of course, a computer has no notion of left hand and right hand, and does not understand English. But a computer does understand computer language. Every computer language has a very precise syntax, and very precise semantics, which means that a computer program is an unambiguous explanation of how to perform a task. To have a computer perform a task, you have to use a computer language, bound by its syntax, to explain step-by-step how the task should be performed. Then, and only then, a computer can perform the task.

Since it is often very hard to think of all the steps needed to perform a task, you will have to divide the task into smaller subtasks, which you may have to divide again into even smaller subtasks, until the subtasks are so small that you can envision the steps needed to perform those subtasks. Then you can create implementations for each of the subtasks, and put them together to form a program for the task as a whole.

Thinking like a programmer means that you are able to approach a task from the perspective of programming a computer to perform that task, that you are able to recognize what a logical division into subtasks is, and that you can recognize when subtasks are sufficiently small so that you can implement them. This is a skill that most people can learn, but that requires a lot of practice and a thought process that is different from what most people are used to.

Using this book, by learning to create programs in Python, starting with small programs that gradually increase in complexity, you should also learn to use the thought processes that come naturally to a programmer.

1.6 The art of programming

Programming is an art form. A teacher of programming in many ways is comparable to a teacher of art.

Most people have had art classes in secondary school. An art teacher first teaches about art materials: pencils and paper, different colors of pencils, different hardness of pencils, erasers, inks, ink pens, paints, etcetera. The students use the knowledge acquired to create their first drawings. Then the art teacher makes them familiar with art techniques: mixing paints to get different colors, special kinds of paints that create special effects, combining techniques, how to use perspective, etcetera. Students get assignments like “draw a cat,” and the art teacher assesses their results both from the use of materials and the mastery of techniques, as well as from an artistic perspective of what makes a good likeness of a cat.

A teacher of programming has similar tasks. At first he teaches students about the core principles behind programming languages, basic statements that every programming language has, and how these can be used to create simple programs. Then he delves into more advanced techniques, by which students can construct more complex programs, and can incorporate advanced functionalities in easier ways. Students get assignments like “create a program that alphabetizes a text,” and the teacher assesses their results both from their use of programming techniques, as well as from the perspective of how well they manage to accomplish the task set.

From the art teacher’s perspective, for an assignment that asks students to draw a cat, a student who drew a circle with two triangles on top and two dots in the middle, arguably drew a cat but has no grasp of the use of materials. A student who hands in a beautiful

picture of a tree, may be a master of techniques but cannot use them to accomplish a task. And two students who hand in the exact same picture of a cat, clearly have been plagiarizing. Still, there isn't just one right "cat picture." There are many different cat pictures that are acceptable and that show that students are learning and becoming artists.

In the same way, a teacher of programming who gives an assignment, wants his students to creatively use the knowledge they acquired to construct their own version of a program that solves the task. Students who have not mastered the techniques, will be unable to solve the task, or will only be able to create a distant approximation of a program that solves the task. Students who did master the techniques, may still lack the aptitude of combining what they have learned in new and original ways to create a solution. And two students who hand in the exact same solution, clearly have been copying it and are trying to get away with plagiarism.

Programming is an art form, where you not only have to master the techniques that form the basis of your art, but also have to be able to apply these techniques creatively to problems. The main difference with producing programs and producing visual art, is that in visual art you can still debate about whether an image of a bulldog with pointy ears can be accepted as a picture of a cat, while in programming it is much easier to disqualify programs as solutions to a particular problem.

Moreover, everybody knows and realizes that you will never become an artist by just studying the materials. You will have to practice, use the materials, and develop your skills by applying them to many different tasks. In programming, it is exactly the same: you cannot learn how to program without writing many programs. Programming not only requires knowledge, but also skills that need to be developed in practice, and a form of creativity that allows you to expand your abilities to accomplish new tasks.

Naturally, there are few master artists whose work will feature in art galleries. But we can all draw pictures of cats, and for most people, that ability suffices for their daily needs. In the same vein, there is no need for any student to become a master programmer, as long as they can create straightforward programming solutions for the problems that they encounter in study and work. But be aware that next to mastery of basic techniques, creativity is always involved.

1.7 Start small, grow big

This is not the only Python book available, although most books that I have seen assume quite a bit of knowledge and past experience on the part of the student. Books that are aimed at absolute beginners are rare. Still, several alternatives for this book, even free alternatives, exist.

A problem that I have with most Python books aimed at beginners is that they attempt to make programming attractive by focusing from the start on applications that are immediately useful or entertaining. "Learn Python by Programming Games!" is a typical approach that I have encountered.

Such a setup is misleading. First of all, if you examine a book that teaches Python using games, you will discover that the games used are very simple word and number games, rather than the next *Halo*, *Civilization*, *Bejeweled*, or even *Flappy Bird*. This generally is not what the student would expect from a book on game programming. Moreover, you will

find that even those very simple "games" are of a complexity that is too high to allow a novice to learn programming. I understand that such a book tries to evoke enthusiasm amongst the students by framing its material in a way that seems attractive. However, that attraction is lost very quickly when students realize that the material is not what they expected, and at that point the topics become an obstruction to learning rather than a stimulant.

Regardless which way you look at it, like any other topic of study, learning how to program requires studying basic concepts before you can advance to more attractive and useful applications. The desire or need to learn programming is what should drive the student, not the erroneous expectation that they can hammer out a flashy game after a couple of hours of studying. That is why I designed this book to start small, with basic programming statements, and building knowledge up with a steady pace. The book does not remain stuck at the small stuff, however – if you progress through all the chapters, you will be an accomplished programmer when you finish it.

I try to insert exercises that can be entertaining to solve if such things appeal to the student. I have had students telling me that they really enjoy working on them. However, I have also seen students suffer and longing to do something else. Regardless, if you want to learn how to program, and do the exercises, the book will teach you everything that you need to program any application that you want – even entertaining games if you so desire.

1.8 Python 2 or Python 3?

Different versions of Python exist. At the moment of creating this book, the most popular versions are Python 2 and Python 3. Python 3 is, as can be expected, an update of Python 2. Python 2 programs are, unfortunately, not completely compatible with Python 3. Since a lot of Python 2 code is still in use, Python 2 is still an active language, and still being maintained.

The reason why Python 3 was created is to resolve a number of inconsistencies and idiosyncrasies in the Python 2 language. For people new to programming, this is a big plus, because there are less "weird" language elements they need to learn and understand if they choose Python 3 instead of Python 2.

To give an example, when you calculate 7/4 in Python 2, the answer is 1, and not 1.75 as you might expect. The reason is that both 7 and 4 are whole numbers ("integers"), and therefore the result of their division is a whole number. If you want to make sure that the result is 1.75, you must make at least one of the numbers involved a floating-point number. Therefore, 7.0/4 gives the result 1.75. This is how almost all computer languages do calculations. Naturally, for people who are not familiar with programming computers, this is counter-intuitive. Python 3 has resolved this issue, and automatically does the floating-point conversion when a floating-point result would be expected, i.e., in Python 3, 7/4 gives the result 1.75. Many Python 2 programs, however, are written with the assumption that integer-division rounds down, which means that, when you run them as Python 3 programs, they no longer give the desired results. Thus, Python 2 and Python 3 are not compatible.

Since Python 3 is more intuitive than Python 2, and since nowadays most Python programs and modules have been converted to Python 3, this book is written for Python 3. If you ever have to revert back to Python 2, it is not hard to make the change. An overview of the

differences between Python 2 and Python 3 is given in Appendix B (which is not a complete overview, but contains all the differences that I am aware of). If you are only using Python 3, you can ignore this appendix. However, considering how often I see the question "What exactly are the differences between Python 2 and Python 3?," and how hard it seems to be to find an answer to that question, I thought it prudent to add it.

1.9 Practice

Most chapters have small exercises sprinkled throughout the text. These exercises are there to enlighten a point or for you to do a quick check if you understood the material up to that moment. You should try to do these exercises immediately when you encounter them. Answers to these exercises are seldom provided, because if you understood the material, they should be really easy to do, while if you did not understand the material, you should either re-read the chapter until you do, or ask someone for assistance.

At the end of most chapters, a separate "Exercises" section is given, with one or more numbered exercises. You are supposed to do all of these exercises, and you should be able to do them independently (i.e., without help of other people and without looking up solutions from outside sources). Answers to these numbered exercises are provided in the back of the book, in Appendix F, and can also be downloaded from the website associated with this book (`http://www.spronck.net/pythonbook`). I wish to stress the following points:

- You should work on the numbered exercises until you have solved them. Do not dabble a bit and then look up the answer. Such an approach is utterly useless. There is no way that you are going to learn programming if you do not think about algorithms, write code, and test code. If you cannot solve an exercise even after you have worked on it for quite some time, it is better to ask for assistance than to just look up the answer. Being unable to solve an exercise means that there is something in the material that you have not grasped yet, and it is important that you identify what that is, and get to grips with it.
- You should do all of the exercises. The only way to learn programming is by practicing. You will have to write lots and lots of code before you have internalized the practice of programming. The few exercises that I place at the end of the chapters are not enough to accomplish that, but at least they are a start. If you cannot even bother to do all of those, you should not bother to try to learn programming.
- You should try to do the exercises independently. Working in teams on the exercises will allow one member of the team to learn, while the rest sits by and learns nothing. Students often say that they have a method of learning from assignments that involves working on them in small groups and discussing. That may be fine for analyzing texts or setting up an experiment, but in general does not work for coding. Watching someone write code teaches you very little about writing code. You have to write code by yourself.
- For none of the exercises you need information that was not discussed in the book up to that point. While there definitely may be easier ways to do some of the exercises if you would use Python constructs that I did not discuss yet at the time you get to the exercises, you do not need them. The purpose of the exercises is to practice with the discussed material. They are not meant to let you investigate future material. Even if

you are aware of different ways to solve an exercise, try to do it with only the material discussed. Once you have done that, if you want to return to an exercise later and solve it in a different way, that is, of course, fine.

- Once you have solved an exercise by yourself and have tested it extensively, then and only then you should compare it to the answer that I provided. You may find that it is different from yours. That does not mean that your answer is wrong! There are usually very many ways to solve a programming problem. Some might be "better" in some way than others. But there are many answers that are equally correct. Moreover, in this book it is important that you learn to solve a problem by coding, not that you learn to code the most efficient solution to a problem. Just being able to solve the problem suffices, making solutions more efficient is of much lesser importance. For instance, "being efficient" is less important than "being easy to understand" and "being easy to maintain."

For starters, here are two numbered exercises for the first chapter. Learn from them.

Exercises

Exercise 1.1 Get together with another person, and do the following exercise. Get four playing cards with different values. Shuffle them, and put them face-down on the table. One of you has to sort the cards, from low to high. This person is allowed to move the cards around, but is not allowed to look at their face sides. However, this person is allowed to point at two of the cards, after which the other one picks up those two cards, looks at their faces, and then puts them back and says which of the two is higher. Count how often such a comparison is made. Once the first person is satisfied that the cards are sorted, they are turned over to check if they are indeed sorted correctly.

In this exercise, the first person basically takes the role of a computer program, that follows instructions without actually understanding values. The second person takes the role of the computer processor, which can perform certain functions for the program, in this case, comparing numbers.

If you did not manage to sort the cards correctly, think about how you can accomplish this task under the given circumstances, and after that try it once more. If you did sort them correctly and needed more than six comparisons, think about how you can do it with six comparisons. If you managed to do it with six comparisons, think about if you can do it with less than six. If you did it with less than six, think about if your procedure is guaranteed to sort any collection of four cards.

Exercise 1.2 After doing the first exercise, together with your partner write down exact instructions, in plain English, on how to sort cards under the circumstances described above. Get a third person and ask this person to take your instructions and follow them, while one of you two takes the role of processor. Ask the third person to perform the steps as literally as possible, without trying to interpret meaning. This exercise is most illustrative if the third person has no idea what the exact function of the instructions is. Once the sorting procedure has finished, check if the result is correct.

Your textual description is comparable to a real program. If the third person is unable to follow the steps, you seem to have made a syntax error. If the person can follow the steps

but the end result is not as you want, you seem to have made a functional error. When programming computers, you will have to deal with both kinds of errors.

Note: writing such instructions is actually quite hard. Fortunately, writing similar instructions in a computer language is easier, as the syntax and semantics of the language are well-defined. English, as any other human language, is rather unsuitable to write unambiguous instructions.

Chapter 2

Using Python

As explained in the introduction, you will need to write and run Python code to learn anything from this book. That means that you need a computer on which Python is installed, and you need to know how to write and run Python programs. This chapter will explain to you how to get "up and running" with Python.

2.1 Getting Python

To run Python programs, you need a "Python interpreter." Fortunately, Python interpreters are freely available for almost every machine in existence. Visit `http://www.python.org` to download a Python interpreter for your computer. Make sure that you get a Python 3 interpreter. Install it. After the installation finishes, in principle you are ready to write and run Python programs.

It does not matter which operating system you use, whether it is Windows, Mac OS X, Linux, or something else: you write the same code for every machine. You can even take a program that you wrote on one machine and copy it to another, which may have a different operating system, and it will probably still run as intended (unless the program has some system-specific content, but I will get to that in a much later chapter).

Some Python courses use an online system in which students write Python code. That is a possible approach, but it has three disadvantages: (1) there are free systems which are very limited and therefore less useful; (2) there are paid systems which cost money and also have you deal with some peculiarities (as they run in a browser); and (3) at some point you will have to run Python on your own computer anyway, so why not start with it? That said, if you prefer to start with an online system and only move towards a locally installed version of Python in a later chapter, that is certainly possible.

2.2 Creating Python programs

Python programs are created in the form of files. By convention, the name of the file that contains a Python program has the extension `.py`.

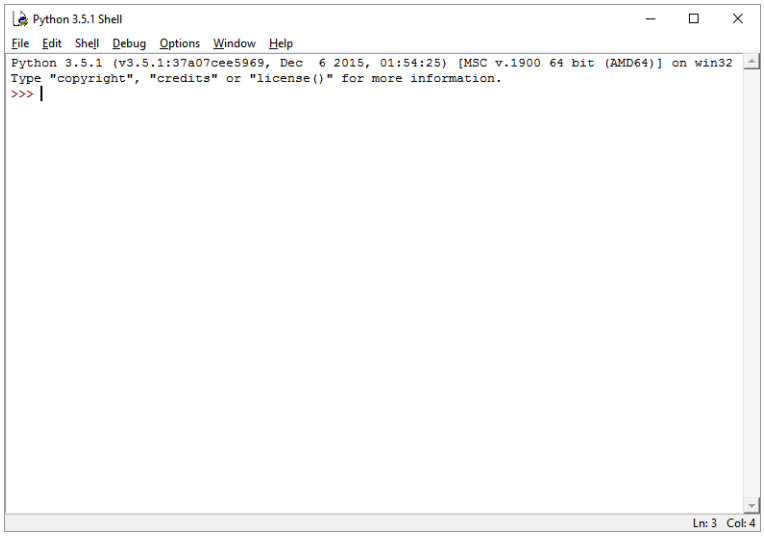


Figure 2.1: The IDLE environment.

Most Python installations (definitely those for Windows and Mac OS) also install an environment to create programs in, which is called IDLE. They usually also create some access point for the IDLE environment (for instance, and icon or link on the desktop or in a program menu). IDLE is a quite bare, but suitable environment to do your programming in.

When you start IDLE, you are in a so-called "Python shell" (see Figure 2.1) This is, as it were, an interactive Python program, in which you can type lines of Python code, which are run immediately. For instance, if you type `print(7/4)`, IDLE will show you the answer 1.75. In general, this is not how you wish to create and run code, but you can use the shell to quickly test the effect of Python statements.

To write Python programs with IDLE, you can create new Python files, or open existing Python files, using the "File" menu. IDLE then gives you a new window in which you can write code, edit code, and save code. You can even run the code immediately from this window, using the "Run" menu (there is a shortcut key to immediately run the program, usually `F5`). The program then actually runs in the shell, so that is where you supply input and can see the output. Make sure that you save your programs with a name that ends in `.py`.

There are user-friendlier ways to create Python programs. What you need is a text editor, preferably one that supports writing Python code specifically. Note that a text editor is different from a word processor; a text editor has no formatting options. You might see text getting formatted while typing in a text editor, with bold, italics, and colors, but this is so-called "syntax highlighting," which shows for a particular programming language how certain words that you type are understood by the program.

There are many text editors available that support writing Python code, for many different operating systems, most of them free. If you are unhappy with IDLE to write code, you can search for alternatives on the Internet. All editors have their own advantages and disadvantages, so what you prefer to use is up to you.

2.3 Running Python programs

Once you have created a Python program, you see the program name displayed in the folder where you saved it. You can try to run it in the same way that you run other programs (e.g., by double-clicking on it). For many Python programs, when you activate them in this way, you either see nothing happening, or a quick flash of a black window, after which nothing happens anymore. The reason is that Python programs run in a "command-line shell" of the operating system. If you are not a Linux user, this is probably not something that you are used to. Basically, what happens is that Python opens the command-line shell, runs the program, and when the program finishes, closes the shell, giving you the feeling that nothing has happened. But something did happen; you just did not notice it.

For the purpose of most of this book, you should simply run programs in the editor that you use, as I describe for IDLE above. You may open the command-line shell (which usually is a somewhat hidden option in the list of installed programs on your machine, falling under System Commands) and "manually" run Python programs from that shell, but there seldom is a need to do that.

2.4 Reference material

Besides this book, you might occasionally want to reference the Python manual. The easiest way to do that is using the Internet. Just search for "Python" with whatever you are interested in, and you will quickly see links that lead straight into the Python manuals (the Python manuals are stored at `http://docs.python.org`). You might also run into links that lead you to code that solves a problem for you directly. While that is great by the time you have to use Python for practical problems, it does not help you to learn. So my advice is that you avoid such links while trying to learn programming.

When you install Python, there usually is a manual installed in a Doc folder under the Python folder. You can use it if, for some reason, you are not connected to the Internet.

If you are interested in another book besides this one, I recommend the classic *Think Python: How to Think Like a Computer Scientist*, by Allen B. Downey. It is freely available from `http://greenteapress.com/wp/`. A version for Python 3 was released in 2016. The main differences with my book as far as content is concerned, is that my book has more exercises, is aimed more at people who are completely new to programming, takes more time for topics that I know are hard for students who have no real aptitude for programming, and covers a few important topics that Downey's book lacks, such as extensive file processing.

Besides this book and other books like it to learn Python from, there are several open video courses available. I do not believe that it is possible to learn programming mainly from watching a video. The only way to learn programming is by doing.

I have included a troubleshooting appendix (Appendix A) to explain the most common problems that you may encounter during the writing and running of programs.

Exercises

Exercise 2.1 Download Python and install it on the machine of your choice. Run IDLE. Create a file `hello.py`, in which you place the code of the *Hello World* program shown in Chapter 1 – it consists of one line of code, namely:

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

Run the program, and observe how the text "Hello, world!" is displayed in the IDLE shell.

Exercise 2.2 In the IDLE shell you can type commands on the IDLE prompt (>>>). Give the command **`print`**`(7/4)`. You will see that it prints the answer 1.75. Then give the command `7/4` (i.e., without **`print`**). Observe that it also prints the answer 1.75.

The reason is that the IDLE shell will always display the result of a command. The result of `7/4` is 1.75, and therefore it displays 1.75. The result of a **`print`** command is nothing, so the shell displays nothing – however, the **`print`** command causes the display of whatever is within the parentheses, which is the value resulting from dividing 7 by 4, which is 1.75. Therefore, in both cases you see 1.75, but one is the result of the use of the **`print`** command, while the other is the result of the shell showing you the evaluation of a calculation.

Now write a Python program that contains only the command `7/4`. Before you run it, think about what you expect to happen when you run it. Will the shell display 1.75? Will it display nothing? Or will you see an error?

Check if your expectation is correct.

Chapter 3

Expressions

Welcome to the first real programming chapter. In this chapter I discuss "expressions," which are straightforward calculations which you can also do with any simple calculator. It is a small start, but you are going to need such expressions for every chapter after this one.

3.1 Displaying results

When you write an expression in the Python shell, and you run it, the result of the expression is shown below it. For instance, if you type the following command in the shell and press `Enter`, you see the result 12.

```
5 + 7
```

However, as I showed in Exercise 2.2, a program that contains the statement `5 + 7` will not produce a result in the shell. Instead, you have to explicitly display everything that you want to see, even if it is on the last line of the program.

So, even though this chapter is about expressions, the first thing I need to explain is not an expression, but a function, that allows you to display results. The function that does that is **`print`**. I already showed the **`print`** function in Chapters 1 and 2.

The **`print`** function is used as follows: you write the word **`print`**, followed by an opening parenthesis, followed by whatever you want to display, followed by a closing parenthesis. For example (and I showed this one before):

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

If you run this code (by saving it into a Python file and running it in IDLE), you will see that it displays the text "Hello, world!" in the shell.

By the way, when referring to a function by name in a text, authors of texts about programming often put an opening and closing parenthesis after the name of the function, to indicate that it is a function name. From now on, I will follow this convention. Moreover, instead of referring to a "function," authors sometimes call it a "statement" or a "command."

However, these terms are usually used to refer to anything that Python can execute, not just functions. I.e., an expression can also be called a "command."

You can display multiple things with one **`print()`** function by putting everything that you want to display between the parentheses with commas in between. The **`print()`** function will then display all of the items, with one space in between each pair or them. For example:

```
print( "I", "own", "two", "apples", "and", "one", "banana" )
```

Note that the spaces in this statement are all superfluous. The statement:

```
print("I","own","two","apples","and","one","banana")
```

is equivalent to the previous one. You can add such spaces for readability. You can even put spaces between the word **`print`** and the opening parenthesis, but by convention, for functions (and **`print()`** is a function), the opening parenthesis is placed against the function name.

Note that you can not only use **`print()`** to display texts, but also to display numbers. You can even mix them up, as the following code shows.

```
print( "I", "own", 2, "apples", "and", 1, "banana" )
```

Exercise Display some texts of your liking using a Python program. But take note that if you want to display text strings, you have to enclose them in double quotes – or single quotes, those work too.

3.2 Data types

Before I can get to expressions, there is one more topic that requires some discussion, and that is data types. Specifically, there are three different data types that you need to be aware of at this time: strings, integers, and floats.

3.2.1 Strings

A string is a text, consisting of zero or more characters. In Python, a string is enclosed by either double quotes, or single quotes. In principle, it does not matter which of the two you use, i.e., `"orange"` is equivalent to `'orange'`. However, if you have a text which contains a single quote, if you want to avoid problems you will have to enclose it in double quotes, i.e., `"I can't stand it"` is a legal string, while `'I can't stand it'` is not. Vice versa for double quotes in a string, of course.

What if a string contains both double quotes and single quotes? You can solve that issue by putting a backslash (\) in front of the single or double quote that is part of the string to tell Python to treat that single or double quote as a character of the string rather than something that ends the string, i.e., `'I can\'t stand it'` is a legal string. You can see that when you try to print it:

```
print( 'I can\'t stand it' )
```

But what if I want to put an actual backslash in a string, and that backslash is, by chance, in front of a single or double quote? Well, I can do the same thing for a backslash, namely put a backslash in front of a backslash to make it a literal backslash, rather than a backslash that changes the interpretation of the character that comes after it. For an example, check out what the next bit of code displays (you can type it into the Python shell).

```
print( 'I can\\\'t stand it' )
```

If this all is a bit confusing, forget about these details for now, as I will come back to them in a later chapter. For now, just remember that a string is a text, enclosed by either single or double quotes. A string might be of any length, including zero characters long.

Be careful that you only use "straight" single or double quotes in your Python programs, and not "rounded" ones. Word processors are in the habit of changing your straight quotes into rounded quotes, and Python does not recognize those. Text editors will not do that, but should you, for some reason, copy code to and from a word processor, your quotes might get changed. Watch out for that.

3.2.2 Integers

Integers are whole numbers, which can be positive or negative (or zero). There is a certain maximum size that integers can become, which depends on the kind of computer and operating system you are running. For most purposes, however, you will not run into those boundaries. Python is not like those calculators with a 10-digit display that cannot use numbers higher than 10 billion.

There are different ways of writing integers that result in the same value. `1` is the same as `+1` (there are other ways than these to write the value `1`, but these follow in a later chapter). So both `print( 1 )` and `print( +1 )` produce the same outcome. This is different for strings, of course. The string `"1"` is not the same as the string `"+1"`.

When you use integers in Python, you cannot write them with "thousands separators" (commas in English) to make them more readable. I.e., the number one billion should be written as `1000000000` rather than `1,000,000,000`.

Check out the following code and think about what it will display when you run it. Then copy it to the Python shell and run it.

```
print( 1,000,000,000 )
```

Exercise If your prediction of what this code would do was not correct, find out why it produces this result.

3.2.3 Floats

Floats, or "floating-point numbers," are numbers with decimals. For instance, `3.14159265` is a float. Note that you have to use a period as the decimal separator. Many countries use

a comma as the decimal separator, but Python uses the convention of English-speaking countries and uses the period.

If there is an integer that for some reason you want to use as a float, you can do so by adding `.0` to it. I.e., `13` is an integer, while `13.0` is a float. Still, they represent the same value, and if you use Python to compare them (which I will get to in a short while), Python will tell you that they are the same value.

Just like with integers, there are certain maximum boundaries for floats, and there is also a maximum precision. You are unlikely to ever reach those maximum boundaries, as Python will switch over to scientific notation when the numbers get very big, but if you use Python to do very precise calculations, you might run into problems with precision. That is unlikely to happen for most applications, but if you are a physicist whose calculations involve huge numbers of particles on the molecular or quantum level, it is something to be aware of.

Note that due to the way that Python stores floats, certain numbers cannot be expressed exactly. For instance, the statement `print( (431 / 100) * 100 )` prints as answer 430.99999999999994, and not 431 as you might expect. If you know that the outcome of a floating-point calculation must be an integer, then you best make sure that you round the outcome to the nearest whole number. You can use the `round()` function for that, which will be explained in Chapter 5.

3.3 Expressions

Finally, I can get to the topic of this chapter, which is "expressions." An expression is a combination of one or more values (such as strings, integers, or floats) using operators, which result in a new value. In other words, you can think of expressions as calculations.

3.3.1 Basic calculations

Basic calculations combine two values with one operator in between them. Some straightforward operators are:

```
+   addition
-   subtraction
*   multiplication
/   division
//  integer division
**  power
%   modulo
```

Here are some examples:

```
print( 15+4 )
print( 15-4 )
print( 15*4 )
print( 15/4 )
print( 15//4 )
print( 15**4 )
print( 15%4 )
```

I assume you know what each of these operators entails, except perhaps the integer division and modulo operators.

The integer division (also called "floor division") is simply a division that rounds down to a whole number. If you involve floats in the calculation, the result will still be a float, but rounded down. If you only involve integers in the calculation, the result will be an integer.

The modulo operator (%) takes the remainder of a division. For example: If I divide 14 by 5, the result is 2.8, right? This means I can subtract 5 twice from 14, and still have a positive result, but if I subtract it a third time, the result will become negative. So, after subtracting 5 twice from 14 I have a remainder that is less than 5. This remainder is what the modulo operator produces.

In very simplistic terms: if I have 14 cookies which I have to divide over 5 children, each child gets 2 cookies. And I still have 4 cookies left, because there are more children than I have cookies at that point. Thus, dividing 14 by 5 as an integer division is 2 (cookies per child), while 14 modulo 5 is the remainder 4 (cookies I have left in my hand).

On a side note, I wish to point out that the code shown above consists of multiple lines. Each line is said to be a "statement," and it consists of one command that Python executes (in the code above, a `print()` function on every line). Most programming languages make it mandatory to end each statement with a special character, usually a semi-colon (;). Python does not require a semi-colon after each statement, but each statement must (in general) be on its own line. In principle, you are allowed to place multiple Python statements on one line, but then you should put semi-colons between the statements. However, it is Python practice and convention not to do that, as it makes code ugly, hard to read, and difficult to maintain. So, please stick to the convention and give each statement its own line.

3.3.2 More complex calculations

You are allowed to combine operators into bigger calculations, just as you can do on the more advanced calculators. To help you out, you are also allowed to used parentheses in your calculations, and you can even nest these parentheses. Python will process the operators in the order prescribed by mathematicians, often referred to as PEMDAS (Parentheses, Exponents, Multiplication and Division, Addition and Subtraction).

Check out the calculation below, and try to predict what it will result in before you copy it to the Python shell and run the code.

```
print( 5*2-3+4/2 )
```

There are a couple of things to note about this calculation.

First, the end result is a float (even though it has no decimals, or, if you will, only zero as a decimal). The reason is that a division is part of the calculation, and for Python that means that it should turn this into a floating-point calculation.

Second, just as explained above, spaces are ignored by Python, so the code above is the same as:

```
print( 5 * 2 - 3 + 4 / 2 )
```

It is even the same as:

```
print( 5*2 - 3+4    / 2 )
```

I have been in long discussions with people who keep arguing that the code above should result in 6.5 or 1.5, because *clearly* you have to calculate the $5 * 2$ and the $3 + 4$ before you do the subtraction and division. That is hogwash. It does not matter how close you place operands together, spaces are ignored. If you really want to calculate the $3 + 4$ first, you have to put it between parentheses. You can then still use spaces to improve readability, but they mean nothing to Python.

```
print( (5*2) - (3+4)/2 )
print( ((5*2)-(3+4)) / 2 )
```

Exercise Now it is time to write your first program. Write a program that displays the number of seconds in a week. You should, of course, not grab your calculator or smartphone to do the calculation and then just print the resulting number, but you should do the calculation in Python code. Since this program needs only one line of code, you could just write it in the Python shell, though you are encouraged to create a program file and use that.

3.3.3 String expressions

Some of the operators given above can also be used for strings, though not all of them.

In particular, you can use the addition operator ($+$) to concatenate two strings, and you can use the multiplication operator ($*$) with a number and a string to create a string that contains a repetition of the original string. Check it out:

```
print( "hello"+"world" )
print( 3*"hello" )
print( "goodbye"*3 )
```

You cannot add a number to a string, or multiply two strings. Such use of the operators is undefined, and will give error messages. None of the other operators listed for numbers will work on strings either.

3.3.4 Type casting

Sometimes you need to change the data type of a value into a different data type. You can do that using type casting functions.

I will discuss functions in a lot more detail in a later chapter, but for now you just need to know that a function has a name, and may have parameters (values) between parentheses after the name. It will do something with the parameters, and then may give back a result. For instance, the `print()` function displays the parameter values that are given to it between the parentheses, and gives nothing in return.

The type casting functions take the parameter value between the parentheses and give back a value that is the (almost) the same as the parameter value, but of a different data type. The three main type casting functions are the following:

- **int()** will return the value between the parentheses as an integer (rounding down if necessary)
- **float()** will return the value between the parentheses as a float (adding .0 if necessary)
- **str()** will return the value between the parentheses as a string

See the difference between the following two lines of code:

```
print( 15/4 )
print( int( 15/4 ) )
```

Or the following two lines of code:

```
print( 15+4 )
print( float( 15+4 ) )
```

I stated that you cannot use the addition operator to concatenate a number to a string. However, if you need to do something like that, you can work around the issue by using string type casting:

```
print( "I own " + str( 15 ) + " apples." )
```

3.4 Style

You might have noticed that in my example code I use white spaces a lot. For instance, for parentheses attached to functions, I almost always have a white space after the opening parenthesis and before the closing parenthesis. In calculations, I often have white spaces around operators if that makes the calculations better readable. I also often insert empty lines in my code to make it more readable, and consistently use four spaces as indentations.

Most of these things are just "style." The white spaces next to the parentheses and around operators are not necessary, Python understands the code just as well when they are gone. These four statements are all equivalent:

```
# All equivalent statements
print( 2 + 3 )
print(2+3)
print( 2+3)
print       (       2       +       3       )
```

Attaching the opening parenthesis to a function is something that almost every programmer does, but for the rest, styles of placing white spaces differ between programmers (my style of placing a space before the closing parenthesis is rare). You can choose your own

style in this respect, you do not need to follow mine. But I recommend that you use your chosen style consistently, which will make your code more readable even for programmers who use a different style.

Note that in the code above there is a hash mark (#) on the first line, with a text after that which explains some details of the code. The line with the hash mark is a comment line: whenever you put a hash mark in your code (except when it is within a string, of course), everything to the right of the hash mark for the remainder of the line is commentary, which Python ignores. You can use comments to clarify your code, if such clarification is needed. More details on providing comments to code I will give in a later chapter.

What you learned

In this chapter, you learned about:

- Using the **`print()`** function to display results
- Data types string, integer, and float
- Calculations
- Basic string expressions
- Type casting between strings, integers, and floats, using **`str()`**, **`int()`**, and **`float()`**

Exercises

Exercise 3.1 The cover price of a book is $24.95, but bookstores get a 40 percent discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. Calculate the total wholesale costs for 60 copies.

Exercise 3.2 Can you identify and explain the errors in the following lines of code? Correct them.

exercise0302.py

```
print( "A message" ).
print( "A message' )
print( 'A messagef"' )
```

Exercise 3.3 When something is wrong with your code, Python will raise errors. Often these will be "syntax errors" that signal that something is wrong with the form of your code (e.g., the code in the previous exercise raised a `SyntaxError`). There are also "runtime errors," which signal that your code was in itself formally correct, but that something went wrong during the code's execution. A good example is the `ZeroDivisionError`, which indicates that you tried to divide a number by zero (which, as you may know, is not allowed). Try to make Python raise such a `ZeroDivisionError`.

Exercise 3.4 Here is another illustrative example of a runtime error. Run the follow code and study the error that it generates. Can you locate the problem?

exercise0304.py

```
print( ((2*3)/4 + (5-6/7)*8 )
print( ((12*13)/14 + (15-16)/17)*18 )
```

Exercise 3.5 You look at the clock and see that it is currently 14.00h. You set an alarm to go off 535 hours later. At what time will the alarm go off? Write a program that prints the answer. Hint: for the best solution, you will need the modulo operator.

Chapter 4

Variables

When working with program code, very often you are designing a procedure (or "algorithm") that solves a problem in a general way. For instance, in the previous chapter one of the exercises had you calculate the wholesale price for a stack of books, for a given book price and a given number of books. The code you wrote did not solve this problem for a general case, but only for the specific case of 60 books costing 24.95 per book. If you want to write code that solves problems in a more general way, you need to use variables that store values.

4.1 Variables and values

A variable is a labeled place in the computer memory that you can use to store a value in. The label you can choose yourself, and is usually called the "variable name."

To create a variable (i.e., choose the variable name), you must "assign" it a value. The assign-operator is the equals (=) symbol. To the left of it you put the variable name, and to the right of it you put the value that you want to store in the variable. This is best illustrated with an example:

```
x = 5
print( x )
```

In the code block above, two things happen. First, I create a variable with the name `x` and give it a value, in this case 5. This is called an "assignment." I then display the contents of the variable `x`, using **`print`**`()`. Note that **`print`**`( x )` does not display the letter `x`, but actually displays the value that was assigned to `x`.

The variable `x` behaves pretty much like a box on which you write an `x` with a thick, black marker to be able to find it later. You can put something in the box, and then look into the box to see what you put in (though only one thing at a time will fit in the box). You can refer to the contents of the box by using the name written on the box. The term "variable" means the variable name, i.e., the letter `x` on the box. The term "value" means the value that is stored in the variable, i.e., the contents of the box.

To the right of the assign operator you can place anything that results in a value. Therefore, it does not need to be a single number. It can be, for instance, a calculation, a string, or a call to a function that results in a value (such as the **int()** function).

Exercise In the previous chapter you wrote a calculation that determines the number of seconds in a week. Copy this calculation into a program, and assign it to a variable. Then add a statement to print the contents of the variable.

When you assign a value to a variable name in your program, the first time you do that for a specific variable name, it creates the variable. If later in the program you assign another value to the same variable name, it "overwrites" the previous value. In the box metaphor: you empty the box and put something else in it. A variable always holds the value that was last assigned to it.

```
x = 5
print( x )
x = 7 * 9 + 13   # overwrite the previous value of x
print( x )
x = "A nod's as good as a wink to a blind bat."
print( x )
x = int( 15 / 4 ) - 27
print( x )
```

Once a variable is created (and thus has a value), you can use it in your code where you otherwise would use values. You can, for instance, use it in calculations.

```
x = 2
y = 3
print( "x =", x )
print( "y =", y )
print( "x * y =", x * y )
print( "x + y =", x + y )
```

You may copy the contents from one variable to another, using the assignment operator.

```
x = 2
y = 3
print( "x =", x, "and y =", y )

# Swap the values of x and y using z as intermediary storage.
z = x
x = y
y = z
print( "x =", x, "and y =", y )
```

When you assign something to a variable, you might even use the variable itself on the right-hand side of the assignment operator, provided it was created earlier. The right-hand side of an assignment is always evaluated completely before the actual assignment takes place.

```
x = 2
print( x )
x = x + 3
print( x )
```

Note that a variable must be created before you can use it! Running the following code will result in an error, because `days_in_a_year` has not (yet) been created before I use it on the first line:

```
print( days_in_a_year )
days_in_a_year = 365
```

4.2 Variable names

So far, I have only used variables called `x`, `y`, and `z` (and one erroneous `days_in_a_year`). However, you are free to choose the names of your variables as you like them, provided that you follow a few simple rules, namely:

- A variable name must consist of only letters, digits, and/or underscores (_)
- A variable name must start with a letter or an underscore
- A variable name should not be a reserved word

"Reserved words" (or "keywords") are:

```
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
```

You can use capitals and lower case letters in variable names, but you should realize that variable names are case sensitive, i.e., the variable `world` is not the same as the variable `World`.

4.2.1 Conventions

Programmers follow many conventions when choosing variable names. The major ones are the following:

- Programmers *never* choose variable names that are also the names of functions (whether they are functions provided by Python or functions they wrote themselves). Doing so will cause the corresponding function to be no longer accessible by the code, and may then lead to rather eccentric errors.
- Programmers try to choose variable names that are in some way meaningful to the code. For instance, a variable that stores the number of seconds in a week, might have the name `secs_per_week`, but not the name `i_hate_my_job`. It would be even worse to name a variable that contains the numbers of seconds in a week `secs_per_month`.
- An exception to choosing meaningful variable names is choosing names for "throw-away" variables, i.e., variables that you only use in a very small section of the code and that are no longer needed afterwards, and that have no good meaning by themselves. Programmers usually choose a single-letter name for such variables. For instance, if a variable is needed to quickly count to 100, after which it is not needed anymore, programmers often choose the letter `i` or `j` for such a variable.
- To avoid confusion with capitals and lower case letters, programmers tend to use only lower case letters in variable names.
- If a variable name is chosen that consists of multiple words, programmers put one underscore between each of the words.
- Programmers never choose variable names that start with an underscore. Such variable names are considered reserved for the authors of the Python interpreter.

You should try to stick to these conventions for your own code. In particular the convention of choosing meaningful variable names is important to follow, because meaningful variable names make code readable and maintainable. Look, for instance, at the following code:

```
a = 3.14159265
b = 7.5
c = 8.25
d = a * b * b * c / 3
print( d )
```

Do you understand what this code does? You probably see that a seems to be an approximation of π, but what is `d` supposed to be?

I can tell you that this code calculates the volume of a cone. You probably would not have guessed that, but that is what it does. Now I ask you to change the code to calculate the volume of a cone that is 4 meters high. What change will you make? If height is part of the calculation, it is probably `b` or `c`. But which is it? Maybe if you know a bit of maths and you

look at the calculation of `d`, you realize that `b` is squared in this calculation, which seems to refer to the base of the cone, which is a circle. So it is probably `c`. But you cannot be sure.

Now look at the following, equivalent code:

```
pi = 3.14159265
radius = 7.5
height = 8.25
volume_of_cone = pi * radius * radius * height / 3
print( volume_of_cone )
```

This is much more readable, right? If I asked you to look at this code and tell me what it does, and make the requested change, I don't expect you to hesitate in answering.

Such code with meaningful variable names tends to become "self-documenting"; you do not need to add any comments to make the user understand what it does and how it does it. Still, in the code above a line of comment that says:
`# calculation of volume of a cone with radius 7.5 and height 8.25`
would not be misplaced.

4.2.2 Practicing with variable names

Exercise In the code block below, the value 1 is assigned to a number of (potential) variable names. Some of these are legal, others are not. Identify the illegal variable names, and explain why they are illegal.

```
classification = 1   # 1
Classification = 1   # 2
cl@ssification = 1   # 3
class1f1cat10n = 1   # 4
1classification = 1  # 5
_classification = 1  # 6
class = 1            # 7
Class = 1            # 8
```

Answer The third, fifth, and seventh assignments are illegal. The third because it does not consist of only letters, digits, and underscores. The fifth because it starts with a digit. The seventh because it is a reserved word (fortunately, syntax hightlighting makes it stand out). While the others are legal, according to convention the sixth should be avoided because it starts with an underscore, and the second and eighth too, as they contain capitals. The eighth is the worst in this respect, as it also looks like a reserved word.

4.2.3 Constants

Many programming languages offer the ability to create "constants," which are values assigned to a variable which can no longer be changed after the value has been first assigned. It is convention in most such languages that the name of a constant is written in all capitals.

Constants can be useful to make code more readable. For instance, to calculate the total of a bill of 24.95 EUR with a 15% service charge, you can use:

```
total = 24.95
final_total = int( 100 * total * 1.15 ) / 100
print( final_total )
```

However, it is more readable to write:

```
SERVICE_CHARGE = 1.15
CENTS = 100

total = 24.95
final_total = int( CENTS * total * SERVICE_CHARGE ) / CENTS
print( final_total )
```

Not only is it more readable, but it also makes the code easier to change should the service charge be calculated differently in the future. Especially if the service charge occurs in the code multiple times, if it is defined just once as a constant at the top of the code, it can be easily found and changed. When they are numerical, special values such as the service charge are often called "magic numbers," i.e., their particular value has a special meaning, which is unclear if you just see the number, so you are better off using a meaningful name instead of the number.

While constants are very useful for coding purposes, Python does not support them (which is a pity), i.e., in the code above `SERVICE_CHARGE` is a regular variable and can be changed anywhere in the code. Still, it is convention that any variable that is written in all capitals is supposed to be a constant and should not be changed in the code, after it got its initial value at the top of the code.

You are encouraged to use such "all capitals variable names" whenever magic numbers occur in your code.

4.3 Debugging variables

Typically, when things go wrong in a program, the reason is that variables are holding values that you did not expect them to have when writing the code. A good way of "debugging" your code (i.e., finding out where problems are and fixing them) is printing the variable names and values in appropriate places. For instance, the following code gives an error when you run it.

listing0401.py

```
nr1 = 5
nr2 = 4
nr3 = 5
print( nr3 / (nr1 % nr2) )
nr1 = nr1 + 1
print( nr3 / (nr1 % nr2) )
nr1 = nr1 + 1
```

```
print( nr3 / (nr1 % nr2) )
nr1 = nr1 + 1
print( nr3 / (nr1 % nr2) )
```

In this case you might see what the problem is, but suppose you do not, how are you going to find out what is wrong? You see that the error occurs on line 10 of the code (the last line), which means that everything is still running okay at line 9. If you insert a new line of code between line 9 and line 10 that prints the values of `nr1`, `nr2`, `nr3` and perhaps also `nr1%nr2`, you probably quickly determine what the problem is. Adding print statements does not actually change anything about the variables, so print statements are safe to add. A nice fix for the problem (i.e., something else than just removing the offending line) will be introduced in a later chapter.

Exercise Add the line suggested above, printing the variables right before the error occurs, to the erroneous code.

4.4 Soft typing

All variables have a data type. In many programming languages, the type of a variable is given when the variable is first created. For instance, in C++, when you create a variable you declare the type in front of it, like so:

```
int secs_per_week = 7 * 24 * 60 * 60;
```

This is called "hard typing," and it has the advantage that if you create a variable that you intend to be of a certain type, but then assign it a value of a different type, the program can announce that you made a mistake. This avoids some annoying and confusing errors that might occur.

In Python, you do not "declare" the type of a variable, but a variable still has a type, namely the type of the value that was assigned to it. This entails that if you assign a new value to a variable, its type might change. This is called "soft typing." (Note: I am personally of the opinion that Python would be an even better language to teach people programming if it had hard typing instead of soft typing, and I am not alone in that opinion, but Guido van Rossum, the original creator of Python, disagrees.)

The types that you have seen until now are integer, float, and string. You can use the function **`type()`** to see what the type of a variable is.

```
a = 3
print( type( a ) )
a = 3.0
print( type( a ) )
a = "3.0"
print( type( a ) )
```

Since variables have a type, the effect of operators might change depending on the types of the variables involved. For instance, in the following code, the addition operator (+) is used twice, but its effect changes due to the types of the variables involved.

```
a = 1
b = 4
c = "1"
d = "4"
print( a + b )
print( c + d )
```

Since `a` and `b` are both numbers, for `a + b` the addition operator is a numerical addition. Since `c` and `d` are both strings, the addition operator for `c + d` is the string concatenation.

Exercise In the code above, what would happen if you try to print `a + c`? If you do not know, try it.

Exercise What does the code given below display? First think about it, then run the code, and make sure that you understand what happens.

```
name = "John Cleese"
print( "name" )
```

Exercise Change the code above so that it displays the name of a famous member of Monty Python.

4.5 Shorthand operators

Using the operators you have learned about above, you can change the variables in your code as many times as you want. You can assign new values to existing variables. Very often, you want to make changes to existing variables. For instance, it is common in code that you want to add 1 to a number (you will find out why that is in a later chapter). Since this occurs fairly often, Python offers some shorthand notation to deal with changes to variables.

The following code:

```
number_of_bananas = 100
number_of_bananas = number_of_bananas + 1
print( number_of_bananas )
```

is equivalent to:

```
number_of_bananas = 100
number_of_bananas += 1
print( number_of_bananas )
```

The difference is in the second line. If you want to add something to a variable, you can write += as the assignment operator and to the right-hand side of the += the thing that you want to add to the variable. This saves you the trouble of repeating the variable name at

the right-hand side, and tends to make your code more readable (because programmers expect you to code "adding something to an existing variable" with the += operator).

Similar to the += operator, you can use -= to subtract something from a variable, *= to multiply a variable by something, /= to divide a variable by something, **= to raise a variable to a power, and %= to turn a variable into itself modulo the right-hand side. Most of these are uncommon, except for the +=, which is used a lot, and the -=, which is used occasionally.

Exercise What will the code given below display? Run it to see if you are correct.

listing0402.py

```
number_of_bananas = 100
number_of_bananas += 12
number_of_bananas -= 13
number_of_bananas *= 19
number_of_bananas /= number_of_bananas
print( number_of_bananas )
```

4.6 Comments

Since the code that you have to write has now increased to more than five lines or so, it has become sufficiently complex to warrant discussing the use of comments. Comments are texts in code that Python ignores, but that explain parts of the code. Comments are not only useful to other people which might need to use or change your code, but also to yourself, as you may need to change your own code some time after you wrote it and you might not remember exactly what you did.

There are two main ways to include comments in Python code. The first is to use a hash mark (#), which turns everything to the right of the hash mark on the line into commentary (of course, this is only the case if the hash mark is not part of a string). The second is to use triple double-quotes or triple single-quotes to indicate the start and end of some commentary, which may be spread over multiple lines. In this case, the starting triple quotes should always be at the start of a line, and you cannot use this way of commenting in an indented code block. The reason is that you are basically placing a multi-line string in your code (more on this in Chapter 10).

Learn more about comments by studying the code below.

listing0403.py

```
# comment: insert your code here.
# BTW: Have you noticed that everything right of the hash mark
print( "Something..." ) # is ignored by your python interpreter?
print( "and something else.." ) # Use this to comment your code!
"""Another way of commenting on your code is via triple quotes
-- these can be distributed over multiple """ # lines
'''which can also be done with single quotes''' # but be careful
# with there being quotes IN your comments when you use this
# multi-line method
print( "Done." )
```

What you learned

In this chapter, you learned about:

- What variables are
- Assigning a value to a variable
- Legal names for variables
- Good names for variables
- Soft typing
- Debugging code in which variables might have unexpected values
- Shorthand statements for changing variable values
- Code commentary

Exercises

Exercise 4.1 Define three variables `var1`, `var2` and `var3`. Calculate the average of these variables and assign it to `average`. Print the average. Add three comments.

Exercise 4.2 Write code that can compute the surface of circle, using the variables `radius` and `pi` = 3.14159. The formula, in case you do not know, is radius times radius times pi. Print the outcome of your program as follows: "The surface area of a circle with radius ... is ..."

Exercise 4.3 Write code that classifies a given amount of money (which you store in a variable named `amount`), specified in cents, as greater monetary units. Your code lists the monetary equivalent in dollars (100 ct), quarters (25 ct), dimes (10 ct), nickels (5 ct), and pennies (1 ct). Your program should report the maximum number of dollars that fit in the amount, then the maximum number of quarters that fit in the remainder after you subtract the dollars, then the maximum number of dimes that fit in the remainder after you subtract the dollars and quarters, and so on for nickels and pennies. The result is that you express the amount as the minimum number of coins needed.

Exercise 4.4 Can you think of a way to swap the values of two variables that does not need a third variable as a temporary storage? In the code block below, try to implement the swapping of the values of `a` and `b` without using a third variable. To help you out, the first step to do this is already given. You just need to add two more lines of code.

exercise0404.py

```
a = 17
b = 23
print( "a =", a, "and b =", b )
a += b
# add two more lines of code here to cause swapping of a and b
print( "a =", a, "and b =", b )
```

Chapter 5

Simple Functions

Up to this point, I have already introduced some basic "functions," such as **`print()`** and **`int()`**. In this chapter these functions will be discussed a bit more in-depth, and a few other functions will be introduced, which will be helpful in the coming chapters. In Chapter 8, I will discuss how you can create your own functions.

5.1 Elements of a function

A function is a block of reusable code that performs some action. To get a function to do its job, you "call" it, with some appropriate parameters if the function requires them. The idea is that you do not need to have knowledge about how a function performs its action. You only need to know three things:

- The name of the function
- The parameters it needs (if any)
- The return value of the function (if any)

These will now be discussed in turn.

5.1.1 Function name

Each function has a name. Like a variable name, a function name may consist of letters, digits, and underscores, and cannot start with a digit. Almost all standard Python functions consist only of lower case letters. Usually a function name expresses concisely what the function does.

When referring to a function, it is convention to use the name, and put an opening and closing parenthesis after the name, as functions are always called in code with such parentheses.

5.1.2 Parameters

Some functions are called with parameters ("arguments"), which may or may not be mandatory. The parameters are placed between the parentheses that follow the function name. If there are multiple parameters, you place commas between them.

The parameters are the values that the user supplies to the function to work with. For instance, the **int()** function must be called with one parameter, which is the value that the function will try make an integer representation of. The **print()** function may be called with any number of parameters (even zero), which it will display, after which it will go to a new line.

In general, a function cannot change parameters. For instance, look at the following code:

```
x = 1.56
print( int( x ) )
print( x )
```

As you can see when you run this code, the **int()** function has not changed the actual value of x; it only told the **print()** function what the integer value of x is. The reason is that, in general, parameters are "passed by value." This means that the function does not get access to the actual parameters, but it gets copies of the values of the parameters. I say "in general" because not all data types are "passed by value," but the ones I have discussed until now are. It will be a while before you get to a chapter that introduces data types that can be changed by functions when they are passed as parameters, and I will make abundantly clear how that works when it comes up.

If a function gets multiple parameters, their order matters. For instance, the function **pow()** gets two parameters, and raises the first to the power of the second.

```
base = 2
exponent = 3
print( pow( base, exponent ) )
```

The names of the variables that are used as parameters do not matter, the first is raised to the power of the second. So the following example will give a different outcome than the first, as the same variables are given to the function in a different (rather confusing) order.

```
base = 2
exponent = 3
print( pow( exponent, base ) ) # confusing use of variables
```

What happens if you try to call a function with parameters that it cannot work with? For instance, what happens if I call the **int()** function with a string that does not contain an integer value, or the **pow()** function with strings instead of numbers? In general, this will lead to runtime errors in your code. For instance, both lines of the code below give a runtime error.

```
x = pow( 3, "2" )
y = int( "two-and-a-half" )
```

5.1.3 Return value

A function may or may not "return" a value. If a function returns a value, that value can be used in your code. For instance, the function **int()** returns an integer representation of the parameter it gets. You can place this return value in a variable, using an assignment, or use it in a different manner, for instance immediately print it. You can even not do anything with it, though there is little reason to call the function in that case.

```
x = 2.1
y = '3'
z = int( x )
print( z )
print( int( y ) )
```

As you can see from the example above, you can even use function calls as parameters for a function; e.g., the second call to the **print()** function in the example gets as parameter a call to the function **int()**. In this example, the call to the **int()** function is executed before the **print()** function is called, as Python first calculates the values for all the parameters before it makes a function call. So the return value of **int()** is a parameter for **print()**.

Not all functions return a value. For instance, the **print()** function does not. If you are not careful, this may lead to strange behavior of your program. For instance, examine and run the following code:

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

You can see that this code prints two lines, the first containing the text "Hello, world!" and the second containing the word "None." What is that "None" doing there? To find that out, let's examine how Python evaluates this statement.

When Python first encounters this statement, it must evaluate **print(** <something>). Since <something> is an argument, it starts by evaluating that. <something> is actually **print(** <something_else>). Since <something_else> is an argument, it now evaluates that. <something_else> is the string "Hello, world!". This is not something that needs to be evaluated, so it calls **print()** with this string as argument, and "captures" the return value of **print()** because it needs it as the evaluation of <something>.

Here is the crux: **print()** has no return value, so there is nothing that Python can use for <something>. For situations such as this, Python has a special value called **None**. So the first **print()** gets called with **None** as argument, and this leads to Python displaying the word "None."

None is a special value that indicates "no value at all." If you try to print such a value, Python prints the word "None," but is not actually printing a string that is "None". It only indicates that there was nothing to print. **None** is different from, for instance, an empty string (""). An empty string is still a value, namely a string of length zero. **None** is no string at all, no integer, no float, nothing. So be careful when trying to use a function call as a parameter; if the function does not actually return a value, weird things may happen.

5.1.4 A function is a black box

Let me stress once more that you may consider a function a "black box": you do not need to know how the function works or how it is implemented. The name, parameters, and return value are all you need to know. The function might, internally, create variables and do calculations, but they do not have an effect on the rest of your code.

...At least, if the function is implemented well. A function that has no effect on your code is called a "pure function," and the functions that I discuss here are all "pure functions." However, sometimes functions are designed that actually do have an effect outside the function, specifically, that the user may provide parameters to that undergo a change. That may be fine, if it is intentional and well-documented. Such functions are called "modifiers." Modifiers will come up in later chapters.

For now, you can just assume that any function that you use, has no effect on the rest of your code. So calling a function is safe.

5.2 Some basic functions

At this point, I introduce some basic functions that you can use in your Python programs.

5.2.1 Type casting

I already introduced the type casting functions, but now I have explained more details of functions, I can give a complete description.

- **`float`**`()` has one parameter and returns a floating-point representation of the value of that parameter. If the parameter holds an integer, it returns the same value as a float (if you print it, you will see `.0` added). If the parameter holds a float, it returns the same value. If the parameter holds a string which can be interpreted as an integer or a float, it returns that interpretation as a float; otherwise it will give a runtime error.
- **`int`**`()` has one parameter and returns an integer representation of the value of that parameter. If the parameter holds an integer, it returns the same integer. If the parameter holds a float, it returns the integer part of the float, i.e., the float value rounded down. If the parameter holds a string, and the string contains only digits, optionally with a preceding minus-sign, it returns the integer represented by those digits; otherwise it will give a runtime error.
- **`str`**`()` has one parameter and returns a string representation of the value of that parameter.

Exercise What will happen if you run the following code? If you do not know, try it and find out.

```
print( 10 * int( "100,000,000" ) )
```

Exercise The code above gives a runtime error. Fix it by removing a few characters.

5.2.2 Calculations

Basic Python functions also have limited support for calculations.

- **abs()** has one numerical parameter (an integer or a float). If the value is positive, it will return the value. If the value is negative, it will return the value multiplied by -1.
- **max()** has two or more numerical parameters, and returns the largest.
- **min()** has two or more numerical parameters, and returns the smallest.
- **pow()** has two numerical parameters, and returns the first to the power of the second. Optionally, it has a third numerical parameter. If that third parameter is supplied, it will return the value modulo that third parameter.
- **round()** has a numerical parameter and rounds it, mathematically, to a whole number. It has an optional second parameter. The second parameter must be an integer, and if it is provided, the function will round the first parameter to the number of decimals specified by the second parameter.

Exercise Examine the code below and try to determine what it displays. Then run the code and see if you are correct.

listing0501.py

```
x = -2
y = 3
z = 1.27

print( abs( x ) )
print( max( x, y, z ) )
print( min( x, y, z ) )
print( pow( x, y ) )
print( round( z, 1 ) )
```

5.2.3 len()

len() is a basic function that gets one parameter, and it returns the length of that parameter. For now, the only data type which you will use **len()** for is the string. **len()** returns the length of the string, i.e., the number of characters.

Exercise What does the code below print? Run it and check if you are correct.

```
print( len( 'can' ) )
print( len( 'cannot' ) )
print( len( "" ) )  # "" is an empty string
```

Exercise And what about the code below? Think carefully, then check the result.

```
print( len( 'can\'t' ) )
```

5.2.4 input()

You will often want the user of a program to supply some data. You can ask the user to supply a string value by using the **input()** function. The function has one parameter, which is a string. This string is the so-called "prompt." When **input()** is called, the prompt is displayed on the screen and the user gets to enter something. The user may type anything they want, including nothing, and then press Enter to stop entering input. The return value of the function is a string which contains what the user entered, excluding that final press of the Enter key.

It depends on the environment in which you use Python how exactly the user gets asked to enter input. Sometimes a box is displayed in which you can type something. If you run Python from the command prompt, it is done as a command line. In different editors, it is done differently; for instance, there are editors that show a pop-up box.

Here is an example:

```
text = input( "Please enter some text: " )
print( "You entered:", text )
```

Be aware that **input()** always returns a string. Check the following code:

```
number = input( "Please enter a number: " )
print( "Your number squared is", number * number )
```

Regardless of what you entered, this code gives a runtime error, because since the **input()** function returns a string, number is a string, and you are not allowed to multiply two strings. You may resolve this by using a type casting function to turn the string result of **input()** into a numerical value, for instance:

```
number = input( "Please enter a number: " )
number = float( number )
print( "Your number squared is", number * number )
```

As long as the user enters a value that can be turned into a number, this code runs as intended. However, if the user enters something that cannot be turned into a number, you again get a runtime error. There are ways to resolve this issue, but I have not discussed the means to do that yet, and it will take a while before I do that. However, below I will introduce a way for you to ask the user for numbers without the code crashing if the user is trying to be a wise-ass and enters something else.

Exercise Write some code that asks the user for two numbers, then shows the result when you add them, and when you multiply them.

5.2.5 print()

The function **print()** takes zero or more parameters, displays them (if there are multiple, with a separating space in between each pair of them), and then "goes to the next line" (i.e., if you use two **print()** statements, the second one will display its parameters below what the first one displays).

If **print**() is called without parameters, the function simply will "go to the next line." This way, you can display empty lines.

You can supply **print**() with anything as a parameter, and it will do its best to print it. For now, you will only print the basic data types.

print() can get two special parameters, called sep and end.

sep indicates what should be printed between each of the parameters, and by default is a space. You can use sep to turn the separating space into anything else, including an empty string.

end indicates what **print**() should put after all the parameters have been displayed, and by default is a "newline." You can use end to change what **print**() does after displaying the parameters, for instance, you can ensure that **print**() does not "go to the next line."

To use sep and end, you include parameters sep=<string> and/or end=<string> (note: when in a code description you see something between < and >, that usually means that you are not supposed to type that literally, but that you have to replace it with something of the type listed, e.g., <string> means that you have to type a string in that place). For example:

```
print( "X", "X", "X", sep="x" )
print( "X", end="" )
print( "Y", end="" )
print( "Z" )
```

When you run this code, you see two lines on the output. The first contains "XxXxX," because the first line of code said that three times the letter "X" should be displayed, with a lower case "x" as separator between each pair. The second line contains "XYZ," because even though there are three **print**() statements which together created this line, the code says that Python should not go to the next line at the end of the first two.

5.2.6 format()

format() represents a rather complex functionality that is employed in a particular way. It allows you to create a formatted string, i.e., a string in which certain values appear in a specific format. To give an example, suppose I want to display a calculated float:

```
print( 7/11 )
```

Now I ask you to display that float with only three decimals. Until now, you would use the **round**() function (introduced above), or something like:

```
print( round( 7/11, 3 ) )
```

This works. However, when I put more requirements on it (for instance, "also reserve 10 positions for it, and left align the outcome in that reserved space"), it may become convoluted. Using the **format**() function, you can display the requested value in a much easier and more readable way:

```
print( "{:.3f}".format( 7/11 ) )
```

format() is a function that "works" on a string. Up until this point, I have only used functions that get parameters. However, there are functions that work only on a particular data type, and are defined in such a way that a variable (or value) of that data type has to be placed in front of the function call, with a period in between. The reason why this is, has to do with something called "object orientation," which I will discuss in Chapters 20 to 23. For now, just know that such functions are called "methods," and to call them, you have to place the variable (or value) of the right data type in front of them, with a period in between. The variable (or value) that is used in this way is also accessible to the method, just like its parameters are.

So, the **format()** method (let's refer to it by its correct name, it is not a function but a method) is called as follows: `<string>.format()`. It will return a new string, which is a formatted version of the string for which it is called. It can take any number of parameters, and in the process of formatting, will insert these parameter values in particular places in the resulting string.

The places where **format()** inserts the parameter values in the string are indicated in the string by opening and closing curly brackets ({ and }). If you only use {} to refer to the parameters, it will process the string from left to right, and process the parameters from left to right, inserting them in the order that they are given. For example:

```
print( "The first 3 numbers are {}, {} and {}.".format(
    "one", "two", "three" ) )
```

If you want to process them in a different order, you can indicate the order by putting a number between the curly brackets. The first parameter has number 0, the second has number 1, the third has number 2, etcetera (if you find numbering starting with zero strange, then know that this is very common in programming languages and you will see this many more times). For example:

```
print( "Backwards they are {2}, {1} and {0}.".format(
    "one", "two", "three" ) )
```

format() can deal with parameters of any type, as long as they have a suitable string representation. For instance, it can deal with integers and floats, and you can mix those up with strings as you like:

```
print( "The first 3 numbers are {}, {} and {}.".format(
    "one", 2, 3.0 ) )
```

If you want to format the parameters in a more specific way, there are possibilities to do that, if you put a colon (:) in between the curly brackets, after the order number if you have one, and place some formatting instructions to the right of the colon. There are many possibilities for formatting instructions, and I will introduce only a few.

First I discuss some formatting instructions for string parameters. If you want to reserve a certain number of places for a string parameter, then you can indicate that with an integer

to the right side of the colon. This is called the "precision." The following code uses a precision of 7.

```
print( "The first 3 numbers are {:7}, {:7} and {:7}.".format(
    "one", "two", "three" ) )
```

If you do not reserve sufficient space for a parameter with the precision, **format**() will take as much space as it needs. So you cannot use the precision to, for instance, break off a string prematurely.

```
print( "The first 3 numbers are {:4}, {:4} and {:4}.".format(
    "one", "two", "three" ) )
```

If you use precision, you can align the parameter to the left, center, or right. You do that by placing an alignment character between the colon and the precision. Alignment characters are < for align left, ^ for align center, and > for align right.

```
print( "The first 3 numbers are {:>7}, {:^7} and {:<7}.".format(
    "one", "two", "three" ) )
```

Now I will discuss some number formatting instructions. If you want a number to be interpreted as an integer, you place a "d" to the right side of the colon. If instead you want it to be interpreted as a float, you place an "f." If you want to display an integer as a float, **format**() will do the necessary conversions for you. If you want to display a float as an integer, **format**() will cause a runtime error.

```
print( "{} divided by {} is {}".format( 1, 2, 1/2 ) )
print( "{:d} divided by {:d} is {:f}".format( 1, 2, 1/2 ) )
print( "{:f} divided by {:f} is {:f}".format( 1, 2, 1/2 ) )
```

Just as with strings, you can use precision and alignment with numbers. You use the same instruction characters, and place them between the colon and the `d` or `f`. And just as with strings, if the precision does not provide enough places, **format**() will take extra places as needed. Note that a preceding minus-sign and the decimal period each also take a place.

```
print( "{:5d} divided by {:5d} is {:5f}".format( 1, 2, 1/2 ) )
print( "{:<5f} divided by {:^5f} is {:>5f}".format( 1, 2, 1/2 ) )
```

Finally, and perhaps most useful, you can indicate how many decimals you want a floating point number to be displayed with, by placing a period and an integer to the left of the `f`. **format**() will round the parameter to the requested number of decimals. Note that you can indicate zero decimals using `.0`, which will display floats as integers.

```
print( "{:.2f} divided by {:.2f} is {:.2f}".format( 1, 2, 1/2 ) )
```

The combination of precision, alignment, and decimals, allows you to create nice, table-like displays.

listing0502.py

```
s = "{:>5d} times {:>5.2f} is {:>5.2f}"
print( s.format( 1, 3.75, 1 * 3.75 ) )
print( s.format( 2, 3.75, 2 * 3.75 ) )
print( s.format( 3, 3.75, 3 * 3.75 ) )
print( s.format( 4, 3.75, 4 * 3.75 ) )
print( s.format( 5, 3.75, 5 * 3.75 ) )
```

5.3 Modules

Python offers some basic functions, some of which are introduced above. Besides those, Python offers a large assortment of so-called "modules," which contain many more useful functions. To use functions from a module in your program, you have to import the module, by writing a line **import** <modulename> at the top of your code. You can then use all the functions in the module, though you have to precede the function calls with the name of the module and a period, e.g., to call the `sqrt()` function from the `math` module (which calculates the square root of a number), you call `math.sqrt()` after importing `math`.

Alternatively, you can import only specific functions from a module, by stating:
from <modulename> **import** <function1>, <function2>, <function3>, ...
The main advantage of importing specific functions from a module in this way is that in your code, you no longer need to precede the call to a function with the module name.

For example:

```
import math

print( math.sqrt( 4 ) )
```

is equivalent to:

```
from math import sqrt

print( sqrt( 4 ) )
```

If you want to rename something that you import from a module, you can do so with the keyword **as**. This might be useful when you use multiple modules that contain things with equal names.

```
from math import sqrt as squareroot

print( squareroot( 4 ) )
```

I will now introduce some functions from two standard modules that are often used, and some functions from a module which was developed for this book (you will learn to develop your own modules later). There are many more modules besides the ones introduced here, some of which will come up later in the book, and others which you will have to look

up by yourself by the time you need them in practice. However, you may assume that for any more-or-less general problem that you want to solve, someone has made a module that makes solving that problem simple or even trivial. So, in practice, do not start coding immediately, but first investigate whether you can exploit someone else's efforts.

5.3.1 `math`

The `math` module contains some useful mathematical functions. These functions have usually been implemented in a very efficient way, and in general they return a float. I will introduce only a few of these functions here (if you want to learn more of them, look up the `math` module in the Python reference):

- `exp()` gets one numerical parameter and returns e to the power of that parameter. If you do not remember e from math class: e is a special value that has many interesting properties, which have applications in physics, maths, and statistics.
- `log()` gets one numerical parameter and returns the natural logarithm of that parameter. The natural logarithm is the value which, when e is raised to the power of that value, gives the requested parameter. Just like e, the natural logarithm has many applications in physics, maths, and statistics.
- `log10()` gets one numerical parameter and returns the base-10 logarithm of that parameter.
- `sqrt()` gets one numerical parameter and returns the square root of that parameter.

For example:

listing0503.py

```
from math import exp, log

print( "The value of e is approximately", exp( 1 ) )
e_sqr = exp( 2 )
print( "e squared is", e_sqr )
print( "which means that log(", e_sqr, ") is", log( e_sqr ) )
```

5.3.2 `random`

The `random` module contains functions that return pseudo-random numbers. I say "pseudo-random" and not "random," because it is impossible for digital computers to generate actual random numbers. However, for all intents and purposes you may assume that the functions in the `random` module cough up random values.

- `random()` gets no parameters, and returns a random float in the range $[0, 1)$, i.e., a range that includes 0.0, but excludes 1.0.
- `randint()` gets two parameters, both integers, and the first should be smaller than or equal to the second. It returns a random integer in the range for which the two parameters are boundaries, e.g., `randint(2,5)` returns 2, 3, 4, or 5, with an equal chance for each of them.

- `seed()` initializes the random number generator of Python. If you want a sequence of random numbers that are always the same, start by calling `seed()` with a fixed value as parameter, for instance, 0. This can be useful for testing purposes. If you want to re-initialize the random number generator so that it starts behaving completely randomly again, call `seed()` without parameter.

For example:

listing0504.py

```
from random import random, randint, seed

seed()
print( "A random number between 1 and 10 is", randint( 1, 10 ) )
print( "Another is", randint( 1, 10 ) )
seed( 0 )
print( "3 random numbers are:", random(), random(), random() )
seed( 0 )
print( "The same 3 numbers are:", random(), random(), random() )
```

5.3.3 `pcinput`

`pcinput` is a module I wrote for this book. You can find it in Appendix C, and can easily recreate it (or simply download it from `http://www.spronck.net/pythonbook`). It contains four functions which are helpful for getting particular kinds of input from the user in a safe way. The functions are the following:

- `getInteger()` gets one string parameter, the prompt, and asks the user to supply an integer using that prompt. If the user enters something that is not an integer, the user is asked to enter a new input. The function will continue asking the user for inputs until a legal integer is entered, and then it will return that value, as an integer.
- `getFloat()` gets one string parameter, the prompt, and asks the user to supply a float using that prompt. If the user enters something that is not a float or an integer, the user is asked to enter a new input. The function will continue asking the user for inputs until a legal float or integer is entered, and then it will return that value, as a float.
- `getString()` gets one string parameter, the prompt, and asks the user to supply a string using that prompt. Any value that the user enters is accepted. The function will return the string that was entered, with leading and trailing spaces removed.
- `getLetter()` gets one string parameter, the prompt, and asks the user to supply a letter using that prompt. The user's input must be a single letter, in the range A to Z. Both capitals and lower case letters are accepted. The function returns the letter entered, converted to a capital.

These functions allow you to write code that asks the user for inputs of a specific data type, and guarantee that the input will indeed be of that data type, i.e., the code does not crash if the user enters something that is unacceptable. The functions are not very nicely designed, as they display messages in English when the user enters something that is wrong (so the functions are less useful if your code is meant to support a different language). But for the purpose of learning Python, they work fine.

Exercise Create or download the `pcinput` module, make sure that it is located in the folder where you write your Python code, then create a file with the code below in it. Run it, try to enter something else than an integer, and see what happens.

```
from pcinput import getInteger

num1 = getInteger( "Please enter an integer: " )
num2 = getInteger( "Please enter another integer: " )

print( "The sum of", num1, "and", num2, "is", num1 + num2 )
```

Exercise Ask the user to supply a string. Then use that string as a prompt to ask for a float.

Note: I do not explain here how the functions of `pcinput` work, as they are implemented using concepts that are discussed much later in the book. You will learn, in time, how to develop such functions yourself. For now, do not worry about how they work, but just use them. This is the attitude that you should have towards most standard functions: as long as you know what they do, which parameters they need, and what they return, you do not need to spend time considering how they work.

What you learned

In this chapter, you learned about:

- What functions are
- Function names
- Function parameters
- Function return values
- Details of type casting with **`float`**`()`, **`int`**`()`, and **`str`**`()`
- Basic calculation functions **`abs`**`()`, **`max`**`()`, **`min`**`()`, **`pow`**`()`, and **`round`**`()`
- **`len`**`()`
- **`input`**`()`
- Details of the **`print`**`()` function
- String formatting using **`format`**`()`
- What modules are
- The `math` module functions `exp()`, `log()`, `log10()`, and `sqrt()`
- The `random` module functions `random()`, `randint()`, and `seed()`
- The `pcinput` module functions `getInteger()`, `getFloat()`, `getString()`, and `getLetter()`

Exercises

Exercise 5.1 Ask the user to enter a string. Then print the length of that string. Use the `input()` function rather that the `getString()` function from `pcinput`, as the `getString()` function removes leading and trailing spaces.

Exercise 5.2 The Pythagorean theorem states that of a right triangle, the square of the length of the diagonal side is equal to the sum of the squares of the lengths of the other two sides (or $a^2 + b^2 = c^2$). Write a program that asks the user for the lengths of the two sides that meet at a right angle, then calculate the length of the third side (in other words: take the square root of the sum of the squares of the two sides that you asked for), and display it in a nicely formatted way. You may ignore the fact that the user can enter negative or zero lengths for the sides.

Exercise 5.3 Ask the user to enter three numbers. Then print the largest, the smallest, and their average, rounded to 2 decimals.

Exercise 5.4 Calculate the value of `e` to the power of -1, 0, 1, 2, and 3, and display the results, with 5 decimals, in a nicely formatted manner.

Exercise 5.5 Suppose you want to generate a random integer between 1 and 10 (1 and 10 both included), but from the random module you only have the `random()` function available (you can use functions from other modules, though). How do you do that?

Chapter 6

Conditions

In program code, there are often statements that you only want to execute when certain conditions hold. Every programming language therefore supports conditional statements. In this chapter I will explain how to use conditions in Python.

6.1 Boolean expressions

A conditional statement, often called an "if"-statement, consists of a test and one or more actions. The test is a so-called "boolean expression." The actions are executed when the test evaluates to True. For instance, an app on a smartphone might give a warning if the battery level is lower than 5%. This means that the app needs to check if a certain variable `battery_level` is lower than the value 5, i.e., if `battery_level < 5` evaluates to **`True`**. If the variable `battery_level` holds the value 17, then `battery_level < 5` evaluates to **`False`**.

6.1.1 Booleans

`True` and **`False`** are so-called "boolean values" that are predefined in Python. **`True`** and **`False`** are actually the *only* boolean values, and anything that is not **`False`**, is **`True`**.

You might wonder what the data type of **`True`** and **`False`** is. The answer is that they are of the type **`bool`**. However, in Python every value can be interpreted as a boolean value, regardless of its data type. I.e., when you test a condition, and your test is of a value that is not **`True`** or **`False`**, it will still be interpreted as either **`True`** or **`False`**.

The following values are interpreted as **`False`**:

- The special value **`False`**
- The special value **`None`** (which you encountered in Chapter 5)
- Every numerical value that is zero, e.g., 0 and 0.0
- Every empty sequence, e.g., an empty string ("")

- Every empty "mapping," e.g., an empty dictionary (dictionaries follow in Chapter 13)
- Any function or method call that returns one of these listed values (this includes functions that return nothing)

Every other value is interpreted as **`True`**.

Any expression that is evaluated as **`True`** or **`False`** is called a "boolean expression."

6.1.2 Comparisons

The most common boolean expressions are comparisons. A comparison consists of two values, and a comparison operator in between. Comparison operators are:

```
<    less than
<=   less than or equal to
==   equal to
>=   equal to or greater than
>    greater than
!=   not equal
```

A common mistake is to use a single = as a comparison operator, as the single = is the assignment operator. In general, Python will produce a syntax or runtime error if you try to use a single = to make a comparison.

You can use the comparison operators to compare both numbers and strings. Comparison for strings is an alphabetical comparison, whereby all capitals come before all lower case letters (and digits come before both of them). More on this will follow in Chapter 10.

Here are some examples of the results of comparisons:

listing0601.py

```
print( "1.", 2 < 5 )
print( "2.", 2 <= 5 )
print( "3.", 3 > 3 )
print( "4.", 3 >= 3 )
print( "5.", 3 == 3.0 )
print( "6.", 3 == "3" )
print( "7.", "syntax" == "syntax" )
print( "8.", "syntax" == "semantics" )
print( "9.", "syntax" == " syntax" )
print( "10.", "Python" != "rubbish" )
print( "11.", "Python" > "Perl" )
print( "12.", "banana" < "orange" )
print( "13.", "banana" < "Orange" )
```

Make sure that you run these evaluations, check their outcome, and understand them!

Exercise Do you understand why `3 < 13` is **`True`**, but `"3" < "13"` is **`False`**? Think about it!

You can assign the outcome of a boolean expression to a variable if you like:

```
greater = 5 > 2
print( greater )
greater = 5 < 2
print( greater )
print( type( greater ) )
```

Exercise Write some code that allows you to test if $1/2$ is greater than, equal to, or less than 0.5. Do the same for $1/3$ and 0.33. Then do the same for $(1/3) * 3$ and 1.

Comparisons of data types that cannot be compared, in general lead to runtime errors.

```
# This code gives a runtime error.
print( 3 < "3" )
```

6.1.3 `in` operator

Python has a special operator called the "membership test operator," which is usually abbreviated to the "in operator" as it is written as **`in`**. The **`in`** operator tests if the value to the left side of the operator is found in a "collection" to the right side of the operator.

At this time, I have discussed only one "collection," which is the string. A string is a collection of characters. You can test if a particular character or a sequence of characters is part of the string using the **`in`** operator. The opposite of the **`in`** operator is the **`not in`** operator, which gives **`True`** when **`in`** gives **`False`**, and which gives **`False`** when **`in`** gives **`True`**. For example:

```
print( "y" in "Python" )
print( "x" in "Python" )
print( "p" in "Python" )
print( "th" in "Python" )
print( "to" in "Python" )
print( "y" not in "Python" )
```

Again, make sure that you understand these evaluations!

Exercise Write some code that allows you to test for each vowel whether it occurs in your name. You may ignore capitals.

6.1.4 Logical operators

Boolean expressions can be combined with logical operators. There are three logical operators, **`and`**, **`or`**, and **`not`**.

`and` and **`or`** are placed between two boolean expressions. When **`and`** is between two boolean expressions, the result is **`True`** if and only if both expressions evaluate to **`True`**; otherwise it is **`False`**.

When **or** is between two boolean expressions, the result is **True** when one or both of the expressions evaluate to **True**; it is only **False** if both expressions evaluate to **False**.

not is placed in front of a boolean expression to switch it from **True** to **False** or vice versa.

For example:

```
t = True
f = False
print( t and t )
print( t and f )
print( f and t )
print( f and f )
print( t or t )
print( t or f )
print( f or t )
print( f or f )
print( not t )
print( not f )
```

You have to be careful with logical operators, because combinations of **and**s and **or**s might lead to unexpected results. To ensure that they are evaluated in the order that you intend, you can use parentheses. For example, rather than writing `a and b or c` you should write `(a and b) or c` or you should write `a and (b or c)` (depending on the order in which you want to evaluate the logical operators), so that it is immediately clear from your code which evaluation you want the code to do. Even if you know the order in which the logical operators are processed by Python, someone else who reads your code might not.

Exercise For the code below, give values **True** or **False** to each of the variables `a`, `b`, and `c`, so that the two expressions evaluate to different values.

listing0602.py

```
a = # True or False?
b = # True or False?
c = # True or False?

print( (a and b) or c )
print( a and (b or c) )
```

If all the logical operators in a boolean expression are **and**, or they all are **or**, the use of parentheses is not needed, since there is only one possible evaluation of the expression.

Boolean expressions are processed from left to right, and Python will stop the processing of an expression when it already knows whether it will end in **True** or **False**. Take, for instance, the following code:

```
x = 1
y = 0
print( (x == 0) or (y == 0) or (x / y == 1) )
```

When you divide by zero, Python gives a runtime error, so the expression `x / y == 1` crashes the program if `y` is zero. And `y` actually is zero. However, when Python processed the whole boolean expression, at the point where it tested `y == 0` it determined that the expression as a whole is **`True`**, because if any of the expressions that are connected by an **`or`** to the expression as a whole is **`True`**, then the whole expression is **`True`**. So there was no need for Python to determine the value of `x / y == 1`, and it did not even attempt to evaluate it. Of course, the test `y == 0` must be to the left of `x / y == 1`, so that Python will test `y == 0` first.

Note: While you can make truly complex boolean expressions using logical operators, I recommend that you keep your expressions simple if possible. Simple boolean expressions make code readable.

6.2 Conditional statements

Conditional statements are, as the introduction to this chapter said, statements consisting of a test and one or more actions, whereby the actions only get executed if the test evaluates to **`True`**. Conditional statements are also called "if-statements," as they are written using the special keyword **`if`**.

Here is an example:

```
x = 5
if x == 5:
    print( "x equals 5" )
```

The syntax of the **`if`** statement is as follows:

```
if <boolean expression>:
    <statements>
```

Note the colon (:) after the boolean expression, and the fact that `<statements>` is indented.

6.2.1 Code blocks

In the syntactic description of the **`if`** statement above, you see that the `<statements>` are "indented," i.e., they are placed one tabulation to the right. This is intentional and necessary. Python considers statements that are following each other and that are at the same level of indentation part of a code block. The code block underneath the first line of the **`if`** statement is considered to be the list of actions that are executed when the boolean expression evaluates to **`True`**. For example:

listing0603.py

```
x = 7
if x < 10:
    print( "This line is only executed if x < 10." )
    print( "And the same holds for this line." )
print( "This line, however, is always executed." )
```

Exercise Change the value of `x` to see how it affects the outcome of the code.

Thus, all the statements under the **`if`** that are indented, belong to the code block that is executed when the boolean expression of the **`if`** statement evaluates to **`True`**. This code block is skipped if the boolean expression evaluates to **`False`**. Statements which follow the **`if`** construction which are not indented (as deep as the code block under the **`if`**), are executed, regardless of whether the boolean expression evaluates to **`True`** or **`False`**.

Naturally, you are not restricted to having just a single if statement in your code. You can have as many as you like.

listing0604.py

```
x = 5
if x == 5:
    print( "x equals 5" )
if x > 4:
    print( "x is greater than 4" )
if  x >= 5:
    print( "x is greater than or equal to 5" )
if x < 6:
    print( "x is less than 6" )
if x <= 5:
    print( "x is less than or equal to 5" )
if x != 6 :
    print( "x does not equal 6" )
```

Exercise Again, try changing the value of `x` and see how it affects the outcome.

6.2.2 Indentation

In Python, **correct indenting is of the utmost importance**! Without correct indentation, Python will not be able to recognize which statements belong together as one code block, and therefore cannot execute your code correctly.[1]

Note that you can indent using the `Tab` key, or indent using spaces. Most editors will auto-indent for you, i.e., if, for instance, you write the first line of an **`if`** statement, once you press `Enter` to go to the next line, it will automatically "jump in" one level of indentation (if it does not, it is very likely that you forgot the colon at the end of the conditional expression). Also, when you have indented one line to a certain level of indentation, the next line will use the same level. You can get rid of indentations using the `Backspace` key.

[1] In many programming languages (actually, in almost all programming languages), code blocks are recognized by having them start and end with a specific symbol or keyword. For instance, in languages such as Java and C++, code blocks are enclosed by curly brackets, while in languages such as Pascal and Modula, code blocks are started with the keyword `begin` and ended with the keyword `end`. That means that in almost all languages, indenting to recognize code blocks is not necessary. However, you will find that code written by capable programmers is always nicely indented, regardless of the language. This makes it easy to see which code belongs together, for instance, which commands belong to an `if` statement. Python makes indenting a requirement. While for experienced programmers who are new to Python this seems strange at first, they quickly find that they do not care – they were indenting nicely anyway, and Python's strategy makes that beginning programmers are also required to write nice-looking code.

For Python programs, a normal level of indentation is four spaces, i.e., one press of the `Tab` key should "jump in" four spaces. As long as you are in one editor, you can in such a case either use the `Tab` key, or press the spacebar four times, to go up one indentation level. So far so good. You may get into problems, however, if you port your code to another editor, which might have a different setting for the `Tab` key. If you edit your code in a such a different editor, even though it might look okay, Python may see that there are indentation conflicts (a mix of tabulations and space-indentations) and may report a syntax error when you try to run your code. Most editors therefore offer the option to automatically replace tabulations with spaces, so that such problems do not arise. If you use a text editor to write Python code, check if it contains such an option, and if so, ensure that tabulations are set to 4 and are automatically replaced by spaces.

Exercise The following code contains multiple indentation errors. Fix them all.

listing0605.py

```
# This code contains indentation errors!
x = 3
y = 4
if x == 3 and y == 4:
    print( "x is 3" )
   print( "y is 4" )
if x > 2 and y < 5:
print( "x > 2" )
print( "y < 5" )
if x < 4 and y > 3:
    print( "x < 4" )
        print( "y > 3" )
```

6.2.3 Two-way decisions

Often a decision branches, e.g., if a certain condition arises, you want to take a particular action, but if it does not arise, you want to take another action. This is supported by Python in the form of an expansion to the **`if`** statement that adds an **`else`** branch.

```
x = 4
if x > 2:
    print( "x is bigger than 2" )
else:
    print( "x is smaller than or equal to 2" )
```

The syntax is as follows:

```
if <boolean expression>:
    <statements>
else:
    <statements>
```

Note the colon (:) after both the boolean expression and the **`else`**.

It is important that the word **`else`** is aligned with the word **`if`** that it belongs to. If you do not align them correctly, this results in an indentation error.

A consequence of adding an **else** branch to an **if** statement is that always exactly one of the two code blocks will be executed. If the boolean expression of the **if** statement evaluates to **True**, the code block directly under the **if** will be executed, and the code block directly under the **else** will be skipped. If it evaluates to **False**, the code block directly under the **if** will be skipped, while the code block directly under the **else** will be executed.

Exercise You can test whether an integer is odd or even using the modulo operator. Specifically, when `x%2` equals zero, then `x` is even, else it is odd. Write some code that asks for an integer and then reports whether it is even or odd (you can use the `getInteger()` function from `pcinput` to ask for an integer).

Note: As far as indentation is concerned, it is not absolutely necessary to have the code block under the **else** branch use the same number of spaces in indentation as the code block under the **if** branch, as long as the indentation is consistent within the code block. However, accomplished programmers use consistent indentation throughout their programs, which makes it easier to see what the whole **if**-**else** statement encompasses. For example, in the code below the indentation in the **else** branch uses less spaces than the indentation in the **if** branch. While syntactically correct, this makes the code less readable.

```
# Example of syntactically correct but ugly indenting.
x = 1
if x > 2:
    print( "x is bigger than 2" )
else:
  print( "x is smaller than or equal to 2" )
```

6.2.4 Flow charts

In the early days of programming, programmers often expressed the algorithms they wanted to implement by means of so-called "flow charts." Nowadays, flow charts are

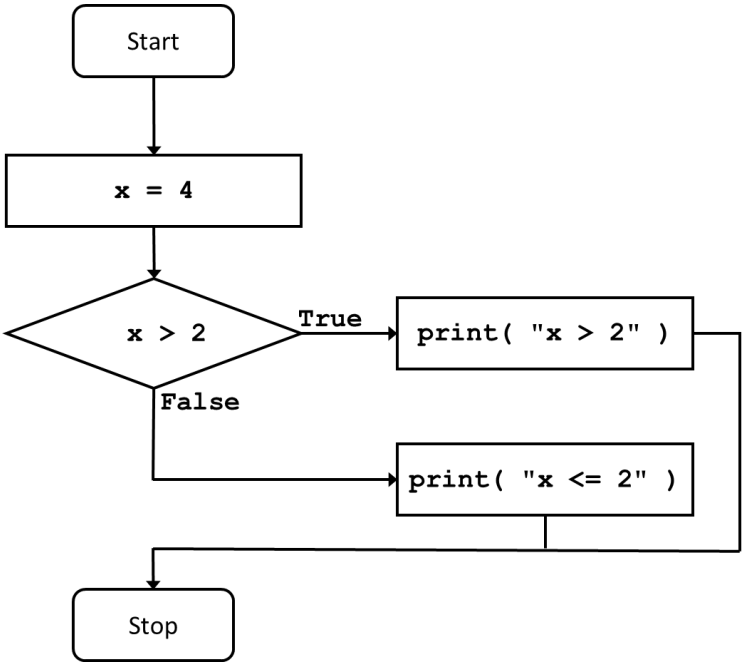


Figure 6.1: Flow chart expressing a two-way decision.

seldom used anymore. However, I found that it often helps students understanding how exactly conditions (and, for the next chapter, iterations) work if they are shown a flow chart of the process.

A flow chart is a diagram that shows different kinds of blocks with arrows in between. The blocks are of three kinds (at least, that is all I need for this book). Rectangular blocks contain statements that are executed. Diamond-shaped blocks contain a condition that evaluates to either **`True`** or **`False`**. Rectangular blocks with rounded corners indicate either the start (using the text "Start") or end (using the text "Stop") of the algorithm.

To interpret a flow chart, you begin at the "Start" block, and follow the arrows, executing all statements that you encounter. When you get to a diamond-shaped block, you evaluate the condition in it, and then either follow the arrow marked **`True`** in case it evaluates to **`True`**, or the arrow marked **`False`** when it evaluates to **`False`**. When you encounter the "Stop" block, you are finished.

For example, the code shown above, that checks if a number is greater than 2 or smaller than or equal to 2, and prints some text that states as much, is equivalent to the flow chart shown in Figure 6.1.

6.2.5 Multi-branch decisions

Occasionally, you encounter multi-branch decisions, where one of multiple blocks of commands has to be executed, but never more than one block. Such multi-branch decisions can be implemented using a further expansion of the **`if`** statement, namely in the form of one or more **`elif`** statements (**`elif`** stands for "else if").

listing0606.py

```
age = 21
if age < 12:
    print( "You're still a child!" )
elif age < 18:
    print( "You are a teenager!" )
elif age < 30:
    print( "You're pretty young!" )
elif age < 50:
    print( "Wisening up, are we?" )
else:
    print( "Aren't the years weighing heavy?" )
```

This code is equivalent to the algorithm expressed by the flow chart in Figure 6.2.

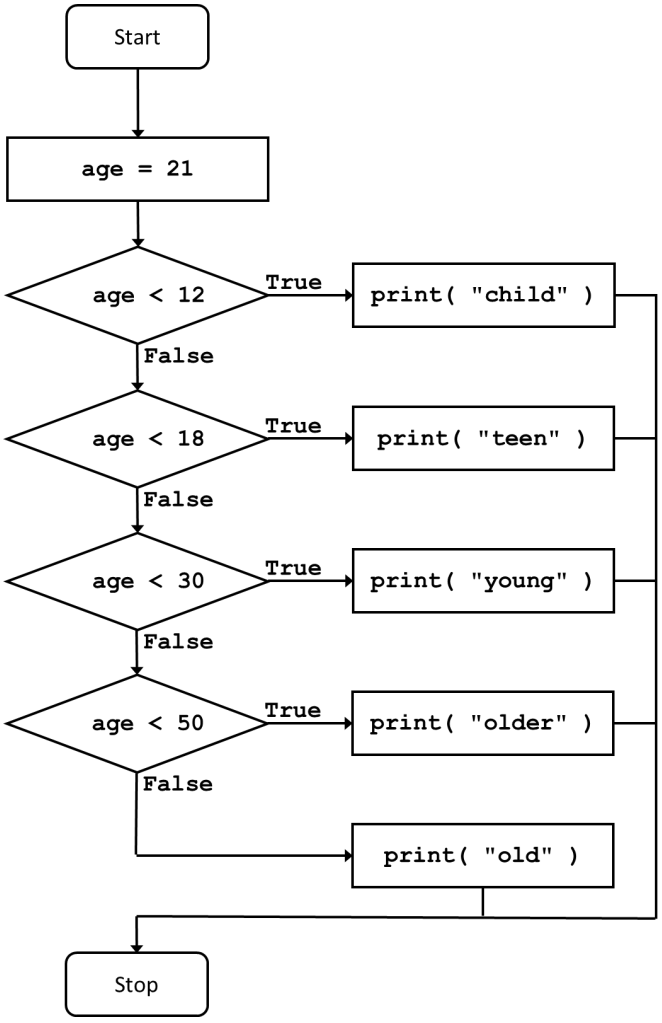


Figure 6.2: Flow chart expressing a multi-branch decision.

Exercise In the code block above, change age to different values and observe the results.

The syntax is as follows:

```
if <boolean expression>:
    <statements>
elif <boolean expression>:
    <statements>
else:
    <statements>
```

The syntax above shows only one **elif**, but you can have multiple. The different tests in an **if**-**elif**-**else** construct are executed in order. The first boolean expression that evaluates to **True** will cause the code block that belongs to that expression to be executed. None of the other code blocks of the construct will be executed.

So, first the boolean expression next to the **if** will be evaluated. If it evaluates to **True**, the code block underneath the **if** will be executed. Otherwise, the boolean expression for the first **elif** will be evaluated. If that turns out to be **True**, the code block underneath it will be executed. Otherwise, the boolean expression for the next **elif** will be evaluated. Etcetera. Only when the boolean expressions for the **if** and all of the **elif**s evaluate to **False**, the code block underneath the **else** will be executed.

Therefore, in the code above, you do not need to test `age >= 12` **and** `age < 18` for the first **elif**. Just testing `age < 18` suffices, because if age was smaller than 12, already the boolean expression for the **if** would have evaluated to **True**, and the boolean expression for the first **elif** would not even have been encountered by Python.

Note that the inclusion of the **else** branch is always optional. However, in most cases where I need **elif**s I include it anyway, if only for error checking.

Exercise Write a program that defines a variable `weight`. If `weight` is greater than 20 (kilo's), print: "There is a $25 surcharge for luggage that is too heavy." If `weight` is smaller than 20, print: "Have a safe flight!" If `weight` is exactly 20, print: "Pfew! The weight is just right!" Make sure that you change the value of `weight` a couple of times to check whether your code works.

6.2.6 Nested conditions

Given the rules of the **if**-**elif**-**else** statements and identation, it is perfectly possible to use an **if** statement within another **if** statement. This second **if** statement is only executed if the condition for the first **if** statement evaluates to **True**, as it belongs to the code block of the first **if** statement. This is called "nesting."

listing0607.py

```
x = 41
if x%7 == 0:
    # --- Here starts a nested block of code ---
    if x%11 == 0:
        print( x, "is dividable by both 7 and 11." )
    else:
        print( x, "is dividable by 7, but not by 11." )
```

```
    # --- Here ends the nested block of code ---
elif x%11 == 0:
    print( x, "is dividable by 11, but not by 7." )
else:
    print( x, "is dividable by neither 7 nor 11." )
```

This code is equivalent to the algorithm expressed in Figure 6.3.

Exercise Change the value of `x` and observe the results.

Note that the example above is just to illustrate nesting; you probably already know different ways of getting the same results that are a bit more readable. In particular, nesting of **`if`** statements can often be avoided by judicious use of **`elif`**s. To give an example, here is a "nested" example of the age code given above:

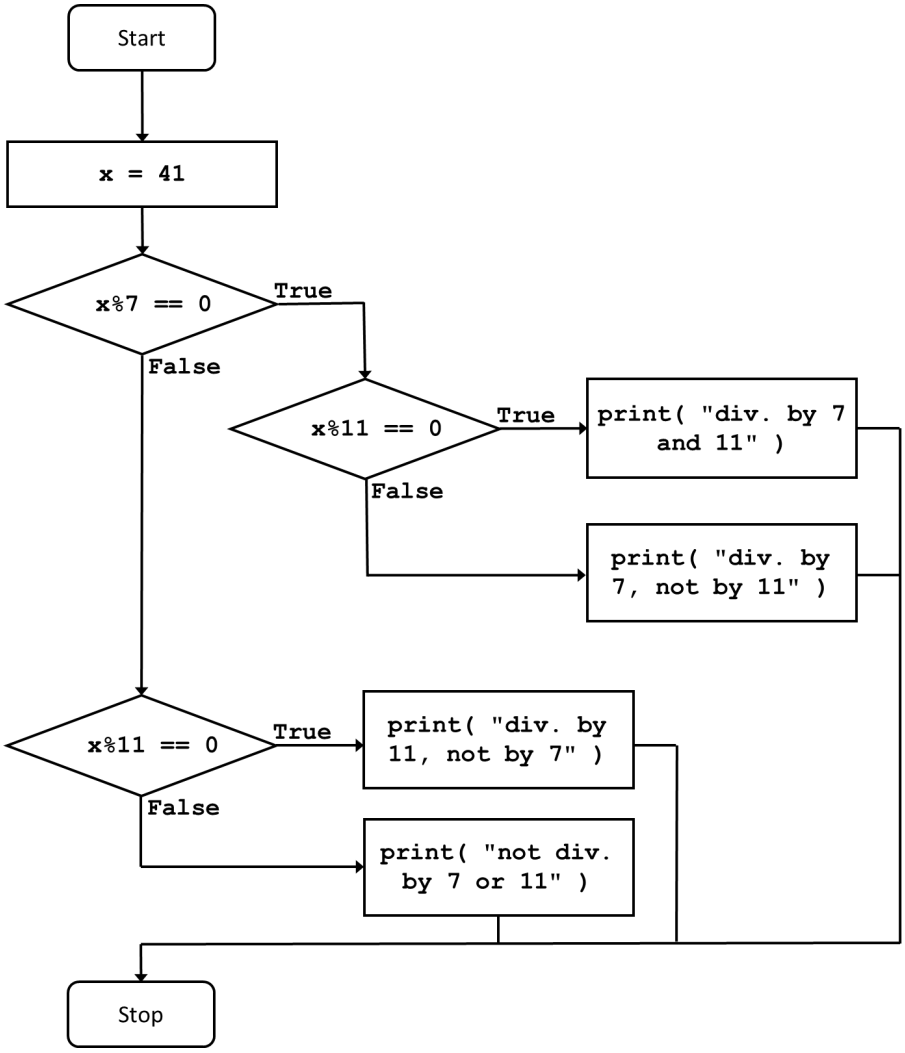


Figure 6.3: Flow chart expressing a nested condition.

listing0608.py

```
age = 21
if age < 12:
    print( "You're still a child!" )
else:
    if age < 18:
        print( "You are a teenager!" )
    else:
        if age < 30:
            print( "You're pretty young!" )
        else:
            if age < 50:
                print( "Wisening up, are we?" )
            else:
                print( "Aren't the years weighing heavy?" )
```

I assume you agree that the version with the **elif**s looks better.

6.3 Early exits

Occasionally it happens that you want to exit a program early when a certain condition arises. For instance, your program asks the user for a value, and then processes that value extensively. But if the user enters a value that cannot be processed, the program should just end. You could code that as follows:

```
from pcinput import getInteger

num = getInteger( "Please enter a positive integer: " )
if num < 0:
    print( "You should have entered a positive integer!" )
else:
    print( "Now I am processing your integer", num )
    print( "Lots and lots of processing" )
    print( "Hundreds of lines of code here" )
```

It is a bit irritating that most of your program is already one indent deep, while you would have preferred to leave the program at the error message, and then have the rest of the program at the top indent level.

You can do that using a special function `exit()` that is found in the module `sys`. The code above becomes:

listing0609.py

```
from pcinput import getInteger
from sys import exit

num = getInteger( "Please enter a positive integer: " )
if num < 0:
```

```
    print( "You should have entered a positive integer!" )
    exit()

print( "Now I am processing your integer", num )
print( "Lots and lots of processing" )
print( "Hundreds of lines of code here" )
```

When you run this code and enter a negative number, depending on which editor you use (IDLE does not have this issue), you may find that Python raises a `SystemExit` exception, which looks like a big, ugly error. It is not, however. This exception just says that you forced the program to end, but that is exactly what you wanted. This is actually a nice, clean exit.

In general, you are not allowed to ignore error messages and warnings. This one is the exception to the rule. You are allowed to exit your program this way. In Chapter 8 I will explain how you can suppress this ugly message (if you get it and if you really want to), but for now, just accept it.

What you learned

In this chapter, you learned about:

- What boolean expressions are
- Boolean values **`True`** and **`False`**
- Comparisons with <, <=, ==, >, >=, and !=
- The **`in`** operator
- Logical operators **`and`**, **`or`**, and **`not`**
- Conditional statements using **`if`**, **`elif`**, and **`else`**
- Code blocks
- Indentation
- Nested conditions
- Using `exit()`

Exercises

Exercise 6.1 Grades are values between zero and 10 (both zero and 10 included), and are always rounded to the nearest half point. To translate grades to the American style, 8.5 to 10 become an "A," 7.5 and 8 become a "B," 6.5 and 7 become a "C," 5.5 and 6 become a "D," and other grades become an "F." Implement this translation, whereby you ask the user for a grade, and then give the American translation. If the user enters a grade lower than zero or higher than 10, just give an error message. You do not need to handle the user entering grades that do not end in .0 or .5, though you may do that if you like – in that case, if the user enters such an illegal grade, give an appropriate error message.

Exercise 6.2 Can you spot the reasoning error in the following code?

exercise0602.py

```
score = 98.0
if score >= 60.0:
    grade = 'D'
elif score >= 70.0:
    grade = 'C'
elif score >= 80.0:
    grade = 'B'
elif score >= 90.0:
    grade = 'A'
else:
    grade = 'F'
print( grade )
```

Exercise 6.3 Ask the user to supply a string. Print how many different vowels there are in the string. The capital version of a lower case vowel is considered to be the same vowel. y is not considered a vowel. Try to print nice output (e.g., printing "There are 1 different vowels in the string" is ugly). Example: When the user enters the string "It's Owl Stretching Time," the program should say that there are 3 different vowels in the string.

Exercise 6.4 You can solve quadratic equations using the quadratic formula. Quadratic equations are of the form $Ax^2 + Bx + C = 0$. Such equations have zero, one or two solutions. The first solution is $(-B + sqrt(B^2 - 4AC))/(2A)$. The second solution is $(-B - sqrt(B^2 - 4AC))/(2A)$. There are no solutions if the value under the square root is negative. There is one solution if the value under the square root is zero. Write a program that asks the user for the values of A, B, and C, then reports whether there are zero, one, or two solutions, then prints those solutions. Note: Make sure that you also take into account the case that A is zero (there is only one solution then, namely $-C/B$), and the case that both A and B are zero.

Chapter 7

Iterations

Computers do not get bored. If you want the computer to repeat a certain task hundreds of thousands of times, it does not protest. Humans hate too much repetition. Therefore, repetitious tasks should be performed by computers. All programming languages support repetitions. The general class of programming constructs that allow the definition of repetitions are called "iterations." A term which is even more common for such tasks is "loops."

This chapter explains all you need to know about loops in Python. Students who are completely new to programming often find loops the first really hard topic in programming that they encounter. If that is the case for you, then make sure you take your time for this chapter, and work on it until you understand it completely. Loops are such a basic concept in programming that you need to understand them in all their details. Each and every chapter after this one needs loops.

7.1 `while` loop

Suppose you have to ask the user for five numbers, then add them up, and show the total. With the material from the previous chapters, you would program that as follows:

```
from pcinput import getInteger

num1 = getInteger( "Number 1: " )
num2 = getInteger( "Number 2: " )
num3 = getInteger( "Number 3: " )
num4 = getInteger( "Number 4: " )
num5 = getInteger( "Number 5: " )

print( "Total is", num1 + num2 + num3 + num4 + num5 )
```

But what if I want you to ask the user for 500 numbers? Are you going to create a block of code of more than 500 lines long? Surely there must be an easier way to do this?

Of course there is. You can use a loop to do this.

The first loop I am going to present to you is the **while** loop. A **while** statement is quite similar to an **if** statement. The syntax is:

```
while <boolean expression>:
    <statements>
```

Just like an **if** statement, the **while** statement tests a boolean expression, and if the expression evaluates to **True**, it executes the code block below it. However, contrary to the **if** statement, once the code block has finished, the code "loops" back to the boolean expression to test it again. If it still evaluates to **True**, the code block below it gets executed once more. And after it has finished, it loops back again, and again, and again...

Note: if the boolean expression immediately evaluates to **False**, then the code block below the **while** is skipped completely, just like with an **if** statement.

7.1.1 `while` loop, first example

Let's take a simple example: printing the numbers 1 to 5. With a **while** loop, that can be done as follows:

listing0701.py

```
num = 1
while num <= 5:
    print( num )
    num += 1
print( "Done" )
```

This code is also expressed by the flow chart in Figure 7.1.

It is crucial that you understand this code, so let's discuss it step by step.

The first line initializes a variable `num`. This is the variable that the code will print, so it is initialized to 1, as 1 is the first value that must be printed.

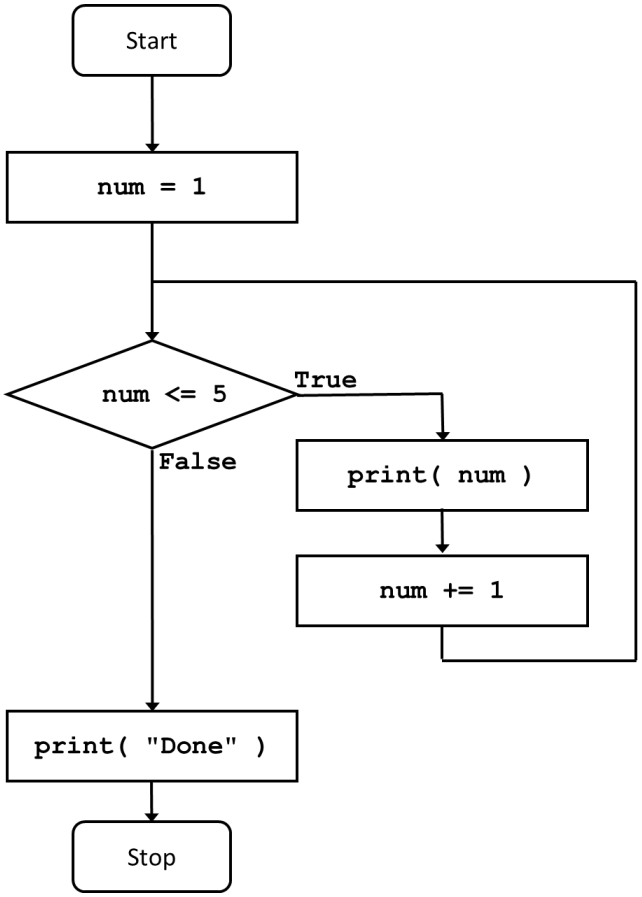


Figure 7.1: Flow chart expressing a `while` loop.

Then the **`while`** loop starts. The boolean expression says `num <= 5`. Since `num` is 1, and 1 is actually smaller than (or equal to) 5, the boolean expression evaluates to **`True`**. Therefore, the code block below the **`while`** gets executed.

The first line of the code block below the **`while`** prints the value of `num`, which is 1. The second line adds 1 to the value of `num`, which makes `num` hold the value 2. Then the code loops back to the boolean expression (i.e., the last line of the code, the printing of "Done," is not executed as it is not part of the loop and the loop has not finished yet).

Since `num` is 2, the boolean expression still evaluates to **`True`**. The code block gets executed once more. 2 is displayed, `num` gets the value 3, and the code loops back to the boolean expression.

Since `num` is 3, the boolean expression still evaluates to **`True`**. The code block gets executed once more. 3 is displayed, `num` gets the value 4, and the code loops back to the boolean expression.

Since `num` is 4, the boolean expression still evaluates to **`True`**. The code block gets executed once more. 4 is displayed, `num` gets the value 5, and the code loops back to the boolean expression.

Since `num` is 5, the boolean expression still evaluates to **`True`** (because `5 <= 5`). The code block gets executed once more. 5 is displayed, `num` gets the value 6, and the code loops back to the boolean expression.

Since `num` is 6, the boolean expression now evaluates to **`False`** (because 6 is bigger than 5). Therefore, the code block gets skipped, and the code continues with the first line below the code block, which is the last line of the code. The word "Done" is printed, and the code ends.

Exercise Change the code above so that it prints the numbers 1, 3, 5, 7, and 9.

7.1.2 `while` loop, second example

If you understand the first example, you probably also understand how to ask the user for five numbers and print the total. This is implemented as follows:

listing0702.py

```
from pcinput import getInteger

total = 0
count = 0
while count < 5:
    total += getInteger( "Please give a number: " )
    count += 1

print( "Total is", total )
```

Study this code closely. There are two variables used. `total` is used to add up the five numbers that the user enters. It is started at zero, as at the start of the code the user has not yet entered any numbers, so the total is still zero. `count` is used to count how often the code has gone through the loop. Since the loop must be done five times, `count` is started at 0 and the boolean expression tests if `count` smaller is than 5. Since in the loop `count` gets increased by 1 at the end of every cycle through the loop, the loop gets processed five times before the boolean expression **`False`** is.

You may wonder why `count` is started at 0 and the boolean expression checks if `count < 5`. Why not start `count` at 1 and check if `count <= 5`? The reason is convention: programmers are used to start indices at 0, and if they count, they count "up to but not including." When you continue with programming, you will find that most code sticks to this convention. Most standard programming constructs that use indices or count things apply this convention too. My advice is therefore that you get used to it, as it makes code easier to read.

Note: The variable `count` is what programmers call a "throw-away variable." Its only purpose is to count how often the loop has been cycled through, and it has no real meaning before the loop, in the loop, or after the loop has ended. Programmers often choose a single-character variable name for such a variable, usually `i` or `j`. In this example I chose the name `count` because it is illustrative of what the variable does for the code, but a single-character name for this variable would have been acceptable.

Exercise Change the code block above so that it not only prints the total, but also the average of the five numbers.

Exercise The first code block of this chapter also asks the user for five numbers, and prints the total. However, that code block uses "Enter number `x`: " as a prompt, whereby `x` is a digit. Can you change the code block above so that it also uses such a changing prompt to ask for each number?

7.1.3 Putting the `while` loop under user control

Suppose that, in the second example, you do not want the user to be restricted to entering exactly five numbers. You want the user to enter as many numbers as he wants, including none. This means that you cannot predict how many iterations through the **`while`** loop are needed. Instead, it is the user who controls when the loop ends. You therefore have to give the user the means to indicate that the loop should end.

The code block below shows how to use a **`while`** loop to allow the user to enter numbers as long as he wants, until he enters a zero. Once a zero is entered, the total is printed, and the program ends.

listing0703.py

```
from pcinput import getInteger

num = -1
total = 0
while num != 0:
    num = getInteger( "Enter a number: " )
    total += num
print( "Total is", total )
```

This code works, but there are (at least) two ugly things about it. First, `num` is initialized to -1. The -1 is meaningless, I just needed an initialization that would ensure that the **`while`** loop would be entered at least once. Second, when the user enters zero, `total` still gets increased by `num`. Since `num` is zero at that point, it does not matter for the total, but if I wanted the user to end the program by typing something else (for instance, a negative number), then `total` would now hold the wrong value.

Because of these ugly elements, some programmers prefer to write this code as follows:

listing0704.py

```
from pcinput import getInteger

num = getInteger( "Enter a number: " )
total = 0
while num != 0:
    total += num
    num = getInteger( "Enter a number: " )
print( "Total is", total )
```

This solves the ugly parts from the previous code, but introduces something new that is ugly, namely the repetition of the `getInteger()` function. How this can be solved follows at the end of this chapter. For now, make sure that you understand how **`while`** loops work.

Exercise Create a loop that lets the user enter some numbers until he enters zero, and then prints their total and their average. Make sure you test the loop with no numbers entered, and with several copies of the same number entered.

7.1.4 Endless loops

The code below is supposed to start at number 1, and add up numbers, until it encounters a number that, when squared, is divisible by 1000. The code contains an error, though. See if you can spot it (without running the code!).

```
number = 1
total = 0
while (number * number) % 1000 != 0:
    total += number
print( "Total is", total )
```

The heading of this subsection gave away the answer, of course: this code contains a loop that never terminates. If you run it, it looks like the program "hangs," i.e., sits there and does nothing. It is not doing nothing, it is actually highly active, but it is in a never-ending addition. `number` starts at 1, and is never increased in the loop, so the boolean expression will always be **`True`**. This is called an "endless loop," and it is the single one great danger in using **`while`** loops.

If you run this code in IDLE, you can stop it by pressing `Ctrl-C`. Other editors may have menu options to interrupt code execution. In user-unfriendly environments, you may actually have to "kill" the process that runs the code using a system command.

Since every programmer writes endless loops by accident now and again, it is good practice when you program a loop to immediately add a statement to the loop that makes a change that is tested in the boolean expression, so that you do not forget about it. I.e., if you write **`while`** `i < 10:`, immediately add a line `i += 1` below it, and then start adding the rest of your code in between.

Exercise Fix the code above so that it no longer is an endless loop.

7.1.5 `while` loop practice exercises

You should now practice a bit with simple **`while`** loops.

Exercise Write countdown code. It starts with a given number (e.g., 10), and counts down to zero, printing each number it encounters (10, 9, 8, ...). It does not print 0, instead it prints "Blast off!"

Exercise The factorial of a positive integer is that integer, multiplied by all positive integers that are lower (excluding zero). You write the factorial as the number with an exclamation mark after it. E.g., the factorial of 5 is $5! = 5 * 4 * 3 * 2 * 1 = 120$. Write some code

that calculates the factorial of a number. Do not test your program with numbers that are too high, as factorials grow exponentially (testing it up to 10! is more than enough). Hint: to do this with a **`while`** loop, you need two variables: one variable which at the end of the loop must contain the answer, and one variable that contains the current factor. In the loop, you multiply the answer with the current factor, before subtracting 1 from the factor. Choose the right initializations of these variables before the loop.

7.2 `for` loop

An alternative way of implementing loops is by using a **`for`** loop. **`for`** loops tends to be easier and safer to use than **`while`** loops, but cannot be applied to all iteration problems. **`while`** loops are more general. In other words, everything that a **`for`** loop can do, a **`while`** loop can do too, but not the other way around.

The syntax of a **`for`** loop is as follows:

```
for <variable> in <collection>:
    <statements>
```

A **`for`** loop gets presented with a collection of items, and it will process these items, in order, one by one. Every cycle through the loop will put one item in the variable given next to the **`for`**, and can then be used in the code block under the **`for`**. The variable does not need to exist before the **`for`** loop is encountered. If it does, it gets overwritten. It is a real variable, by the way, in the sense that it still exists after the loop has finished. It will contain the last value that it got assigned during the processing of the loop.

At this point you might wonder what a "collection" is. There are many different kinds of collections in Python, and in this section I will introduce a few. In later chapters collections will be discussed in more detail.

7.2.1 `for` loop with strings

The only collection introduced until now is the string. A string is a collection of characters, e.g., the string `"banana"` is a collection of the characters `"b"`, `"a"`, `"n"`, `"a"`, `"n"`, and `"a"`, in that specific order. The following code loops through each of these letters:

```
for letter in "banana":
    print( letter )
print( "Done" )
```

While this code is fairly trivial, let's go through it step by step (I did not make a flow chart, as that is not easy for **`for`** loops).

When the **`for`** loop is encountered, Python takes the collection (i.e., the string `"banana"`) and turns it into an "iterable." What that is exactly I will get to in Chapter 23, but for now assume that it is a list of all the letters in the string, in the order that they appear in the string. Python then takes the first of those letters, and puts it in the variable `letter`. It then executes the code block below the **`for`**.

The code block contains only one statement, which is the printing of `letter`. So the program prints "b," and then loops back to the **`for`**. Python then takes the next letter, which is

an "a," and it executes the code block with `letter` being `"a"`. It then repeats this process for each of the remaining letters. Once all the letters have been used, the **`for`** loop ends, and Python executes the last line of the code, which is the printing of the word "Done."

To be absolutely clear: In a **`for`** loop you do not have to write code that explicitly increases some kind of variable that then grabs the next letter, or something like that. The **`for`** statement handles that automatically: every time it is looped back to, it takes the next item from the collection.

7.2.2 `for` loop using a variable as collection

In the code above, the literal string `"banana"` was used as the collection, but it could also be a variable that contains a string. For instance, the following code runs similar to the previous code:

```
fruit = "banana"
for letter in fruit:
    print( letter )
print( "Done" )
```

You might wonder if this isn't dangerous. What happens if the programmer changes the contents of the variable `fruit` in the loop's code block? You can try this out using the following code:

listing0705.py

```
fruit = "banana"
for letter in fruit:
    print( letter )
    if letter == "n":
        fruit = "orange"
print( "Done" )
```

As you can see when you run this code, changing the contents of the variable `fruit` in the loop has no effect on the loop's processing. The sequence of characters that the loop processes is only constituted once, when the **`for`** loop is first entered. Changing the value of `fruit` into `"orange"` while the loop is still processing the value `"banana"`, does not stop it from continuing to process `"banana"`. This is a great feature of **`for`** loops, because it means they are guaranteed to end. No **`for`** loops are endless![2]

Note that there is a conditional statement in the loop above. There is nothing that stops you from putting conditions in the code block for a loop. There is also nothing against putting loops in the code block for a condition, or even putting loops inside loops (more on that last option follows later in this chapter). Most readers probably are not surprised to hear that, but for the few who are completely new to programming: as long as you stick to the syntactic requirements, you can use conditional statements and loops wherever you can write Python statements.

[2] Unfortunately, I will have to revise this statement in Chapter 23, but it requires knowledge of pretty advanced Python to create an endless `for` loop – for now, and in general practice, you may assume that `for` loops are guaranteed to end.

7.2.3 for loop using a range of numbers

Python offers a **range()** function that generates a collection of sequential numbers, which is often used for **for** loops. The simplest call to **range()** has one parameter, which is a number. It will generate all integers, starting at zero, up to but not including the parameter.

```
for x in range( 10 ):
    print( x )
```

range() can get multiple parameters. If you give two parameters, then the first will be the starting number (default is zero), while the second will be the "up to but not including" number. If you give three parameters, the third will be a step size (default is 1). You can choose a negative step size if you want to count down. With a negative step size, make sure that the starting number is higher than the number that you want to count up to (or down to, in this case).

```
for x in range( 1, 11, 2 ):
    print( x )
```

Exercise Change the three parameters above to observe their effect, until you fully understand the **range()** function.

Exercise Use the **for** loop and **range()** function to print multiples of 3, starting at 21, counting down to 3, in just two lines of code.

7.2.4 for loop with manual collections

If you want to use a **for** loop to cycle through items in a collection that you create manually, you can do so by listing all your items between parentheses. This defines a "tuple" for the items of your collection. Tuples will be discussed in detail in Chapter 11.

```
for x in ( 10, 100, 1000, 10000 ):
    print( x )
```

Or:

```
for x in ("apple", "pear", "orange", "banana", "mango", "cherry"):
    print( x )
```

Your collection can even consist of mixed types.

7.2.5 Practice with for loops

To get strong grips on how to use **for** loops, do the following exercises.

Exercise You already created code with a **while** loop that asked the user for five numbers, and displayed their total. Create code for this task, but now use a **for** loop.

Exercise Create a countdown function that starts at a certain count, and counts down to zero. Instead of zero, print "Blast off!" Use a **for** loop.

Exercise I am not going to ask you to build a **for** loop that asks the user to enter numbers until the user enters zero. Why not?

7.3 Loop control statements

There are three extra statements that help you control the flow in a loop. They are **else**, **break**, and **continue**. They work with both **while** and **for** loops.

7.3.1 else

Just like with an **if** statement, you can add an **else** statement to the end of a **while** or **for** loop. The code block for the **else** is executed whenever the loop ends, i.e., when the boolean expression for the **while** loop evaluates to **False**, or when the last item of the collection of the **for** loop is processed. Here is an example for a **while** loop:

listing0706.py

```
i = 0
while i < 5:
    print( i )
    i += 1
else:
    print( "The loop ends, i is now", i )
print( "Done" )
```

This code is equivalent to the flow chart in Figure 7.2. When you look at the flow chart you might think it does not make much sense to use an **else**, but it can be powerful when combined with a **break** (which follows next).

Here is an example of using **else** for a **for** loop:

listing0707.py

```
for fruit in ( "apple", "orange", "strawberry" ):
    print( fruit )
else:
    print( "The loop ends, fruit is now", fruit )
print( "Done" )
```

Notice that after the **while** loop above, the value of `i` is 5. The value of `fruit` after the **for** loop above is the last item that it encountered, i.e., `"strawberry"`.

7.3.2 break

The **break** statement allows you to break out of a loop prematurely. I.e., when Python encounters the **break** statement, it will no longer process the remainder of the code block

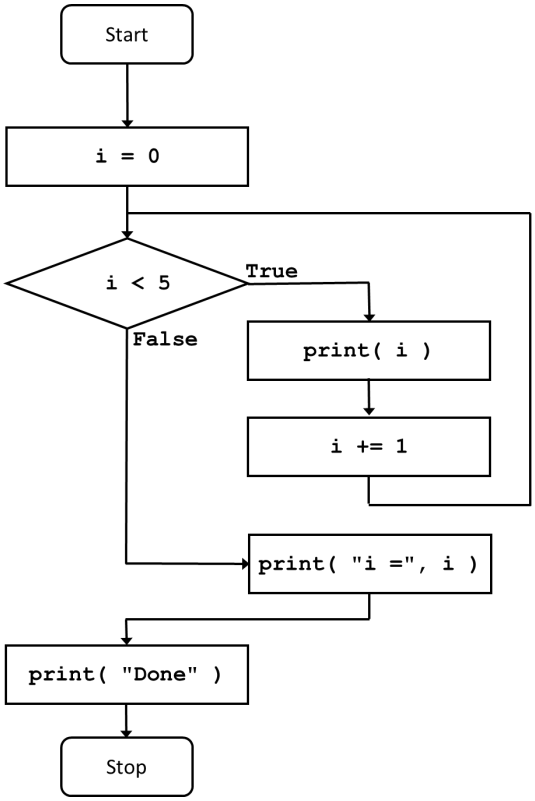


Figure 7.2: Flow chart expressing a `while` loop with an `else` branch.

for the loop, and will not loop back to the loop's first line. It will simply continue with the first statement after the loop's code block.

To see why this is useful, here follows an interesting exercise. I am looking for a number that starts with a 1, and when you transfer that 1 to the end of the number, the result is a number that is three times as high. For example, if I check the number 1867, I move the 1 from the start to the end, which gives 8671. If 8671 would be three times 1867, that is the answer I seek. It is not, so 1867 is not correct. The code to solve this is actually fairly short, and gives the lowest number for which the comparison holds:

listing0708.py

```
i = 1
while i <= 1000000:
    num1 = int( "1" + str( i ) )
    num2 = int( str( i ) + "1" )
    if num2 == 3 * num1:
        print( num2, "is three times", num1 )
        break
    i += 1
else:
    print( "No answer found" )
```

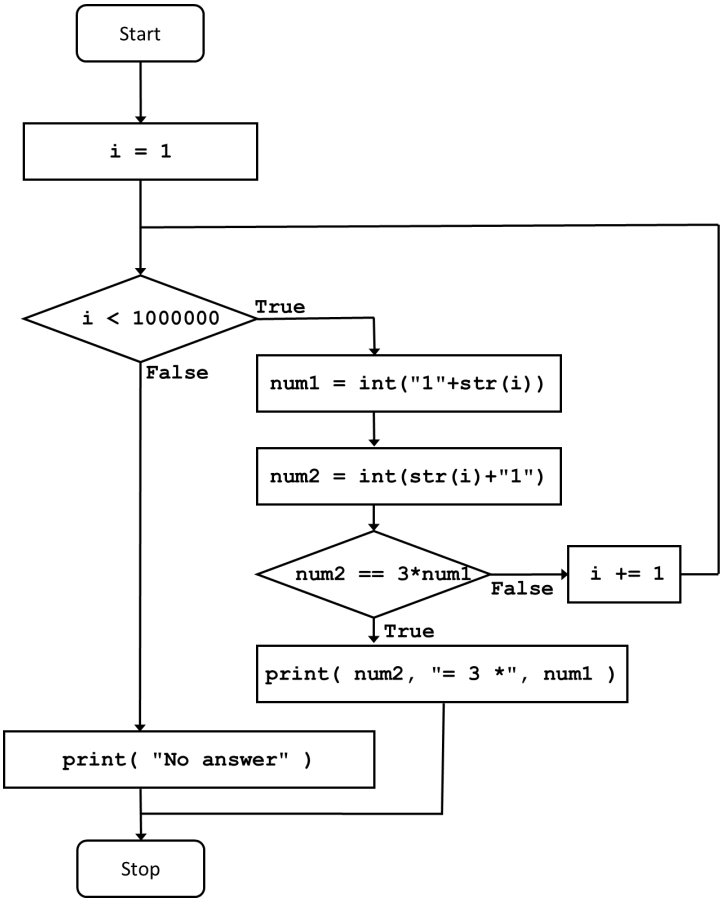


Figure 7.3: Flow chart expressing a `while` loop containing a `break`.

This code is expressed by the flow chart in Figure 7.3.

In this example you see the **`break`** statement used to good effect. Since I have no idea which number I am looking for, I am just going to check a whole bunch of numbers. I let a counter `i` run up to 1000000. Of course, I don't know if I find the answer before `i` has reached 1000000, but I should set a limit somewhere, because I don't know if a number with the requested property exists at all, and I do not want to create an endless loop. I might find the answer at any point, and when I do, I break out of the loop, because further testing of numbers no longer serves a purpose.

The point here is that the setting of the maximum value of `i` to 1000000 is not because I know that I have to generate one million numbers. I have no idea how many times I have to cycle through the loop. I just know that if I encounter the requested number at some point, I am done and can skip the remainder of the cycles. That is exactly what the purpose of the **`break`** is.

With some juggling the boolean expression for the loop actually can do the comparison for me. It would be something like **`while`** `(i < 1000000)` **`and`** `(num1 != 3*num2):`. This becomes a bit convoluted, and I would also have to give `num1` and `num2` starting values before the loop starts. Still, it is always possible to avoid using a **`break`**, but applying the

break might make code more readable and flow better, as it does in this case.

The **break** statement cannot be used outside a loop. It is only defined for loops. (Take note of this. I very often see students putting **break** statements in conditions that are not inside a loop, and then look mystified when Python reports a runtime error.)

Note that when a **break** statement is encountered, and the loop also has an **else** clause, the code block for the **else** will *not* be executed. I use this to good effect in the code above, by ensuring that the text that indicates that no answer is found, only will be printed if the loop ends by checking all the numbers without finding an answer.

The following code checks a list of grades for a student. As long as all grades are 5.5 or higher, the student passes. When one or more grades are lower than 5.5, the student fails. The grades are in a collection that is given to a **for** loop.

listing0709.py

```
for grade in ( 8, 7.5, 9, 6, 6, 6, 5.5, 7, 5, 8, 7, 7.5 ):
    if grade < 5.5:
        print( "The student fails!" )
        break
else:
    print( "The student passes!" )
```

Exercise Run the code above and notice that the student fails. Then remove the 5 from the list of grades and notice that the student now passes.

7.3.3 continue

When the **continue** statement is encountered in the code block of a loop, the current cycle ends immediately and the code loops back to the start of the loop. For a **while** loop, that means that the boolean expression is evaluated again. For a **for** loop, that means that the next item is taken from the collection and processed.

The following code prints all numbers between 1 and 100 that cannot be divided by 2 or 3, and do not end in a 7 or 9.

listing0710.py

```
num = 0
while num < 100:
    num += 1
    if num%2 == 0:
        continue
    if num%3 == 0:
        continue
    if num%10 == 7:
        continue
    if num%10 == 9:
        continue
    print( num )
```

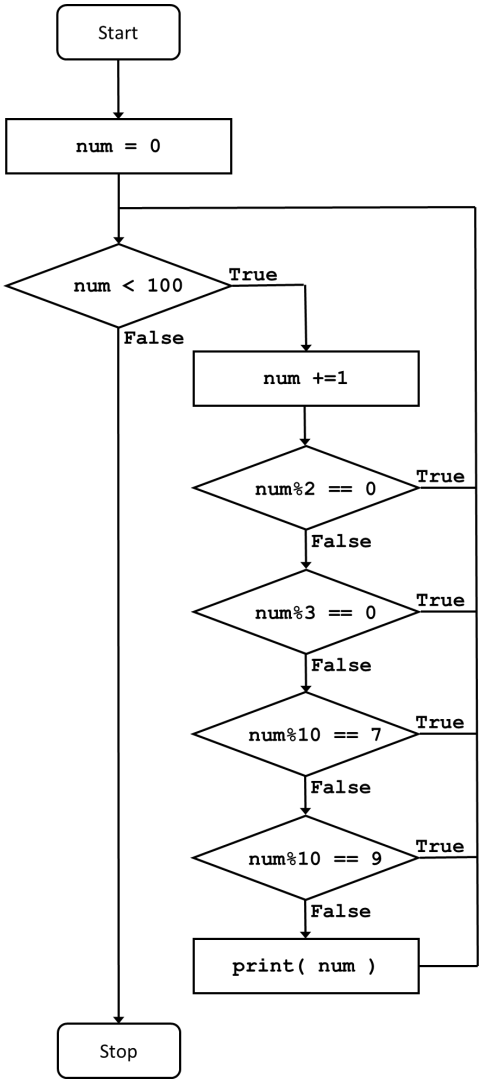


Figure 7.4: Flow chart expressing a `while` loop containing several `continues`.

This code is also expressed by the flow chart in Figure 7.4.

I don't know why you would want this list, but the use of **`continue`** statements to implement it helps. Alternatively, you could have created one big boolean expression for an **`if`** statement, but that would become unreadable quickly. Still, just like **`break`** statements, **`continue`** statements can always be avoided if you really want to, but they do help keeping code understandable.

Note that **`continue`** statements, just like **`break`** statements, can only be used inside loops.

Be very, very careful when using a **`continue`** in a **`while`** loop. Most **`while`** loops use a number that restricts the number of cycles through the loop. Usually such a number is increased at the bottom of the code block for the loop. A **`continue`** statement would loop back to the boolean expression immediately, without increasing the number, and thus such a **`continue`** could easily cause an endless loop. I.e.:

```
i = 0
while i < 10:
    if i == 5:
        continue
    i += 1
```

causes an endless loop!

Exercise Write a program that processes a collection of numbers using a **for** loop. The program should end immediately, printing only the word "Done," when a zero is encountered (use a **break** for this). Negative numbers should be ignored (use a **continue** for this; I know you can also do this with a condition, but I want you to practice with **continue**). If no zero is encountered, the program should display the sum of all numbers (do this in an **else** clause). Always display "Done" at the end of the program. Test your program with the collection `( 12, 4, 3, 33, -2, -5, 7, 0, 22, 4 )`. With these numbers, the program should display only "Done." If you remove the zero, it should display 85 (and "Done").

7.4 Nested loops

You can put a loop inside another loop.

That is a simple statement, but it is one of the hardest concepts for students to wrap their minds around.

Let's first look at an example of a double-nested loop, i.e., a loop which contains one other loop. Usually programmers talk about an "outer loop" and an "inner loop." The inner loop is part of the code block for the outer loop.

listing0711.py

```
for i in range( 3 ):
    print( "Entering the outer loop for i =", i )
    for j in range( 3 ):
        print( "    Entering the inner loop for j =", j )
        print( "    (i,j) = ({},{})".format( i, j ) )
        print( "    Leaving the inner loop for j =", j )
    print( "Leaving the outer loop for i =", i )
```

Study this code and its output until you fully understand it!

The code first gives `i` the value 0, and then lets `j` take on the values 0, 1, and 2. It then gives `i` the value 1, and then lets `j` take on the values 0, 1, and 2. Finally, it gives `i` the value 2, and then lets `j` take on the values 0, 1, and 2. So this code runs through all possible pairs of `(i,j)` with `i` and `j` being 0, 1, or 2.

Notice how variables for the outer loop are also accessible by the inner loop. `i` exists in both the outer and the inner loop.

Suppose that you want to print all pairs `(i,j)` where `i` and `j` can take on the values 0 to 3, but `j` must be higher than `i`. Code that does that is:

```
for i in range( 4 ):
    for j in range( i+1, 4 ):
        print( "({},{})".format( i, j ) )
```

See how I cleverly use `i` to set the range for `j`?

Exercise Write code that prints all pairs `(i,j)` where `i` and `j` can take on the values 0 to 3, but they cannot be equal.

You can, of course, also nest **`while`** loops, or mix nesting **`for`** loops with **`while`** loops.

You should be aware that when you use a **`break`** or **`continue`** in an inner loop, it will only break out of the inner loop or continue with the inner loop, respectively. There is no command that you can give in an inner loop that breaks out of both the inner and outer loop immediately.[3]

Once you understand double-nested loops, it should come as no surpise that you can also triple-nest loops, quadruple-nest loops, or go even deeper. However, in practice I have seldom seen a nesting deeper than triple-nested.

```
for i in range( 3 ):
    for j in range( 3 ):
        for k in range( 3 ):
            print( "({},{},{})".format( i, j, k ) )
```

7.5 The loop-and-a-half

Suppose you want to ask the user for two numbers in a loop. For every two numbers that the user enters, you want to show their multiplication. You allow the user to stop the program when he enters zero for any of the numbers. For some reason, if the numbers are dividers of each other, that is an error and the program also stops, but with an error message. Finally, you will not process numbers higher than 1000 or smaller than zero, but that is not an error; you just want to allow the user to enter new numbers. How do you program that? Here is a first attempt:

listing0712.py

```
from pcinput import getInteger

x = 3
y = 7

while (x != 0) and (y != 0) and (x%y != 0) and (y%x != 0):
    x = getInteger( "Enter number 1: " )
    y = getInteger( "Enter number 2: " )
    if (x > 1000) or (y > 1000) or (x < 0) or (y < 0):
```

[3] Unless you use them in a function, then you can exit the function at any time and so interrupt both the inner and the outer loop. But that will follow in Chapter 8.

```
        print( "Numbers should both be between 0 and 1000" )
        continue
    print( "Multiplication of", x, "and", y, "gives", x * y )

if x == 0 or y == 0:
    print( "Goodbye!" )
else:
    print( "Error: the numbers cannot be dividers" )
```

Exercise Study this code and make a list of everything that you feel is bad about it. Once you have done that, continue reading and compare your notes to the list below. If you noted things that are bad about it that are not on the list below, email the author of the book.

There are many things bad about this code. Here is a list:

- To ensure that the loop is run at least once, `x` and `y` must be initialized. Why did I choose 3 and 7 for that? That was arbitrary, but I had to pick two numbers that are not dividers of each other. Otherwise the loop would not have been entered. On the whole, having to give variables some arbitrary starting values just to make sure that they exist is not nice, as their initial values are meaningless. You want to avoid that.
- When you enter something that should end the loop (e.g., zero for `x`), the multiplication will still take place before the loop ends. That was not supposed to happen.
- If you enter 0 for `x`, the code will still ask for a value for `y`, even if it does not need it anymore.
- The boolean expression next to the **`while`** is rather complex. In this code it is still readable, but you can imagine what happens when you have many more requirements.
- The error message for the dividers is not next to the actual test where you decide to leave the loop (i.e., the boolean expression next to the **`while`**).

The solution to some of these issues that certain programmers prefer, is to initialize `x` and `y` with values that you read from the input. This solves the arbitrary initialization, and also gets around the problem that you print the result of the multiplication even when the loop was already supposed to end. If you do this, however, in the loop you have to move the asking for input to the end of the loop, and if you ever have a **`continue`** in the loop, you also have to copy it there. The code becomes something like this:

listing0713.py

```
from pcinput import getInteger

x = getInteger( "Enter number 1: " )
y = getInteger( "Enter number 2: " )

while (x != 0) and (y != 0) and (x%y != 0) and (y%x != 0):
    if (x > 1000) or (y > 1000) or (x < 0) or (y < 0):
        print( "Numbers should both be between 0 and 1000" )
        x = getInteger( "Enter number 1: " )
```

```
        y = getInteger( "Enter number 2: " )
        continue
    print( "Multiplication of", x, "and", y, "gives", x * y )
    x = getInteger( "Enter number 1: " )
    y = getInteger( "Enter number 2: " )

if x == 0 or y == 0:
    print( "Goodbye!" )
else:
    print( "Error: the numbers cannot be dividers" )
```

So this code removes two of the issues, but it adds a new one, which makes the code a lot worse. The list of issues now is:

- The statements that ask for input for each of the variables occur no less than three times in the code.
- If you enter 0 for `x`, the code will still ask for a value for `y`.
- The boolean expression next to the **`while`** is rather complex.
- The error message for the dividers is not next to the actual test where you decide to leave the loop.

The trick to get around these issues is to control the loop solely through **`continue`**s and **`break`**s (and perhaps the occasional `exit()` when errors occur, though later in the course you will learn to use the much "cleaner" **`return`** for that). I.e., you do the loop "always," but decide to leave the loop or redo the loop when certain events occur which you notice inside the loop. Doing the loop "always" you can effectuate with the statement **`while True`** (as this simply means: the test that decides whether or not you have to do the loop again, always results in **`True`**).

listing0714.py

```
from pcinput import getInteger
from sys import exit

while True:
    x = getInteger( "Enter number 1: " )
    if x == 0:
        break
    y = getInteger( "Enter number 2: " )
    if y == 0:
        break
    if (x < 0 or x > 1000) or (y < 0 or y > 1000):
        print( "The numbers should be between 0 and 1000" )
        continue
    if x%y == 0 or y%x == 0:
        print( "Error: the numbers cannot be dividers" )
        exit()
    print( "Multiplication of", x, "and", y, "gives", x * y )

print( "Goodbye!" )
```

This code gets around almost all the problems. It asks for the input for `x` and `y` only once. There is no arbitrary initialization for `x` and `y`. The loop stops as soon as you enter zero for one of the numbers. It prints error messages at the moment that the errors are noted. There is no complex boolean expression needed with lots of **`and`**s and **`or`**s.

The only issue that is still remaining is that when the user enters a value outside the range 0 to 1000 for `x`, he still gets to enter `y` before the program says that he has to enter the numbers again. That is best solved by writing your own functions, which follows in the next chapter. (If you really want to solve it now, you can do that with a nested loop, but I wouldn't bother.)

The code is slightly longer than the first version, but length is no issue, and the code is a lot more readable.

A loop like this one, that uses **`while True`**, is sometimes called a "loop-and-a-half." It is a common approach to writing loops for which you cannot predict when they will end.

Exercise The user must enter a positive integer. You use the `getInteger()` function from `pcinput` for that. This function also allows entering negative numbers. If the user enters a negative number, you want to print a message and ask him again, until he entered a positive number. Once a positive number is entered, you print that number and the program ends. Such a problem is typically solved using a loop-and-a-half, as you cannot predict how often the user will enter a negative number before he gets wise. Write such a loop-and-a-half (you will need exactly one **`break`**, and you need at most one **`continue`**). Print the final number that the user entered after you have exited the loop. The reason to do it afterwards is that the loop is just there to control the entering of the input, not the processing of the resulting variable.

I have noted in the past that many students find the use of **`while True`** confusing. They see it often in example code, but do not really grasp what the point of it is. And then they start inserting **`while True`** in their code whenever they do not know exactly what they need to do. If you have troubles understanding the loop-and-a-half, study this section again, until you do.

7.6 Being smart about loops

To complete this chapter, I want to discuss a few strategies on loop design, and, in general, the design of algorithms.

7.6.1 When to use a loop

If you roll five 6-sided dice, how big is the probability that you roll five sixes? The answer is $1/(6^5)$, but suppose that you did not know that, and wanted to use a simulation to estimate the probability. You can imitate the rolling of a die using `randint()`, and so you can imitate the rolling of five dice this way. You can check whether they all show a 6. You can do that a large number of times, and then divide the number of times that you rolled five sixes by the number of times that you rolled five dice, to get an estimate. When I put this problem to students (in a slightly more complicated form, so that the answer cannot easily be calculated), I often get code that looks like this:

listing0715.py

```
from random import randint

TESTS = 10000

success = 0
for i in range( TESTS ):
    d1 = randint( 1, 6 )
    d2 = randint( 1, 6 )
    d3 = randint( 1, 6 )
    d4 = randint( 1, 6 )
    d5 = randint( 1, 6 )
    if d1 == 6 and d2 == 6 and d3 == 6 and d4 == 6 and d5 == 6:
        success += 1

print( "Chance at five sixes is", success / TESTS )
```

(You would need a bigger number of tests to get a more accurate estimate, but I did not want this code to run too long.)

When I see code like this, I ask: "What if I had asked you to roll one hundred dice? Would you really repeat that die rolling line 100 times?" Whenever you see lines of code repeated with just a slight change between them (or when you are copying/pasting within a block of code), you should start thinking about loops. You can roll five dice by stating:

```
from random import randint

for i in range( 5 ):
    die = randint( 1, 6 )
```

"But," you might argue: "I need the value of all the five dice to see if they are all sixes! Every time you cycle through the loop, you lose the value of the previous roll!"

True enough, but the line that checks all the dice by using five boolean expressions concatenated with **and**s is particularly ugly too. Can't you streamline this? Is there no way that you can draw some conclusion upon the rolling of one die?

By thinking a bit about it, you might come to the following conclusion: as soon as you roll a die that is not a six, you already have failed on your try, and you can skip to the next try. There are many ways to effectuate this, but here is a brief one using a **break** and an **else**:

listing0716.py

```
from random import randint

TESTS = 10000

success = 0
for i in range( TESTS ):
    for j in range( 5 ):
        if randint( 1, 6 ) != 6:
```

```
            break
    else:
        success += 1

print( "Chance at five sixes is", success / TESTS )
```

You might think this is difficult to come up with, but there are other ways too. You can, for instance, add up the values of the rolls and test if the total is 30 after the inner loop. Or you can keep track of how many dice were rolled to a value of 6 and check if that is 5 after the inner loop. Or you can set a boolean variable to **`True`** before the inner loop, then set it to **`False`** as soon as you roll something that is not 6 in the inner loop, and then test the variable after the inner loop.

The point is that the arbitrary long repetition of pieces of code can probably be replaced by a loop.

7.6.2 Processing data items one by one

Usually, when a loop is applied, you are working through a long series of data items. Each cycle through the loop will process one of those data items. You then often need to remember something about the data items that you have processed so far, for which you need extra variables. You have to be smart in thinking about such variables.

Take the following example: I will provide you with ten numbers, and I need you to create a program that tells me which is the largest, which is the smallest, and how many are divisible by 3. You might say: "It is easy to determine the largest and the smallest; I just use the **`max`**`()` and **`min`**`()` functions (introduced in Chapter 5). Divisible by 3 is a bit tricky, I have to think about that." But **`max`**`()` and **`min`**`()` require you to keep all the numbers in memory. That's fine for 10 numbers, but what about one hundred? Or a million?

Since you will have to process all the numbers, you have to think about a loop, and in particular, a loop wherein you have only one of the numbers available each cycle through the loop (but you will see them all before the loop ends). You must now think about variables that you can use to remember something each cycle through the loop, that allows you to determine, at the end, which number was the largest, which the smallest, and how many are divisible by 3. Which variables do you need?

The answer, which comes easy to anyone who has been doing some programming, is that you need to remember, each cycle through the loop, which is the largest number *until now*, which is the smallest number *until now*, and how many numbers are divisible by 3 *until now*. That means that every cycle through the loop you compare the new number with the variables in which you retain the largest and smallest, and replace them with the new number if that is appropriate. You also check if the new number is divisible by three, and if so, increase the variable that you use to keep track of that.

You will have to find good initial values for the three variables. The divisible-by-3 variable can start at zero, but the largest and smallest need an appropriate value. The best solution in this case is to fill them with the first number, as that number is both the largest and the smallest at that point.

I give this problem as an exercise below. Use the algorithm described here to solve it.

7.6.3 Start with the smallest unit and build outward

Suppose that I give you the following assignment: Of all the books on all the shelves in the library, count the number of words and report the average number of words for the books. If you ask a human to perform this task, he or she will probably think: "I go to the library, get the first book from the first shelf, count the words, write that number down, then take the second book and do the same thing, etcetera. When I finished the first shelf, I go to the second shelf and treat that one in the same way, until I have done all the books on all the shelves in the library. Then I add up the counts and divide by the number of books." For humans this approach works, but when you need to tell a computer how to do this, the task seems hard.

To solve this problem, I should start with the smallest unit that I need to work with. In this case, the smallest unit is a "book." It is not "word," because I don't need to do anything with a "word"; what I need to do is count the words in a book. In pseudocode,[4] that would be something like:

```
wordcount = 0
for word in book:
    wordcount += 1
```

When I code something like this, I can already test it. Once I am satisfied that I can count the words in a book, I can move up to the next smallest unit, which is the shelf. How do I process all the books on a shelf? In pseudocode, it is something like:

```
for book in shelf:
    process_book()
```

But what does `process_book()` do? It counts the words. I already wrote pseudocode for that, which I simply need to insert in place of the statement `process_book()`. This then becomes:

```
for book in shelf:
    wordcount = 0
    for word in book:
        wordcount += 1
```

When I test this, I run into a problem. I find that I am counting the words per book, but I am not doing anything with those word counts. I just overwrite them. To get the average, I first need a count of all the words in all the books. This means I have to initialize `wordcount` only once.

```
wordcount = 0
for book in shelf:
    for word in book:
        wordcount += 1
```

To calculate the average, I need to also count the books. Again, I only need to initialize the `bookcount` once, at the start, and I have to increase the `bookcount` every time I have processed one book. At the end, I can then print the average.

```
wordcount = 0
bookcount = 0
for book in shelf:
    for word in book:
```

[4] Pseudocode isn't real code, but it looks like code that more or less explains how an algorithm works. As such, it should be easy to implement it in various programming languages.

```
            wordcount += 1
        bookcount += 1
    print( wordcount / bookcount )
```

Finally, I can go up to the highest level: the library as a whole. I know how to process one shelf, now I need to process all the shelves. I should, of course, remember that the initialization of wordcount and bookcount only should be done once, and the printing of the average too. With that in mind, it is easy to extend the pseudocode to encompass the library as a whole:

```
wordcount = 0
bookcount = 0
for shelf in library:
    for book in shelf:
        for word in book:
            wordcount += 1
    bookcount += 1
print( wordcount / bookcount )
```

As you can see, I built a triple-nested loop, working from the inner loop outward. To learn how to deal with nested loops, this is usually the best approach.

7.7 On designing algorithms

At this point in the book, you will often run into exercises and coding problems for which you are unsure how to solve them. I gave an example of such a problem above (finding of ten numbers the largest, the smallest, and the number divisible by 3), and the solution I came to. Such a solution approach is called an "algorithm." But how do you design such algorithms?

I often see students typing code without really knowing what they are doing. They are trying to solve a problem but do not know how, so they start typing. You may realize that this is not a good approach to creating solutions (even though experimenting a bit might help).

What you have to do in such a situation is sit back, leave the keyboard alone, and think "How would I solve this problem as a human?" Try to write down what you would do if you would do it by hand. It does not matter if what you would do is a very boring task that you would never want to do by hand – you have a computer to do the boring things for you.

Once you have figured out what you would do, then try to think about how you would translate that to code. Because basically, that is what you need to tell the computer: the steps that you as a human would take to get to a solution. If you really cannot think of any way that you as a human would use to solve a problem, then you sure as hell won't be able to tell the computer how to do it for you.

What you learned

In this chapter, you learned about:

- What loops are
- **`while`** loops
- **`for`** loops
- Endless loops
- Loop control via **`else`**, **`break`**, and **`continue`**
- Nested loops
- The loop-and-a-half
- Being smart about loops

Exercises

Since loops are incredibly important and students often have problems with them, I provide a considerable number of exercises here. I recommend that you do them all. You will learn a lot.

Exercise 7.1 Write a program that lets the user enter a number. Then the program displays the multiplication table for that number from 1 to 10. E.g., when the user enters 12, the first line printed is "`1 * 12 = 12`" and the last line printed is "`10 * 12 = 120`".

Exercise 7.2 If you did the previous exercise with a **`while`** loop, then do it again with a **`for`** loop. If you did it with a **`for`** loop, then do it again with a **`while`** loop. If you did not use a loop at all, you should be ashamed of yourself.

Exercise 7.3 Write a program that asks the user for ten numbers, and then prints the largest, the smallest, and how many are divisible by 3. Use the algorithm described earlier in this chapter.

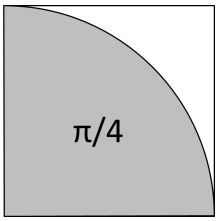
Exercise 7.4 "99 bottles of beer" is a traditional song in the United States and Canada. It is popular to sing on long trips, as it has a very repetitive format which is easy to memorize, and can take a long time to sing. The song's simple lyrics are as follows: "99 bottles of beer on the wall, 99 bottles of beer. Take one down, pass it around, 98 bottles of beer on the wall." The same verse is repeated, each time with one fewer bottle. The song is completed when the singer or singers reach zero. Write a program that generates all the verses of the song (though you might start a bit lower, for instance with 10 bottles). Make sure that your loop is not endless, and that you use the proper inflection for the word "bottle."

Exercise 7.5 The Fibonacci sequence is a sequence of numbers that starts with 1, followed by 1 again. Every next number is the sum of the two previous numbers. I.e., the sequence starts with 1, 1, 2, 3, 5, 8, 13, 21,... Write a program that calculates and prints the Fibonacci sequence until the numbers get higher than 1000.

Exercise 7.6 Write a program that asks the user for two words. Then print all the characters that the words have in common. You can consider capitals different from lower case letters, but each character that you report, should be reported only once (e.g., the strings "bee" and "peer" only have one character in common, namely the letter "e"). Hint: Gather the characters in a third string, and when you find a character that the two words have in common, check if it is already in the third string before reporting it.

Exercise 7.7 Write a program that approximates π by using random numbers, as follows. Consider a square measuring 1 by 1. If you throw a dart into that square in a random location, the probability that it will have a distance of 1 or less to the lower left corner is $\pi/4$. To see why that is, remember that the area of a circle with a radius of 1 is π, so the area of a quarter circle is $\pi/4$. Thus, if a dart lands in a random point in the square, the chance that it lands in the quarter circle with its centre at the lower left corner is $\pi/4$. Therefore, if you throw N darts into the square, and M of those land inside a distance of 1 to the lower left corner, then $4M/N$ approximates π if N is very large.

The program holds a constant that determines how many darts it will simulate. It prints an approximation of π derived by simulating the throwing of that number of darts. Remember that the distance of a point (x, y) to the lower-left corner is calculated as $sqrt(x^2 + y^2)$. Use the `random()` function from the `random` module.



Exercise 7.8 Write a program that takes a random integer between 1 and 1000 (you can use the `randint()` function for that). The program then asks the user to guess the number. After every guess, the program will say either "Lower" if the number it took is lower, "Higher" if the number it took is higher, and "You guessed it!" if the number it took is equal to the number that the user entered. It will end with displaying how many guesses the user needed. It might be wise, for testing purposes, to also display the number that the program randomly picks, until you are sure that the program works correctly.

Exercise 7.9 Write a program that is the opposite of the previous one: now you take a number in mind, and the computer will try to guess it. You respond to the computer's guesses by entering a letter: "L" for lower if the number to guess is lower, "H" for higher if the number to guess is higher, and "C" for correct (you can use the `getLetter()` function from `pcinput` for that). Once the computer has guessed your number, it displays how many guesses it needed. Make sure that you let the computer recognize when there is no answer (maybe because you made a mistake or because you tried to fool the computer). A smart program will need at most ten guesses.

Exercise 7.10 A prime number is a positive integer that is divisible by exactly two different numbers, namely 1 and itself. The lowest (and only even) prime number is 2. The first 10 prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29. Write a program that asks the

user for a number and then displays whether or not it is prime. Hint: In a loop where you test the possible dividers of the number, you can conclude that the number is not prime as soon as you encounter a number other than 1 or itself that divides it. However, you can only conclude that it actually is prime after you have tested all possible dividers.

Exercise 7.11 Write a program that prints a multiplication table for digits 1 to a certain number `num` (you may assume for the output that `num` is one digit). A multiplication table for the numbers 1 to `num = 3` looks as follows:

```
. |  1  2  3
-----------
1 |  1  2  3
2 |  2  4  6
3 |  3  6  9
```

So the labels on the rows are multiplied by the labels on the columns, and the result is shown in the cell that is on that row/column combination.

Exercise 7.12 Write a program that displays all integers between 1 and 100 that can be written as the sum of two squares. Produce output in the form of `z = x**2 + y**2`, e.g., `58 = 3**2 + 7**2`. If a number occurs on the list with multiple different ways of writing it as the sum of two squares, that is acceptable (this may be the case for 50, 65, and 85).

Exercise 7.13 You roll five six-sided dice, one by one. How big is the probability that the value of each die is greater than or equal to the value of the previous die that you rolled? For example, the sequence "1, 1, 4, 4, 6" is a success, but "1, 1, 4, 3, 6" is not. Determine the probability of success using a simulation of a large number of trials.

Exercise 7.14 A, B, C, and D are all different digits. The number DCBA is equal to 4 times the number ABCD. What are the digits? Note: to make ABCD and DCBA conventional numbers, neither A nor D can be zero. Use a quadruple-nested loop.

Exercise 7.15 According to an old puzzle, five pirates and their monkey are stranded on an island. During the day they gather coconuts, which they put in a big pile. When night falls, they go asleep.

In the middle of the night, the first pirate wakes up, and, not trusting his buddies, he divides the pile into five equal parts, takes what he believes to be his share and hides it. Since he had one coconut left after the division, he gives it to the monkey. Then he goes back to sleep.

An hour later, the next pirate wakes up. He behaves in the same way as the first pirate: he divides the pile into five equal shares, with one coconut left over which he gives to the monkey, hides what he believes to be his share, and goes to sleep again.

The same happens to the other pirates: they wake up one by one, divide the pile, give one coconut to the monkey, hide their share, and go back to sleep.

In the morning they all wake up. They divide what remains of the coconuts equally amongst them. Since that leaves one coconut, they give it to the monkey.

The question is: what is the smallest number of coconuts that they can have started with?

Write a Python program that solves this puzzle. If you can solve it for any number of pirates, all the better.

Exercise 7.16 Consider the triangle shown below. This triangle houses a colony of Triangle Crawlers, and one big Eater of Triangle Crawlers. The Eater is located in point D. All Triangle Crawlers are born in point A. A Triangle Crawler which ends up in point D gets eaten.

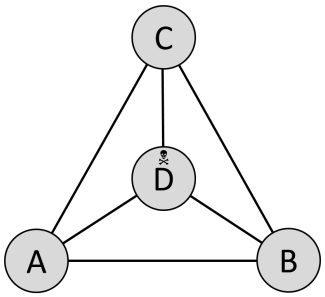
Every day, each Triangle Crawler moves over one of the lines to a randomly-determined neighboring point, but not to the point where he was the day before. This movement takes one day. For instance, a Triangle Crawler that was just born in A, on the first day of his life will move to B, C, or D. If he moves to B, the next day he will move to C or D (but not back to A). If on his first day he moves to C instead, the next day he will move to B or D (but not back to A). If he moves to D, he gets eaten.

There is a one-third probability that Triangle Crawler on the first day of his life immediately goes to D, and therefore only lives one day. In principle, a Triangle Crawler may reach any age, however high, by moving in circles from A to B to C and back to A again (or counterclockwise, from A to C to B and back to A again). However, since every day he makes a random choice between the two possible follow-up directions, every day after the first there is a one-half probability that he ends up in point D, and dies.

Write a program that calculates an approximation of the average age that a Triangle Crawler reaches. Do this by simulating the lives of 100,000 Triangle Crawlers, counting the days that they live, and dividing the total by 100,000. The output of your program should be a single floating point number, rounded to two decimals.

Hint 1: You can follow two different approaches: either you simulate the behavior of one single Triangle Crawler and repeat that 100,000 times, or you start with a population of 100,000 triangle crawlers in point A, and divide these over variables that keep track of how many Triangles are in each point, each day, including the point that they came from (assigning a remaining odd Triangle Crawler to a randomly determined neighboring point). The first method is short and simple but slow, the second is long and complex but fast. You may use either method.

Hint 2: Do not use 100,000 Triangle Crawlers in your first attempts. Start with 1000 (or even only 1), and only try it out with 100,000 once your program is more or less finished. Testing is much quicker with fewer Triangle Crawlers. 1000 Triangle Crawlers should be



done in under a second, so if your program takes longer, you probably have created an endless loop.

Hint 3: I won't be too specific, but the final answer is somewhere between 1 and 5 days. If you get something outside that range, it is definitely wrong. You may try to determine the exact answer mathematically before starting on the exercise, which is doable though quite hard.

Chapter 8

Functions

In Chapter 5 I described how to use simple functions, and how to import functions from modules. This chapter is about how to write your own functions and modules. If you do not remember what Chapter 5 said about functions, re-read that chapter before continuing with this one.

8.1 Why create functions?

Why would you want to create a function? There are several different reasons why you want to have a function:

- You may need a particular functionality for your code that you want to develop in independence of the rest of the code. If you put such a functionality in a function, that means that after developing and testing the functionality you can use it without thinking about it anymore.
- You may need a particular functionality that returns in different places in your code, and rather than copy it to all these places, you write a function for it which you call in all these places.
- You may need a particular functionality in your code that you need to control using parameters. If you put it in a function, the parameters become clearer and the code becomes more readable and easy to maintain.
- Your program may just be getting too long to keep a solid grasp on its contents, and you feel you can improve readability and maintainability by splitting off inherently connected blocks into functions.
- You may have problems solving a big problem in one go, and decide to divide it into sub-problems (which is usually a good idea). You can now create a function for each of these sub-problems, and by connecting them together, solve the big problem.
- Your program may contain deeply nested conditions or loops, and would benefit enormously as far as readability is concerned by moving some of the deeper nestings into functions.
- You may want to re-use code in different programs, and functions are a good way to transfer code between programs.

- You may want to release some of your code to other programmers, and functions are, again, a good way to do that.

In general, the advantage of functions is that they provide a means to effectuate:

- *Encapsulation*: Wrapping up a piece of useful code in such a way that it can be used without knowledge of the specifics
- *Generalization*: Making a piece of code useful for a variety of circumstances by controlling it via parameters
- *Manageability*: Dividing a complex program into easy-to-manage chunks
- *Maintainability*: Using meaningful names and logical wrappings to make a program better readable and understandable
- *Reusability*: Facilitating the transfer of functionalities between programs
- *Recursion*: Allowing the use of a technique called "recursion," which is the topic of Chapter 9.

8.2 Creating functions

Chapter 5 described how each function has a name, may have some parameters, and may have a return value. When you create your own functions, you need to define each of these. To create a function, you use the following syntax:

```
def <function_name>( <parameter_list> ):
    <statements>
```

The function name must meet the same requirements that variable names must meet, i.e., only letters, digits, and underscores, and it cannot start with a digit. The parameter list consists of zero or more variable names, with commas in between. The code block below the function definition must be indented.

Finally, be aware that Python must have "seen" your function definition before it sees the call to it in your code. Therefore it is convention to place all function definitions at the top of a program, right under the **`import`** statements.

8.2.1 How Python deals with functions

To be able to create functions, you have to know how Python deals with functions. Look at the small Python program below. It defines one function, called `goodbyeWorld()`. That function has no parameters. The code block for the function prints the line "Goodbye, world!"

The rest of the program is not part of a function. We often call the parts of a program that are not inside a function the "main" program. The main program prints the line "Hello, world!," and then calls the function `goodbyeWorld()`.

```
def goodbyeWorld():
    print( "Goodbye, world!" )

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

When you run this program, you see that it first prints "Hello, world!," and then "Goodbye, world!" This happens even though Python processes code top-down, so that it sees the line `print( "Goodbye, world!" )` before it sees the line `print( "Hello, world!" )`. This is because Python does not actually run the code inside functions, at least, not until the moment that the function gets called. Python does not even look at the code in functions. It just notices the function name, registers that the function is defined so that it can be used, and then continues, searching for the main program to run.

8.2.2 Parameters and arguments

Examine the code below. It defines a function `hello()` with one parameter, which is called `name`. The function uses the variable `name` in the code block. There is no explicit assignment of the variable name, it exists because it is a parameter of the function.

When a function is called, you must provide a value for every (mandatory) parameter that is defined for the function. Such a value is called an "argument." Therefore, to call the function `hello()`, you must provide an argument for the parameter `name`. You place this argument between the parentheses of the function call. Note that in your main program you do not need to know that this parameter is called `name`. What it is called is unimportant. The only thing you need to know is that there is a parameter that needs a value, and preferably what kind of value the function is expecting (i.e., what the author of the function expects you to provide).

```
def hello( name ):
    print( "Hello, {}!".format( name ) )

hello( "Adrian" )
hello( "Binky" )
hello( "Caroline" )
hello( "Dante" )
```

The parameters of a function are no more and no less than variables that you can use in the function, and that get their value from outside the function (namely by a function call). The parameters are "local" to the function, i.e., they are not accessible outside the code block of the function, nor do they influence any variable values outside the function. More on that later.

Functions can have multiple parameters. For example, the following function multiplies two parameters and prints the result:

listing0801.py

```
def multiply( x, y ):
    result = x * y
    print( result )

multiply( 2020, 5278238 )
multiply( 2, 3 )
```

8.2.3 Parameter types

In many programming languages, you specify what the data types of the parameters of the functions that you create are. This allows the language to check whether calls to the functions are made with correct arguments. In Python, you do not specify data types. This means that, for example, the `multiply()` function in the code block above can be called with string arguments. If you do so, you will generate a runtime error (as you cannot multiply two strings).

If you want to write a "safe" function, you can check the type of arguments that the function is provided with, using the **`isinstance()`** function. **`isinstance()`** gets a value or a variable as first parameter, and a type as second parameter. It returns **`True`** if the value or variable is of the specified type, and **`False`** otherwise. For example:

listing0802.py

```
a = "Hello"
if isinstance( a, int ):
    print( "integer" )
elif isinstance( a, float ):
    print( "float" )
elif isinstance( a, str ):
    print( "string" )
else:
    print( "other" )
```

Of course, should you decide to do such type checking in a function, you must decide what you will do if the user provides the wrong type. The regular way to handle this is by "raising an exception." This will be discussed in Chapter 17. For now, you may assume that the functions that you write are called with parameters of the correct types. As long as you write functions for your own use, you can always guarantee that.

8.2.4 Default parameter values

It is possible to provide default values for some of the parameters. You do this by placing an assignment operator and value next to the parameter name, as if it is a regular assignment. When calling the function, you can specify all parameters, or just some of them. In principle the values get passed to the function's parameters from left to right; if you pass fewer values than there are parameters, as long as default values for the remaining parameters are given, the function call gets executed without runtime errors.

If you define a function with default values for some of the parameters, but not all, it is common to place the ones for which you give a default value to the right of the ones for which you do not.

If you want to override the default value for a specific parameter, and you know its name but not its place in the order of the parameters, or you simply want to leave the other parameters at their default value, you can actually in the function call give a value to a *specific* parameter by name, using an assignment between the parentheses, i.e., `<function>( <parametername>=<value> )`.

The following code gives examples of these possibilities:

listing0802a.py

```
def multiply_xyz( x, y=1, z=7 ):
    print( x * y * z )

multiply_xyz( 2, 2, 2 ) # x=2, y=2, z=2
multiply_xyz( 2, 5 )    # x=2, y=5, z=7
multiply_xyz( 2, z=5 )  # x=2, y=1, z=5
```

8.2.5 return

Parameters can be used to communicate information from outside a function to the code block of the function. Often, you also want a function to communicate information to the rest of the program outside the function. The keyword **return** accomplishes this.

When you use the command **return** in a function, that ends the processing of the function, and Python will continue with the code that comes after the call to the function. You can put one or more values or variables after the **return** statement. These values are communicated to the program outside the function. If you want to use them outside the function, you can put them into a variable when you assign the call to the function to that variable.

For instance, suppose a function is called using a statement <variable> = <function>(). The function makes a calculation and stores it as <result>. **return** <result> then ends the function, and the value stored in <result> "comes out of" the function. Due to the way <function>() was called, this value ends up in <variable>.

If this sounds a bit convoluted, it will probably become clear after studying the following example:

listing0803.py

```
from math import sqrt

def pythagoras( a, b ):
    return sqrt( a*a + b*b )

c = pythagoras( 3, 4 )
print( c )
```

The function pythagoras() calculates the value of the square root of the sum of the squares of its two parameters. Then it returns that value, using the **return** statement. The main program "captures" the value by assigning it to variable c, then prints the contents of c.

Note that the **return** statement in the example above has a complete calculation with it. That calculation is done in the function, which leads to a value. It is the result of the calculation, i.e., the value, which is returned to the main program.

Now suppose that you want to only do this calculation for positive numbers (which would not be strange, as the function clearly is meant to calculate the length of the diagonal side of a right triangle, and who ever heard of a triangle with sides of length zero or less). Examine this code:

listing0804.py

```
from math import sqrt

def pythagoras( a, b ):
    if a <= 0 or b <= 0:
        return
    return sqrt( a*a + b*b )

print( pythagoras( 3, 4 ) )
print( pythagoras( -3, 4 ) )
```

At first glance this code might seem fine: as it has nothing to calculate for negative numbers, it just returns no value. However, when you run this program you find it prints the special value **None**. I discussed this special value in Chapter 5. The main program expected the function `pythagoras()` to return a number, so `pythagoras()` is failing its duties by returning nothing in certain circumstances. You should always be very clear about what data type your function returns, and ensure that in each and every circumstance the function actually returns a value of that type. By the way, the following code is equivalent to the code above (and has the same mistake):

listing0805.py

```
from math import sqrt

def pythagoras( a, b ):
    if a > 0 and b > 0:
        return sqrt( a*a + b*b )

print( pythagoras( 3, 4 ) )
print( pythagoras( -3, 4 ) )
```

In this code, you do not see the **return** without a value explicitly, but it is there nonetheless. If Python reaches the end of a function code block without having found a **return**, it will return from the function without a value.

If you wonder what you should return in circumstances that you do not have a good return value for: that depends on the application. For instance, for the `pythagoras()` function, you could decide that it will return -1 whenever it gets provided with arguments that it cannot process. As long as you communicate that to the user of a function, the user can ensure that the main program handles such exceptional cases in the spirit of the program as a whole. For instance:

listing0806.py

```
from math import sqrt
from pcinput import getInteger

def pythagoras( a, b ):
    if a <= 0 or b <= 0:
        return -1
    return sqrt( a*a + b*b )
```

```
num1 = getInteger( "Give side 1: " )
num2 = getInteger( "Give side 2: " )
num3 = pythagoras( num1, num2 )
if num3 < 0:
    print( "The numbers you provided cannot be used." )
else:
    print( "The diagonal side's length is", num3 )
```

Note that every line of code in the function that occurs immediately after a **return** at the same level of indentation will always be ignored. E.g., in the function:

```
from math import sqrt

def pythagoras( a, b ):
    if a <= 0 or b <= 0:
        return -1
        print( "This line will never be printed" )
    return sqrt( a*a + b*b )
```

the line below **return** -1 clearly states how useless it is.

8.2.6 Difference between `return` and `print`

I noticed in the past that many students struggle with the difference between a function returning a value and a function printing a value. Compare the following two pieces of code:

```
def print3():
   print( 3 )
print3()
```

and:

```
def return3():
   return 3
print( return3() )
```

Both the function `print3()` and `return3()` are called in their respective codes, and result in the printing of the value 3. The difference is that the printing of this value in the case of `print3()` happens in the function, while the function returns nothing, while in the case of `return3()` the function only returns the value 3, which is then printed in the main program. For the user the result of these codes looks the same: both display the number 3. But for the programmer the two functions involved are quite different.

The function `print3()` can only be used for one purpose, namely to display the number 3. The function `return3()`, however, can be used wherever I need the number 3, regardless whether I need to display it, use it in a calculation, or assign it to a variable. For instance, the following code raises 2 to the power of 3 and prints the result:

```
def return3():
   return 3
x = 2 ** return3()
print( x )
```

On the other hand, the following code leads to a runtime error when executed:

```
def print3():
    print( 3 )
x = 2 ** print3() # Erroneous code!
print( x )
```

The reason is that while `print3()` displays the value of 3 on the screen (you even see it above the runtime error when you run the code), it does not produce the actual value 3 in such a way that the calculation can use it. The function `print3()` actually returns the special value **`None`**, which cannot be used in a calculation.

So, if you want to create a function that produces a value that can be used in other parts of the program, then the function must return that value. If you want to create a function that just displays something on the screen, you can use a print statement in the function to do that, but the function does not need to return anything.

8.2.7 Welcome to the machine

If you still have troubles imagining how functions work, think of them like this:

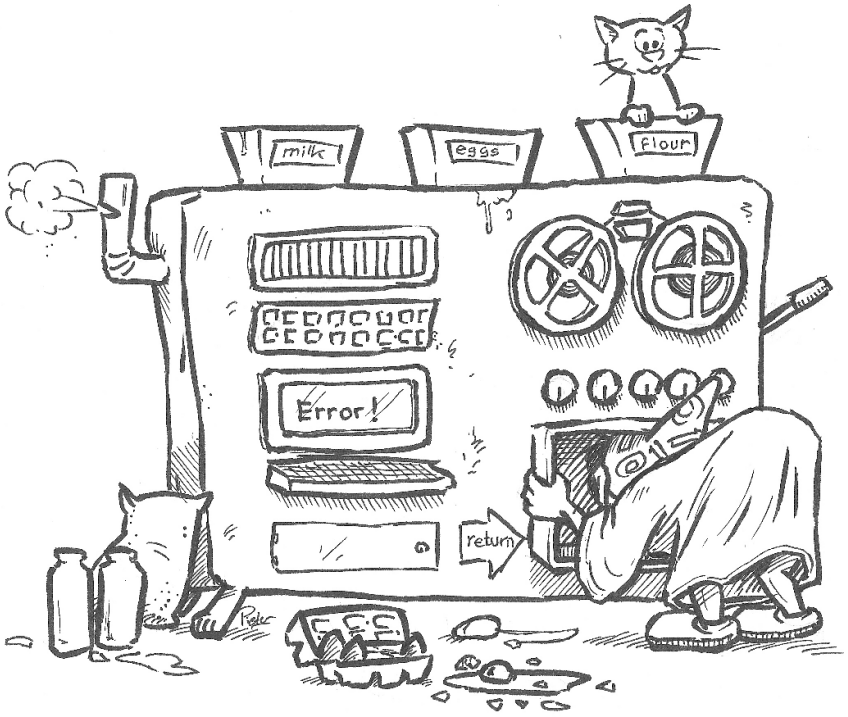
A function is like a big machine, for instance, a machine that makes pancakes. It has some input hoppers at the top, which are labeled "milk," "eggs," and "flour." Those are the input parameters. You can decide what pancakes you want by putting the right stuff in the hoppers; for instance, if you want whole-grain pancakes, you put whole-grain flour into the "flour" hopper. Of course, you can be certain that things will go dramatically wrong if you put eggs into the "milk" hopper – or you try to put a cat into the "flour" hopper.

Anyway, once the hoppers are loaded the machine starts huffing and puffing. You patiently wait next to the output hopper which, surprise surprise, is labeled "return." And after a short while, a pancake slides out. The pancake is the return value that the machine produced.

The machine also has a display. Maybe after you put something inappropriate into the "flour" hopper, the display says: "Cat stuck in the machine, please reset." The display is where everything that you "print" in the machine appears.

Now, you understand that it is useless if, after loading the right ingredients into the input hopper, the machine just displays "Pancake is ready!" You want the actual pancake. That's why the machine must "return" it via its output hopper, from which you can take it and "assign" it to your lunch plate. Just printing that the pancake exists is not sufficient.

By the way, one of the nice things about the pancake making machine is that even if you do not know how pancakes are made, you can still get pancakes as long as you manage to supply the right ingredients. That's also what is so nice about functions: they may do complex things for you without you needing to know how they accomplish them.



8.2.8 Multiple return values

In your functions, you are not limited to returning just one value. You can return multiple values by putting commas in between. If you want to use these values in the program after the call to the function, you have to assign them to multiple variables. You put them to the left of the assignment, also with commas in between. This is easiest to illustrate with and example:

listing0807.py

```
import datetime

def addDays( year, month, day, dayincrement ):
    startdate = datetime.datetime( year, month, day )
    enddate = startdate + datetime.timedelta( days=dayincrement )
    return enddate.year, enddate.month, enddate.day

y, m, d = addDays( 2015, 11, 13, 55 )
print( "{}/{}/{}".format( y, m, d ) )
```

The function `addDays()` gets four arguments, namely integers indicating a year, a month, and a day, and a number of days that you want to add to that date. It returns three values, namely a new year, month, and day. These are in this code captured in the main program in three variables, namely `y`, `m`, and `d`.

When you look at the code above, you might be mystified how exactly `addDays()` is doing its job. As I said, it's a nice thing about functions that as long as the function works and you know what arguments it wants and what it returns, you can use the function without

any knowledge of its internal process. So you can just ignore the code for `addDays()` (note: the contents of `addDays()` use the `datetime` module, which is discussed in Chapter 27).

8.2.9 Calling functions from functions

Functions are allowed to call other functions, as long as those other functions are known to the calling function. For instance, the following code shows how the function `euclideanDistance()` uses the function `pythagoras()` to calculate the distance between two points in 2-dimensional space.

listing0808.py

```
from math import sqrt

def pythagoras( a, b ):
    if a <= 0 or b <= 0:
        return -1
    return sqrt( a*a + b*b )

def euclideanDistance( x1, y1, x2, y2 ):
    return pythagoras( abs( x1 - x2 ), abs( y1 - y2 ) )

print( euclideanDistance( 1, 1, 4, 5 ) )
```

`euclideanDistance()` knows `pythagoras()`, because `pythagoras()` was defined before `euclideanDistance()` is called.

If you want, you can put functions inside other functions, i.e., you can "nest" functions. For instance, you can place `pythagoras()` in `euclideanDistance()`. That means that `pythagoras()` can be called inside `euclideanDistance()`, but not elsewhere in the program.

listing0809.py

```
from math import sqrt

def euclideanDistance( x1, y1, x2, y2 ):

    def pythagoras_inside( a, b ):
        if a <= 0 or b <= 0:
            return -1
        return sqrt( a*a + b*b )

    return pythagoras_inside( abs( x1 - x2 ), abs( y1 - y2 ) )

print( euclideanDistance( 1, 1, 4, 5 ) )
# print( pythagoras_inside( 3, 4 ) )
```

It is not very common that nested functions are used, but it is possible.

Note: if you remove the hash mark before the last line of the code above, it adds a call to `pythagoras_inside()`. This will cause a runtime error, as `pythagoras_inside()` is only visible inside `euclideanDistance()`.

Exercise Write a function called `printx()` that just prints the letter "x." Then write a function called `multiplex()` which takes as argument an integer and prints as many times the letter "x" as the integer indicates by calling the function `printx()` that many times.

8.2.10 Naming functions

Convention prescribes that you should not start a function name with an underscore (such function names are reserved for the developers of Python itself), and that you try to use only lower case letters. If the function name consists of multiple words, you can either put underscores between the words, or start every word except the first with a capital (different programmers prefer different conventions in this respect, but it does not matter much as you can always recognize a function by the fact that it has parentheses after the name).

Certain function names are typical for particular functionalities.

A function that tests if a certain item has a certain property, which then returns either **`True`** or **`False`** depending on whether the property holds, commonly has a name that starts with the word `is`, which is then followed by the name of the property, starting with a capital. For instance, a function that tests if a number is even, would be called `isEven()`.

Exercise Write the function `isEven()`.

Exercise Write the function `isOdd()`, which determines whether a number is odd, by calling the function `isEven()` and inverting its result.

Note: if you want to use a function like `isEven()` in a conditional statement, for instance, because you want to execute an action only for numbers that are even, you do not need to write **`if`** `isEven( num )` == **`True`**:. You can simply write **`if`** `isEven( num )`:, because the function already returns **`True`** or **`False`**. Using an is-function in such a way makes a program more readable.

A function that gets the value of a certain property and returns it, commonly starts with the word `get`, which is then followed by the name of the property, starting with a capital. For instance, a functions that gets the fractional part of a float (i.e., the decimals) would be called `getFraction()`.

Exercise Write the function getFraction().

The opposite of a get function is a function that gives a property a certain value. Such a function commonly starts with `set`, and for the rest is similar to a `get` function. I cannot give an example as at this point I have not yet explained how you can use a function to give something a value, as functions cannot change the values of their parameters (at least, for the data types that we have used until now). This will follow in a later chapter.

If you stick to such naming conventions, it will make your code more readable.

8.2.11 Commenting functions

In all the chapters until now, you have seen very little commenting of code. While books and courses on programming often encourage students to write comments in code, I myself am of the opinion that code should be "self-documenting," i.e., that you can easily derive from code what it does simply by reading it. You can accomplish that often by choosing strong names for variables and functions, judiciously using white spaces and empty lines, good indenting (which Python enforces), and not using any convoluted trickery just because it makes the code a little bit faster or to show off how smart you are.[5]

So while I see comments as an extra that you should use when you feel you need to explain something particular about your code, my opinion on comments for functions is different. The idea behind a function is that the user of the function does not need to look at the function's code to use it. Therefore, what the function does and how it works, should be explained in comments at the top of the function, above the function name.

In the comments for a function, you explain three things:

- What the function does
- What arguments the function needs/accepts, including data types
- What the function returns, including data types

If a function has any side effects, i.e., things it affects in the main program, then this should be carefully documented in the comments too. I do not put that in the list above, because a function *should not have any side effects*.

Note: For the answers to the exercises in this chapter I have added comments to the functions in a form that I find acceptable. In follow-up chapters I often will not do that as I discuss the functions in text anyway, or I want you to study the contents of the function. However, I always write comments for functions that I write in code that I use for other purposes.

8.3 Scope and lifetime

Scope refers to visibility. In particular, when discussing the scope of a variable, it refers to the places in a program where a variable is visible and can be changed. Lifetime refers to how long a variable exists in memory. Lifetime is closely related to scope, which is why I discuss them in one section.

8.3.1 Scope of variables

In general, the scope of a variable is at least the code block in which it is created, and all the code blocks that are nested within that code block at a deeper indent level. The follow code demonstrates how the scope of variables is defined for Python:

[5] A little anecdote on the side here: I once heard someone extol the intellect of a certain programmer by saying "when I see his code, I don't understand any of it!" When someone would say that about my code, I would feel deeply ashamed.

listing0810.py

```
hello = "Hi!"
bye = "Goodbye!"

for i in range( 3 ):
    for j in range( 2 ):
        afternoon = "Good afternoon"
        print( bye )
    print( j )
    print( hello )
    print( afternoon )

print( i )
print( j )
print( afternoon )
```

The variables `hello` and `bye` are created at the top level of the program, which means their scope is the whole program. The variables `i`, `j`, and `afternoon` are defined in code blocks which are at a deeper indent level. In most programming languages, that would mean that their scope would be restricted to those deeper levels, but Python is friendly in this respect and makes their scope extend beyond the loop that they are in. So all these variables are visible in the program after they have been defined.

How does this work with functions?

listing0811.py

```
dozen = 12

def dimeAdozen():
    print( "There are", dozen/dime, "dimes in a dozen" )

dime = 10
dimeAdozen()
print( "dime =", dime, "and dozen =", dozen )
```

Again, we see that both `dozen` and `dime` are visible within the function `dimeAdozen()`. They can be seen by the function because they have been defined before the function is called, and since the code block of the function is at a deeper indent level, it can see these variables.

However, now look at the following code, which contains a small change from the code above:

listing0812.py

```
dozen = 12

def dimeAdozen():
    dozen = 13
    print( "There are", dozen/dime, "dimes in a dozen" )
```

```
dime = 10
dimeAdozen()
print( "dime =", dime, "and dozen =", dozen )
```

Run this code, then examine it and its output closely, and compare it with the code and the output of the code block above it. The variable `dozen` seems to get a new value in the function `dimeAdozen()`, which leads to the function claiming that there are now 1.3 dimes in a dozen. However, when the value of `dozen` is printed in the main program, its value is shown to be still 12, and not 13.

The reason is that the variable `dozen` in the function is a different one than the variable `dozen` in the main program. By assigning a value to a variable in a function, a new, "local" variable is created. And this variable is used for the remainder of the function. The original variable `dozen` still exists, but is invisible to the function once it has created its own `dozen`.

The lifetime of the variable `dozen` in the function is the period for which the code block of the function is executed. As soon as the function ends (for instance, because of a **`return`** or because the last line was executed), the local variables of the function are destroyed. They are no longer in the computer's memory, and can no longer be accessed.

listing0813.py

```
apple = "apple"
banana = "banana"
cherry = "cherry"
durian = "durian"

def printfruits_1():
    print( apple, banana )

def printfruits_2( apple ):
    banana = cherry
    print( apple, banana )

def printfruits_3( apple, banana ):
    cherry = "mango"
    banana = cherry
    print( apple, banana )

printfruits_1()
printfruits_2( cherry )
printfruits_3( cherry, durian )

print( "apple =", apple )
print( "banana =", banana )
print( "cherry =", cherry )
print( "durian =", durian )
```

Run this code and study it closely.

The three functions `printfruits_1()`, `printfruits_2()`, and `printfruits_3()` print the variables `apple` and `banana`.

In `printfruits_1()` these are the two variables `apple` and `banana` that are defined outside the function, as the function itself does not try to define these variables.

In `printfruits_2()`, `apple` is the parameter of the function, which means it is a variable local to the function that gets its value from outside the function. The value it gets is the value of the variable `cherry` (because `cherry` is provided as the argument when the function is called), which is the word "cherry." `banana` is a variable that gets its value in the function. This is a new, local variable `banana`, which has nothing to do with the variable `banana` in the main program. It gets the value of `cherry`, which is not locally known to the function, so it uses the variable `cherry` from the main program for that. Therefore, the local variable `banana` gets the value "cherry," and this is the value that is printed.

In `printfruits_3()`, `apple` and `banana` are both parameters, so they are both variables that are local to the function and that get their initial value from the call to the function. The function then creates a local variable `cherry`, which is independent from the variable `cherry` from the main program. It then assigns the value of `cherry`, which is `"mango"`, to the local variable `banana`. All these changes are therefore made to local variables, and have no influence on the values of variables from the main program.

When after the function calls the values of the variables from the main program are printed, you see that they still have the values that were originally assigned to them, regardless of whether they were used as arguments to the function calls, or whether variables in the functions with the same names got different values assigned. As soon as a variable in a function gets assigned a value, if that variable was not yet created in the function and was not a parameter of the function, a new, local variable is created and used in the function. Such a new, local variable is completely independent from any variable which exists outside the function. Its lifetime is the period for which the function is executed. Parameters can also be considered local variables of a function.

This is a very powerful feature of functions: they do not have to take into account variables that exist outside the function, as any variable that they create is local to the function.

8.3.2 Global variables

I showed above that variables that are created outside a function are visible in the function, unless a new variable with the same name is created in the function. Variables from the main program are called "global" variables, as they are visible anywhere in the program, as opposed to "local" variables that are only visible in a function.

It is good practice to make functions independent from the main program, i.e., to not let them access any of the global variables. If you do need to communicate values from outside a function to the function, then do so by means of parameters. An exception can be made for variables that are used as constants (see Chapter 4). If you do let a function access a constant, then make sure it is clear to anyone who inspects the function that you are referring to a constant, i.e., that the name of the constant is written in all capitals.

You might wonder if it isn't possible to change the values of global variables in a function. This is, in fact, possible, but you have to make clear that you explicitly want the global variable to be affected by using the keyword **`global`**. The statement **`global`** `<variable>` indicates that the particular variable mentioned is actually referring to the global variable of this name. For example:

```
fruit = "apple"

def changeFruit():
    global fruit
    fruit = "banana"

print( fruit )
changeFruit()
print( fruit )
```

While it is possible to affect global variables in functions, this is not recommended as it makes the function dependent on the main program (and thus no longer a "pure" function). Basically, it makes the function have side effects, and (all together now:) *a function should not have any side effects*.

It is also never necessary to include global variables in a function. If you want to allow a function to affect a global variable, then let the function return a value that can be assigned to the global variable. Leave it to the main program to decide whether or not to overwrite the value of one of its own variables. The only reason I mention it here is that I sometimes see students reverting to the keyword **global** because they have insufficient understanding of **return** statements. Denying the existence of **global** is not effective, I rather admit that it exists and warn students against using it.

8.4 Managing program complexity

Suppose that Python would not have built-in **max**() and **min**() function, and neither do you have knowledge of (or are allowed to) use anything of the chapters after this one. You get the following assignment:

Write a program that processes two groups of three numbers (you can write the program for fixed numbers, but later on you will add that the user enters these numbers). It adds up the lowest numbers of each of the groups, the middle numbers of each of the groups, and the highest numbers of each of the groups. It then prints these three results.

How do you do this? You can start with something like:

```
# First initialize variables in group 1 (num11, num12, num13)
# and in group 2 (num21, num22, num23) to some values.

smallest1 = 0
smallest2 = 0
medium1 = 0
medium2 = 0
largest1 = 0
largest2 = 0

if num11 < num12:
    if num11 < num13:
        smallest1 = num11
```

```
    else:
        smallest1 = num13
elif num12 < num13:
    smallest1 = num12
else:
    smallest1 = num13

# Test:
print( smallest1 )

# This works to get the smallest from group 1.
# Now do the same for the smallest from group 2.
# Then do something similar for the largest of group 1 and 2.
# Then invent something for taking the middle one.
# Finally, do all the additions and print the results...
```

You can imagine that with this approach, with nested **if** statements that get repeated six times with different assignments in the branches, this becomes a huge, unreadable, unmanageable program of which it is hard to see whether it is correct or not. You have to approach the problem in a smarter way.

Suppose that you have a function that determines the smallest of three numbers, a function that determines the middle one of three numbers, and a function that determines the largest one of three numbers. Then the program is pretty simple to write! It will be something like:

listing0814.py

```
num11, num12, num13 = 436, 178, 992
num21, num22, num23 = 880, 543, 101

def smallest( n1, n2, n3 ):
    return n1 # just return something for now

def middle( n1, n2, n3 ):
    return n1 # just return something for now

def largest( n1, n2, n3 ):
    return n1 # just return something for now

print( "sum of smallest =", smallest( num11, num12, num13 ) +
    smallest( num21, num22, num23 ) )
print( "sum of middle =", middle( num11, num12, num13 ) +
    middle( num21, num22, num23 ) )
print( "sum of largest =", largest( num11, num12, num13 ) +
    largest( num21, num22, num23 ) )
```

Note: In the code above, to reduce the size, I used a "multiple assignment" to give the variables num*xx* their values. You can have multiple variables at the right of the assignment operator, and an equal number of values to the left, and the first value will then go to the first variable, the second value to the second variable, etcetera. I will discuss this in more depth in Chapter 11.

The program above readable, understandable, and can already be tested. True, the functions `smallest()`, `middle()`, and `largest()` do not return the correct values yet. While writing the program above, you might not even have an idea on how to write them. But you probably feel that they could be written, and you know that you can produce code for them later, and step by step.

So how do you do `smallest()`? Well, as I showed above, doing this with nested **`if`** statements becomes a bit convoluted and unreadable (really, don't look at how I did it and try to write this yourself; it is pretty hard to keep the three variables in your head while writing such a nested **`if`**). Can this be approached in a more readable way?

Is it hard to determine the smallest of two numbers? No, that is really easy:

```
def smallest_of_two( n1, n2 ):
    if n1 < n2:
        return n1
    return n2
```

By nesting such a function, you can make a `smallest()` function that determines the smallest of three numbers. The same can be done for `largest()`. So the program now becomes:

listing0815.py

```
num11, num12, num13 = 436, 178, 992
num21, num22, num23 = 880, 543, 101

def smallest_of_two( n1, n2 ):
    if n1 < n2:
        return n1
    return n2

def largest_of_two( n1, n2 ):
    if n1 > n2:
        return n1
    return n2

def smallest( n1, n2, n3 ):
    return smallest_of_two( smallest_of_two( n1, n2 ), n3 )

def middle( n1, n2, n3 ):
    return n1 # just return something for now

def largest( n1, n2, n3 ):
    return largest_of_two( largest_of_two( n1, n2 ), n3 )

print( "sum of smallest =", smallest( num11, num12, num13 ) +
    smallest( num21, num22, num23 ) )
print( "sum of middle =", middle( num11, num12, num13 ) +
    middle( num21, num22, num23 ) )
print( "sum of largest =", largest( num11, num12, num13 ) +
    largest( num21, num22, num23 ) )
```

The program now works as far as smallest numbers and largest numbers are concerned. To complete the code, a solution must be found for the middle. What is the middle of three numbers? It is the number that remains if the smallest and largest are taken out. Can this be programmed? Yes, it is simply the sum of the three numbers, subtracting the smallest and the largest. This is implemented as follows:

listing0816.py

```
num11, num12, num13 = 436, 178, 992
num21, num22, num23 = 880, 543, 101

def smallest_of_two( n1, n2 ):
    if n1 < n2:
        return n1
    return n2

def largest_of_two( n1, n2 ):
    if n1 > n2:
        return n1
    return n2

def smallest( n1, n2, n3 ):
    return smallest_of_two( smallest_of_two( n1, n2 ), n3 )

def middle( n1, n2, n3 ):
    return n1 + n2 + n3 - smallest( n1, n2, n3 ) \
        - largest( n1, n2, n3 )

def largest( n1, n2, n3 ):
    return largest_of_two( largest_of_two( n1, n2 ), n3 )

print( "sum of smallest =", smallest( num11, num12, num13 ) +
    smallest( num21, num22, num23 ) )
print( "sum of middle =", middle( num11, num12, num13 ) +
    middle( num21, num22, num23 ) )
print( "sum of largest =", largest( num11, num12, num13 ) +
    largest( num21, num22, num23 ) )
```

The program is now finished and it works. It is fairly long, but all the functions are easy to understand, and it is also easy to understand why the program works. It is still shorter than the original attempt, with at least six nested **if** statements, would have been, and it is a lot more readable.

It might be that there are different approaches for this program. With some inventiveness, you might come up with smarter ways to determine smallest, middle, and largest. But the program works, and is understandable, and that is the most important.

You can criticize the approach that I take in this program. For instance, calculation of the smallest of the same three numbers takes place twice: once to determine the smallest, and once to determine the middle. The same holds for the largest. Can this be optimized, so that such a determination takes place only once? Of course it can, for instance by the

introduction of two extra variables that keep track of the smallest and largest numbers. But why would I? That would not make the program more readable, and while it would make the program a bit faster, I am talking nanoseconds here. For a program like this, speed is unimportant and completely subject to readability. Let me stress again that while learning programming, solving a problem correctly comes first, immediately followed by solving a problem in a readable and maintainable way. Efficiency comes much later.

What you should learn from this, is that when a program consists of a series of problems that you find hard to solve, you should try to split it into sub-problems or sub-goals, and solve these independently. You can often already introduce functions for sub-problems when you set up the program, and then for the time being fill these function templates with something simple, like returning a constant. You can then at least test the program. Later on, you can start filling in all the function templates that you created.

8.5 Modules

Creating a module is very simple. You just create a Python file, with extension `.py`, and place functions in it. You can then import this Python file in another Python program (you just use the name of the file without the extension `.py`; the file should be either in the same folder as the program, or in a standard Python modules location), and access its functions just as you access functions from regular Python modules, i.e., you either import specific functions from the module, or you import the module as a whole, and call its functions by using the `<module>.<function>()` syntax.

8.5.1 `main()`

When examining other people's Python programs, in particular those that contain functions that you might want to import, you often see a construct like shown below:

```
def main():
    # code...
if __name__ == '__main__':
    main()
```

The function `main()` contains the core of the program, and may call other functions.

There is no need to understand this exactly, but what happens here is the following: the Python file that contains the code can run as a program, or the functions that it contains can be imported into other programs. The construction shown here ensures that the program only executes `main()` (which is the core program) if the program is run as a separate program, rather than being loaded as a module. If, instead, the program is loaded as a module into another program, only its functions can be accessed, and the code for `main()` is ignored.

If the Python file that contains such a construct is predominantly used as a module, the `main()` function usually contains some code that tests the functions in the module. This is useful during development time.

Here is an example of such a construction:

listing0817.py

```
def isEven( num ):
    return num%2 == 0

if __name__ == '__main__':
    for i in range( 10 ):
        if isEven( i ):
            print( i, "is even" )
```

If you run the code above as a program, it will tell you that the numbers 0, 2, 4, 6, and 8 are all even. If instead you load this program as a module into another program to access the `isEven()` function (using **`from ... import`** `isEven`), then your program will not tell you anything about the numbers 0, 2, 4, 6, and 8, but the function `isEven()` will be available for use. However, if the construction with **`if`** `__name__ == '__main__':` had not been used, every program that would import the module would print the statements about the numbers 0, 2, 4, 6, and 8.

The use of a program `main()` for the main functionality has an extra use. Since it is a function, if you want to leave the program for some reason in the middle of processing, you do not need to use the `exit()` function from the `sys` module. You can simple **`return`** from the `main()` function. This avoids the ugly error message that some editors give on using the `exit()` function.

8.6 Anonymous functions

The concept of "anonymous functions" should be considered optional material: they are rarely used, and never needed. However, for completeness I discuss them here.

Python allows a program to create a function that has no name. The function can be assigned to a variable, and the variable can then be used as if it is a function. To create an anonymous function, you use the following syntax:

```
lambda <parameters>: <statement>
```

`lambda` is a keyword. `<parameters>` is a sequence of parameter names, separated by comma's if there is more than one. `<statement>` is one single statement. The anonymous function does not need the keyword **`return`**, but the value of `<statement>` is used as return value.

For instance, the following code creates an anonymous function that calculates the square of its parameter. The function gets assigned to a variable `f`. `f` can then be called as a function, to calculate the squares of numbers.

```
f = lambda x: x*x
print( f(12) )
```

This code is exactly the same as the following code:

```
def f( x ):
    return x*x
print( f(12) )
```

So, if anonymous functions are no different from regular functions, and actually more limited as they can only use a single line of code, why are they included in Python? Actually, there has been going on quite a lot of debate amongst the people who create Python whether or not the **`lambda`** keyword should remain. It is part of Python because it is also part of other programming languages, in particular functional programming languages such as Lisp and Haskell, which rely on the concept of anonymous functions. But the **`lambda`** keyword in Python is not as powerful as the **`lambda`** keyword in these other languages, and, as we have seen, not really needed. A main reason that it is still part of Python is backwards compatibility and the fact that there are many Python users who like to use it.

Occasionally, anonymous functions have their uses, and can actually make programs a bit more readable. I will show an example in Chapter 12.

What you learned

In this chapter, you learned about:

- The purpose of functions
- Creating functions
- Parameters and arguments
- Returning values from functions with **`return`**
- Naming conventions for functions
- Commenting functions
- Variable scope and lifetime
- Local and global variables
- Using functions to manage program complexity
- Creating modules
- Using a `main()` function
- Anonymous functions

Exercises

In these exercises you write functions. Of course, you should not only write the functions, you should also write code to test them. For practice, you should also comment your functions as explained above.

Exercise 8.1 Create a function that gets a number as parameter, and then prints the multiplication table for that number from 1 to 10. E.g., when the parameter is 12, the first line printed is "1 * 12 = 12" and the last line printed is "10 * 12 = 120."

Exercise 8.2 Write a function that gets as parameters two strings. The function returns the number of characters that the strings have in common. Each character counts only once, e.g., the strings `"bee"` and `"peer"` only have one character in common (the letter "e"). You can consider capitals different from lower case letters. Note: the function should *return* the number of characters that the strings have in common, and not print it. To test the function, you can print the result in your main program.

Exercise 8.3 The Grerory-Leibnitz series approximates pi as $4 * (1/1 - 1/3 + 1/5 - 1/7 + 1/9...)$. Write a function that returns the approximation of pi according to this series. The function gets one parameter, namely an integer that indicates how many of the terms between the parentheses must be calculated.

Exercise 8.4 In Chapter 6 you were asked to implement the quadratic formula to solve quadratic equations. A quadratic equation is described by three numeric values, usually called A, B, and C. It has zero, one, or two solutions, depending on the discriminant (the part under the square root). Write a function that solves a quadratic equation. As parameters it gets A, B, and C. It returns three values. The first is an integer that indicates the number of solutions. The second is the first solution. The third is the second solution. Any of the solutions that do not exist, you can return as zero.

Exercise 8.5 In Chapter 7, the loop-and-a-half was explained. The final code for the example that was presented is given below, and I made the remark that there is still something ugly about this code, namely the fact that if `x` is smaller than zero or higher than 1000, the code still asks for `y` even when it can know that it has to ask a new value for `x`. I also remarked that you can resolve this in an easy way by using a function. Create a function and insert it in this code, so that this issue gets fixed. Also get rid of the `exit()` and thus the possible ugly output by introducing a `main()` function.

exercise0805.py

```
from pcinput import getInteger
from sys import exit

while True:
    x = getInteger( "Enter number 1: " )
    if x == 0:
        break
    y = getInteger( "Enter number 2: " )
    if y == 0:
        break
    if (x < 0 or x > 1000) or (y < 0 or y > 1000):
        print( "The numbers should be between 0 and 1000" )
        continue
    if x%y == 0 or y%x == 0:
        print( "Error: the numbers cannot be dividers" )
        exit()
    print( "Multiplication of", x, "and", y, "gives", x * y )

print( "Goodbye!" )
```

Exercise 8.6 In statistics, the binomial coefficient indexed by `n` and `k` (often expressed as "n over k," whereby `n` must be bigger than or equal to `k`) is calculated as $n!/(k! * (n-k)!)$, whereby $n!$ indicates the factorial of `n`. As I explained in Chapter 7: the factorial of a positive integer is that integer, multiplied by all positive integers that are lower (excluding zero). You write the factorial as the number with an exclamation mark after it. E.g., the factorial of 5 is $5! = 5 * 4 * 3 * 2 * 1 = 120$. If you did all the exercises until now, you wrote some code for this. Write a function that calculates the binomial coefficient for its two parameters, and returns the value. Write the code in such a way that it can be used as a module by another program (i.e., put the tests of your program in a `main()` function that is called as explained above).

Exercise 8.7 What is wrong with the following code? Fix it!

exercise0807.py

```
# What is wrong?
def area_of_triangle( bottom, height ):
    area = 0.5 * bottom * height
    print( "The area of a triangle with a bottom of", bottom,
        "and a height of", height, "is", area )

print( area_of_triangle( 4.5, 1.0 ) )
```

Note: Code like this is typically written by students who have not yet grasped the intricacies of using functions.

Chapter 9

Recursion

Recursion is a special technique that can be used now you are able to create and use functions. Recursion can be very elegant and powerful, but students often find it hard to employ. That is why I decided to spend a separate chapter on it. If, while studying this chapter, you feel that it is getting too complex for you, feel free to skip it for the time being. The following chapters are a lot easier again.

9.1 What is recursion?

Recursion is a technique whereby a function calls itself. In a bit more general sense, it is when a function makes calls in such a way that the function itself is still being executed while it gets called again (e.g., function `a()` calls function `b()`, which calls function `a()` again).

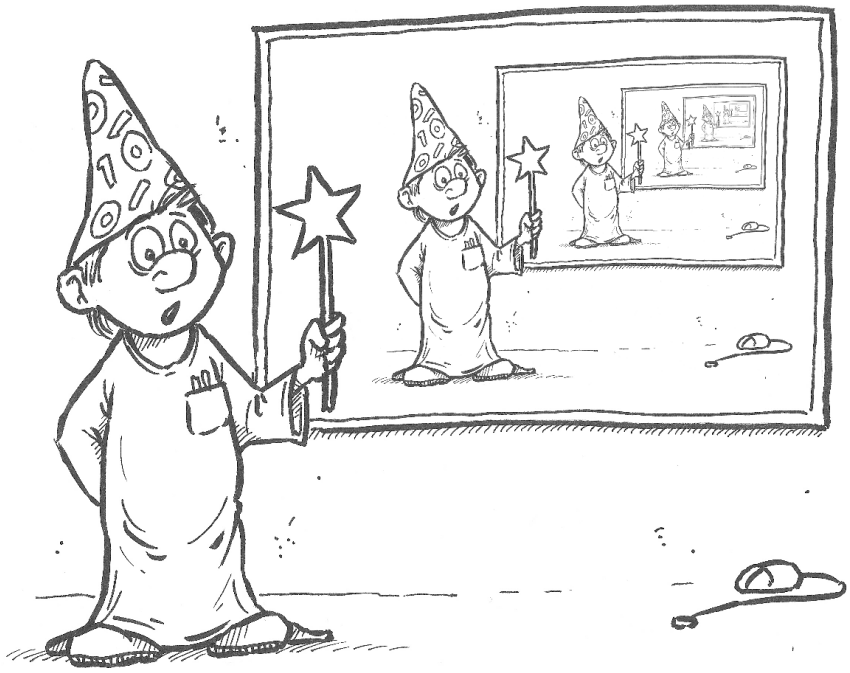
This might sound weird when you first encounter it, but there is nothing against a function calling other functions, and a function can call any function that has been defined by the time that the call takes place. And since a function is defined by the time its code gets executed, it can call itself.

"But," one might say: "if a function calls itself, then it calls itself again, and again, and again... Doesn't that mean it gets into an endless process, similar to an endless loop?" The answer is that there is certainly a danger, with sloppy coding, that a recursive function gets into an endless loop, but recursive functions should be designed in such a way that that does not happen.

There exist many problems for which recursion is the most elegant solution. Therefore it is important that you are aware of the technique, and know how and when to apply it... and its limitations.

9.2 Recursive definitions

An example of a recursive definition is the definition of the factorial, which was already introduced in the previous two chapters. In those chapters I gave the following definition



of the factorial: The factorial of a positive integer is that integer, multiplied by all positive integers that are lower (excluding zero).

Mathematicians prefer the recursive definition: The factorial $n!$ of any positive integer n is calculated as follows: $1! = 1$, and $n! = n * (n-1)!$ for $n > 1$.

This definition is recursive as it refers to the factorial of $n-1$ to define the factorial of n. This is not leading to an endless recursion, however, as at some point n will be 1, and the factorial of 1 is defined separately.

You can implement the factorial as a recursive function as follows:

listing0901.py

```
def factorial( n ):
    if n <= 1:
        return 1
    return n * factorial( n-1 )

print( factorial( 5 ) )
```

Notice how this function describes the recursive definition of the factorial exactly: if `n` is 1, it returns 1, and otherwise it returns `n` times the factorial of `n-1`. (Note that I wrote if `n <= 1` instead of `n == 1` to avoid problems with the user calling the function with, for instance, a negative `n`.)

In case you have troubles understanding what happens in this function, let's describe the details of the calls it makes. I have indented calls that are made while a "high level" call is still active. A return statement which is indented one level deeper than a call statement is given in that call and returns from it with the specified value.

```
call factorial( 5 )
    call factorial( 4 )
        call factorial( 3 )
            call factorial( 2 )
                call factorial( 1 )
                    return 1
                return 2 * 1
            return 3 * 2
        return 4 * 6
    return 5 * 24
print( 120 )
```

9.2.1 When to use recursive implementations

Once you understand the recursive implementation of the factorial, it might look appealing. It is simple, elegant, and has a certain coolness factor. However, the iterative implementation of the factorial is highly preferable over the recursive one.

The reason is clear from the call descriptions above. You see that before the call to `factorial( 1 )` is made, four other calls to `factorial()` already reside in memory. Should you wish to calculate the factorial of 100, no less than 100 calls to the function will reside in memory before it can start returning values. This is not a good idea, and, in fact, Python may easily run out of (stack) memory in such a case, or become really, really slow.

Contrariwise, an iterative implementation of the factorial only needs to keep two variables in memory. It is fast and there is no danger of crashing.

So you should only use recursive implementations if:

- recursion is the most natural way to implement the solution; and
- the recursive process is guaranteed not to go too deep.

Any recursive process can also be implemented as an iterative process. However, occasionally you can encounter problems for which the recursive solution is much more elegant, readable, and maintainable than the iterative one. In that case, consider reverting to the recursive solution.

9.2.2 Searching a maze

At this point in the book it is hard to give a good demonstration of recursion, as it needs particular data structures to show its power. But to still show something non-trivial, I have created a module called `pcmaze`. You can find it in Appendix D, and you need to either create it or download it from the same place where you got `pcinput.py` to be able to run the code in this subsection.

`pcmaze` implements a simple maze, which connects some numbered cells. You can ask for the maze's entrance using the function `entrance()`. The maze's exit is given by the function `exit()` (not to be confused with the `exit()` function from `sys`). The module

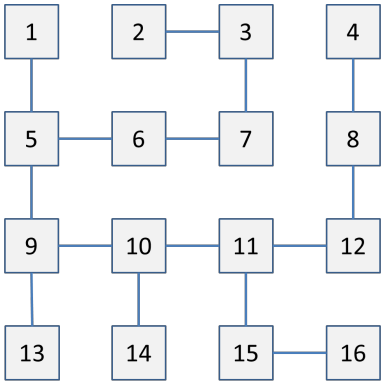


Figure 9.1: The maze implemented in `pcmaze`.

also has a function `connected()` that gets two numeric arguments: it returns **`True`** if there is a direct connection between the cells with those numbers, and **`False`** otherwise. The entrance is guaranteed to be the lowest-numbered cell, while the exit is guaranteed to be the highest-numbered cell.

The goal is to write some code that finds the way from the entrance to the exit (if there is such a way). The maze is visualized in Figure 9.1. Entrance is 1, exit is 16.

So how do you find a way through such a maze (without knowing the exact layout)? Recursively, you can do it as follows: You define a function `leads_to_exit()` that returns a path to the exit if the cell that it is currently examining is on the path that leads to the exit. If that function returns a path, then you know that the current cell is also on the path. If you call it with cell 1, you get a path that leads from the entrance to the exit (if there is such a path).

But how does that function know if a cell is part of a path that leads to the exit? Well, if the current cell actually is the exit, then yes, it leads to the exit. If not, then it leads to the exit if it has a connection with a cell that leads to the exit. This is a recursive definition.

You have to be careful that such a recursive definition cannot get stuck in a circular path in the maze. That means that when the function moves from cell A to cell B, it is not allowed to move back. If that is taken care of, the function should work. It wouldn't work if there would be circular paths in the maze, but fortunately there aren't. The problem is not unsolvable if there are circular paths, but to solve it a data structure is needed that is not discussed yet.

In pseudo-code, the recursive function `leads_to_exit()` is something like this:

```
function leads_to_exit( currentcell ):
    if (currentcell is the exit):
        return (path consisting of only the exit)
    for (every connnectedcell that was not yet explored):
        path = leads_to_exit( connectedcell )
        if (path is not empty):
            add currentcell to path
            return path
    return (empty path)
```

Now let's implement this recursive solution. In the implementation immediately below, the path will not be returned, but I return just **True** or **False** to indicate that the path is found or not, and I print the path in the function itself (a bit further down I provide the complete implementation for the pseudo-code above).

listing0902.py

```
from pcmaze import entrance, exit, connected

def leads_to_exit( comingfrom, cell ):
    if cell == exit():
        return True
    for i in range( entrance(), exit()+1 ):
        if i == comingfrom:
            continue
        if not connected( cell, i ):
            continue
        if leads_to_exit( cell, i ):
            print( cell, "->", i )
            return True
    return False

if leads_to_exit( 0, entrance() ):
    print( "Path found!" )
else:
    print( "Path not found" )
```

Let's look at the recursive function in detail.

It gets two parameters. The first is the cell that the path is coming from. The second is the cell that is checked to see if it leads to the exit. The first parameter is only needed because returning on the path is not allowed.

The function first checks if the exit is reached. If it is, it returns **True**.

If the exit is not reached, the function checks all cells of the maze as possible follow-up cells.

It excludes (a) the cell that it just arrived from; and (b) all the cells to which there are no connections. But it checks all the other cells. There is no need to explicitly exclude the cell itself, as in the definition of the maze a cell is not connected to itself.

As soon as it finds a cell for which a recursive call to the function says that it leads to the exit, while coming from the current cell, it prints that movement and returns **True**. This indicates to the call that arrived here that, yes, a path is found.

Otherwise, once it has checked all possible follow-up connections and no path was found, it returns **False**.

This process prints the whole path from entrance to exit, in reverse order.

To make clear what is happening, I have expanded the function a bit, now also printing every connection that is checked. I have also included a depth parameter, that keeps track of how deep the recursion is going. I translate that into indentations.

listing0903.py

```
from pcmaze import entrance, exit, connected

def leads_to_exit( comingfrom, cell, depth ):
    indent = depth * 4 * " "
    if cell == exit():
        return True
    for i in range( entrance(), exit()+1 ):
        if i == comingfrom:
            continue
        if not connected( cell, i ):
            continue
        print( indent + "Check connection", cell, "->", i )
        if leads_to_exit( cell, i, depth + 1 ):
            print( indent + "Path found:", cell, "->", i )
            return True
    return False

if leads_to_exit( 0, entrance(), 0 ):
    print( "Path found!" )
else:
    print( "Path not found" )
```

9.2.3 Return values of recursive functions

Just like regular functions, recursive functions can communicate information to the rest of the program using their return values.

One of the less nice things about the maze-solving recursive functions above is that they print the path (rather than return it), and that the path is printed in reverse order. It would be better if, instead, the function calls returned their part of the path to the higher level calls, so that the path as a whole is returned from the first call, in the main program. This is what the pseudo-code above proposed. A good way to return a path is in the form of a list, but lists will be discussed in Chapter 12. Instead, I will do it in the form of a string.

It works as follows: A call that finds the exit cell, returns the number of the exit cell as a string. Any call that finds part of the path, returns what it got returned itself, but adds the current cell to that path. Any call that finds nothing, returns nothing, i.e., an empty string.

This means that in the recursive functions above, any **`return True`** will instead return a string containing a (partial) path, and any **`return False`** returns an empty string. The code becomes the following:

listing0904.py

```
from pcmaze import entrance, exit, connected

def leads_to_exit( comingfrom, cell ):
    if cell == exit():
        return "{}".format( exit() )
```

```
    for i in range( entrance(), exit()+1 ):
        if i == comingfrom:
            continue
        if not connected( cell, i ):
            continue
        check = leads_to_exit( cell, i )
        if check != "":
            return "{} -> {}".format( cell, check )
    return ""

check = leads_to_exit( 0, entrance() )
if check != "":
    print( "Path found!", check )
else:
    print( "Path not found" )
```

If you want to understand recursion, study this code closely. This code represents a typical use of return values in recursive functions. Students whose understanding of recursion is wonky and who get an assignment that has them communicate information from a deeper level recursive call to a higher level one, often revert to using a **global** variable. As you can see, that is not necessary.

All in all, there is no real difference between a recursive function call and a regular function call, except that you have to be careful that recursive calls terminate at some point. It only looks strange the first time that you encounter it.

What you learned

In this chapter, you learned about:

- Recursive functions
- When to use (and when not to use) recursive functions
- The purpose of recursive functions
- Using return values from recursive functions

Exercises

Exercise 9.1 A recursive definition of the *n*th number of the Fibonacci sequence `fib(n)` states that `fib(n)` is equal to `fib(n-1) + fib(n-2)`. Moreover, `fib(1)` and `fib(2)` are both 1. Write a recursive function that you can call with an integer argument `n` that returns the *n*th number of the Fibonacci sequence.

Exercise 9.2 To get a bit more insight into how recursion works, add a depth parameter to your Fibonacci function from the previous exercise, that starts at zero and gets increased by 1 for every deeper call. On entry of the function, print with what number argument it

was called, and when returning a value, print what you return. Use the depth parameter to indent the prints. Study your output.

Exercise 9.3 Do you think it is a good idea to implement the Fibonacci sequence recursively? Why or why not?

Exercise 9.4 The greatest common divider is the greatest integer that divides two other integers without remainder. For instance, the greatest common divider of 14 and 21 is 7, as 7 is the greatest number that divides both 14 and 21. Euclid's algorithm that calculates the greatest common divider of two numbers says that if the largest divided by the smallest is an integer, it is the smallest. Otherwise, it is the result of calculating the greatest common divider of the smallest and the remainder of the largest divided by the smallest. This is a recursive defintion. Implement Euclid's algorithm in a recursive function. Hint: testing whether two numbers divide each other, and calculating the remainder, can both be done with the modulo operator. This code can be *really* brief.

Exercise 9.5 In the code below, I have implemented a recursive implementation of asking the user for a string, in which only lower case letters may be used. When someone enters a string with an illegal character in it, a recursive call to the function itself will ask for a new string. This looks like it avoids using the loop-and-a-half to ask the user for new inputs on incorrect inputs. While it is always a bad idea to place control over the depth of recursive calls into a user's hands, this implementation actually is not only bad, it is also quite wrong. Can you see what is wrong with it, and how that is caused? (Note: it is not the `letter < 'a' or letter > 'z'` expression, those comparisons are just fine.)

exercise0905.py

```
def get_input( prompt ):
    value = input( prompt )
    for letter in value:
        if letter < 'a' or letter > 'z':
            print( "The character", letter, "is not allowed!")
            value = get_input( prompt ) # DO NOT DO THIS!
    return value

s = get_input( "Give a string of lower case letters: " )
print( "The user entered:", s )
```

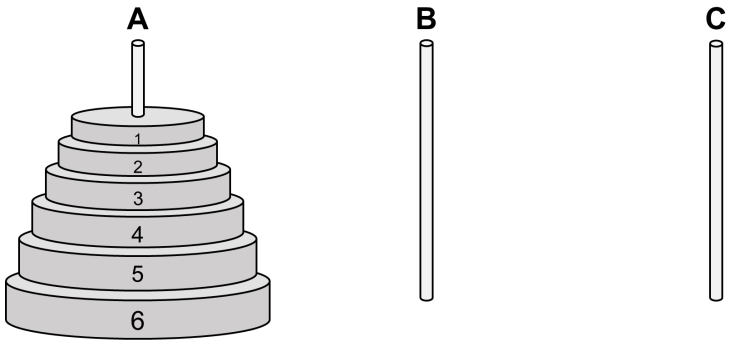
Let me stress once more that the idea above is a *bad* one. You should not use recursion for commonplace problems that can just as well be solved by iterations. Recursion is for exceptional circumstances. Do not see this as an example of recursion, see it as an example of how not to use recursion! The main reason I put it here is that I sometimes observe students writing such code, and I want to make explicit that that is *not a good idea*!

Exercise 9.6 The Towers of Hanoi is a puzzle, which uses three poles, labeled A, B, and C. On pole A there is a stack of discs of varying size; the discs are numbered according to their size. The smallest disc is 1, the next one is 2, the next one is 3, etcetera, up to size N. Typical values for N are 4 and 5, though in the classic puzzle N is supposed to be 64. The

discs are stacked on pole A according to their size, the smallest one on top, and the biggest one on the bottom. You now have to move all the discs from pole A to pole C, whereby you have to follow these four rules: (1) you can only move one disc at a time; (2) you can only move discs between the poles; (3) you can only move a disc from a pole if it is on top, and can only move it to the top of another pole; and (4) you can never place a disc on top of a disc that is smaller. Write a program that solves this puzzle for any value of N (for testing purposes, N should not be chosen higher than 10 or so). Make the program print the solution as a recipe, with lines such as "Move disc 1 from pole A to pole C." At the end, print the number of moves you needed to make, preferably calculated during the process of generating the recipe.

To think about a recursive solution, consider the following: Solving the Towers of Hanoi with a biggest disc of size 10 is easy if you know how to solve it for size 9. Namely, you use your size-9 procedure to move the top 9 discs to the middle pole, then move the disc of size 10 to the target pole, and finally use the size-9 procedure to move the 9 discs from the middle pole to the target pole. But how do you solve the problem with the biggest disc being size 9? Well, that is simple if you know how to solve it for size 8... You can imagine where this is going. You are reducing the complexity of the problem step by step, until you are at "solving the problem for size 2 is simple if you can solve it for size 1." Solving it for size 1 is trivial: you just move the disc to where it must go. Basically, this comes down to a recursive definition of the solution method:

To solve it for size N where you move from pole X to pole Y with Z as temporary pole, you first solve it for size $N-1$ where you move from pole X to pole Z with pole Y as temporary pole, then move the disc of size N from pole X to pole Y, and finally solve the problem for size $N-1$ where you move from pole Z to pole Y with pole X as temporary pole.



Chapter 10

Strings

Until now, most examples and exercises have been using numbers. You might have been wondering by this time if programming is just for number manipulation. In daily life, it is far more commonplace to deal with textual information.

The reason that dealing with texts was postponed until this point, is that dealing with numbers is simply easier than dealing with texts. But in the present chapter, the first steps are taken to learn to manipulate textual information.

Texts, in programming languages, are dealt with in the form of strings. This chapter is on the details of strings, and on readily-available functions to juggle them.

10.1 What you already know about strings

In Chapter 3, strings were quickly introduced. The brief discussion in that chapter ended with the statement that a string is a text, enclosed by either single or double quotes, which might be of any length, including zero characters long. The chapter also explained that you can concatenate two strings using the $+$, and that you can create a string that is the repetition of a shorter string by using a $*$. For example:

```
s1 = "apple"
s2 = 'banana'
print( s1 )
print( s2 )
print( s1 + s2 )
print( 3 * s1 )
print( s2 * 3 )
print( 2 * s1 + 2 * s2 )
```

Chapter 5 introduced the **format()** function to format strings. It also explained how you can get the length of a string using the **len()** function.

String comparisons were explained in Chapter 6, in particular the fact that the comparison operators compare strings using alphabetical rules, whereby capitals are always lower in

the alphabet than lower case letters. This will be explained more in-depth in the present chapter. Chapter 6 also explained how the **in** operator can be used to test the presence of characters or substrings in strings.

Chapter 7 explained how you can use a **for** loop to traverse all the characters in a string.

```
s1 = "orange"
s2 = "banana"
for letter in s1:
    if letter in s2:
        print( s1, "and", s2, "share the letter", letter )
```

10.2 Multi-line strings

Strings in Python may span across multiple lines. This can be useful when you have a very long string, or when you want to format the output of the string in a certain way. Multi-line strings can be achieved in two ways:

- With single or double quotes, and an indication that the remainder of the string continues on the next line with a backslash.
- With triple single or double quotes.

I first demonstrate how this works when you use the regular string enclosure with one double or single quote at each end of the string:

listing1001.py

```
long = "I'm fed up with being treated like sheep. What's the \
point of going abroad if you're just another tourist carted \
around in buses surrounded by sweaty mindless oafs from \
Kettering and Coventry in their cloth caps and their cardigans \
and their transistor radios and their Sunday Mirrors, \
complaining about the tea - 'Oh they don't make it properly \
here, do they, not like at home' - and stopping at Majorcan \
bodegas selling fish and chips and Watney's Red Barrel and \
calamaris and two veg and sitting in their cotton frocks \
squirting Timothy White's suncream all over their puffy raw \
swollen purulent flesh 'cos they 'overdid it on the first day.'"
print( long )
```

As you can see when you run the code, Python interprets this example as a single line of text. The backslash (\) can actually be included after any Python statement to indicate that it continues on the next line, and it can be quite useful for that, for instance when you write long calculations.

The recommended way to write multi-line strings in Python is, however, to use triple double or single quotes. I indicated earlier that you can use those to write multi-line comments. Such comments are basically large strings in the middle of your Python program, which do nothing as they are not assigned to a variable.

Here is an example of a long string with triple double quotes:

listing1002.py

```
long = """And being herded into endless Hotel Miramars and
Bellevueses and Continentales with their modern international
luxury roomettes and draught Red Barrel and swimming pools full
of fat German businessmen pretending they're acrobats forming
pyramids and frightening the children and barging into queues
and if you're not at your table spot on seven you miss the bowl
of Campbell's Cream of Mushroom soup, the first item on the menu
of International Cuisine, and every Thursday night the hotel has
a bloody cabaret in the bar, featuring a tiny emaciated dago with
nine-inch hips and some bloated fat tart with her hair brylcreemed
down and a big arse presenting Flamenco for Foreigners."""
print( long )
```

Note that in the first example the string was interpreted as a continuous series of characters, while in the second example the lines are all printed as different lines. The reason is that in the second example an invisible character is included at the end of each line which indicates that Python should move to the next line before continuing. This is a so-called "newline" character, and you can actually insert it explicitly into a string, using the code `"\n"`. This code should not be read as a backslash and the `"n"`, but as a single newline character. By using it, you can ensure that you print the output on multiple lines, even if you use the backslash to indicate the continuation of the string, as was done in the first example:

listing1003.py

```
long = "And then some adenoidal typists from Birmingham with\n\
flabby white legs and diarrhoea trying to pick up hairy bandy-\n\
legged wop waiters called Manuel and once a week there's an\n\
excursion to the local Roman Ruins to buy cherryade and melted\n\
ice cream and bleeding Watney's Red Barrel and one evening you\n\
visit the so called typical restaurant with local colour and\n\
atmosphere and you sit next to a party from Rhyl who keep\n\
singing 'Torremolinos, torremolinos' and complaining about the\n\
food - 'It's so greasy here, isn't it?' - and you get cornered\n\
by some drunken greengrocer from Luton with an Instamatic\n\
camera and Dr. Scholl sandals and last Tuesday's Daily Express\n\
and he drones on and on and on about how Mr. Smith should be\n\
running this country and how many languages Enoch Powell can\n\
speak and then he throws up over the Cuba Libres."
print( long )
```

If you do not want automatic newline characters in a multi-line string, you have to use the backslash at the end of the line. Otherwise, the second approach is the easiest to read.

10.3 Escape sequences

`"\n"` is a so-called "escape sequence." An escape sequence is a string character written as a backslash followed by a code, which can be one or multiple characters. Python interprets escape sequences in a string as a special character.

Besides the newline character `"\n"`, in Chapter 3 I also introduced the special characters `"\'"` and `"\""`, which can be used to place a single respectively double quote in a string, regardless of what characters surround the string. I also mentioned that you can use `"\\"` to insert a "real" backslash in a string.

Besides these, there are a few more escape sequences which lead to a special character. Most of these are archaic and you do not need to worry about them. The two I want to mention are `"\t"` which represents a single tabulation, and `"\x`*nn*`"` whereby *nn* stands for two hexadecimal digits, which represents the character with hexadecimal number *nn*. For example, `"\x20"` is the character expressed by the hexadecimal number 20, which is the same as the decimal number 32, which is the space (this will be explained later in this chapter).

In case you never learned about hexadecimal counting: hexadecimals use a numbering scheme that uses 16 different digits, namely 0 to 9, and A to F. A direct translation from hexadecimals to decimals turns A into 10, B into 11, etcetera. In decimal counting, the value of a multi-digit number is found by multiplying the digits by increasing powers of 10, from right to left, e.g., the number 1426 is $6 + 2 * 10 + 4 * 100 + 1 * 1000$. For hexadecimal numbers you do the same thing, but multiply by powers of 16, e.g., the hexadecimal number $4AF2$ is $2 + 15 * 16 + 10 * 256 + 4 * 4096$. Programmers tend to like hexadecimal numbers, as computers work with bytes as the smallest unit of memory storage, and a byte can store 256 different values, i.e., any byte value can be expressed by a hexadecimal number of two digits.

Why it is useful to know about hexadecimal counting and hexadecimal representation of characters follows later in the book.

10.4 Accessing characters of a string

As I showed several times before, a string is a collection of characters in a specific order. You can access the individual characters of a string using indices.

10.4.1 String indices

Each symbol in a string has a position, this position can be referred to by the index number of the position. The index numbers start at 0 and then increase to the length of the string. The following table shows the word "python" in the first row and the indices for each letter in the second and third rows:

```
 p  y  t  h  o  n
 0  1  2  3  4  5
-6 -5 -4 -3 -2 -1
```

As you can see, you can use positive indices, which start at the first letter of the string and increase until the end of the string is reached, or negative indices, which start with -1 for the last letter of the string and decrease until the first letter of the string is reached.

As the length of a string `s` is **`len`**`(s)`, the last letter of the string has index **`len`**`(s)-1`. With negative indices, the first letter of the string has index `-`**`len`**`(s)`.

If a string is stored in a variable, the individual letters of the string can be accessed by the variable name and the index of the requested letter between square brackets ([]) next to it.

```
fruit = "orange"
print( fruit[1] )
print( fruit[2] )
print( fruit[3] )
print( fruit[-2] )
print( fruit[-6] )
print( fruit[0] )
print( fruit[-3] )
```

You can also use variables as indices, and even calculations or function calls. You must make sure, however, that calculations result in integers, because you cannot use floats as indices. Below are some examples, most of which are so convoluted that I do not see any reason to incorporate them like this in a program. But they show what is possible.

```
from math import sqrt

fruit = "orange"
x = 3

print( fruit[3-2] )
print( fruit[int( sqrt( 4 ) )] )
print( fruit[2**2] )
print( fruit[int( (x-len( fruit ))/3 )] )
print( fruit[-len( fruit )])
print( fruit[-x] )
```

In principle, you can also use an index with the actual string rather than a variable that contains it, e.g., `"orange"[2]` is the letter `"a"`. For obvious reasons no one ever does that, though.

Besides using single indices you can also access a substring (also called a "slice") from a string by using two numbers between the square brackets with a colon (:) in between. The first of these numbers is the index where the substring starts, the second where it ends. The substring does not include the letter at the second index. By leaving out the left number you indicate that the substring starts at the beginning of the string (i.e., at index 0). By leaving out the right number you indicate that the substring ranges up to and includes the last character of the string.

If you try to access a character using an index that is beyond the reaches of a string, you get a runtime error ("index out of bounds"). For a range of indices to access substrings such limitations do not exist; you can use numbers that are outside the bounds of the string.

```
fruit = "orange"
print( fruit[:] )
print( fruit[0:] )
print( fruit[:6] )
print( fruit[:100] )
print( fruit[:len( fruit )] )
print( fruit[1:-1] )
print( fruit[2], fruit[1:6] )
```

10.4.2 Traversing strings

I already explained how you can traverse the characters of a string using a **for** loop:

```
fruit = 'apple'
for char in fruit:
    print( char, '- ', end='' )
```

Now you know about indices, you probably realize you can also use those to traverse the characters of a string:

listing1004.py

```
fruit = 'apple'

for i in range( 0, len( fruit ) ):
    print( fruit[i], "- ", end="" )
print()

i = 0
while i < len( fruit ):
    print( fruit[i], "- ", end="" )
    i += 1
```

If you just want to traverse the individual characters of a string, the first method, using **for** <character> **in** <string>:, is by far the most elegant and readable. However, occasionally you have to solve problems in which you might prefer one of the other methods.

Exercise Write code that for a string prints the indices of all of its vowels (a, e, i, o, and u). This can be done with a **for** loop or a **while** loop, though the **while** loop is more suitable.

Exercise Write code that uses two strings. For each character in the first string that has exactly the same character at the same index in the second string, you print the character and the index. Watch out that you do not get an "index out of bounds" runtime error. Test it with the strings "The Holy Grail" and "Life of Brian".

Exercise Write a function that takes a string as argument, and creates a new string that is a copy of the argument, except that every non-letter is replaced by a space (e.g., "ph@t l00t" is changed to "ph t l t"). To write such a function, you will start with an empty string, and traverse the characters of the argument one by one. When you encounter a character that is acceptable, you add it to the new string. When it is not acceptable, you add a space to the new string. Note that you can check whether a character is acceptable by simple comparisons. For example, any lower case letter can be found using the test **if** ch >= 'a' **and** ch <= 'z':.

10.4.3 Extended slices

Slices in python can take a third argument, which is the step size (or "stride") that is taken between indices. It is similar to the third argument for the **range()** function. The format for slices then becomes <string>[<begin>:<end>:<step>]. By default the step size is 1.

The most common use for the step size is to use a negative step size in order to create a reversed version of a string.

```
fruit = "banana"
print( fruit[::2] )
print( fruit[1::2] )
print( fruit[::-1] )
print( fruit[::-2] )
```

Reversing a string using `[::-1]` is conceptually similar to traversing the string from the last character to the beginning of the string using backward steps of size 1.

```
fruit = "banana"
print( fruit[::-1] )
for i in range( 5, -1, -1 ):
    print( fruit[i] )
```

10.5 Strings are immutable

A core property of strings is that they are immutable. This means that they cannot be changed. For instance, you cannot change a character of a string by assigning a new value to it. As a demonstration, the following code leads to a runtime error if you try to run it:

```
fruit = "oringe"
fruit[2] = "a"  # Runtime error!
print( fruit )
```

If you want to make a change to a string, you have to create a new string that contains the change; you can then assign the new string to the existing variable if you want. For instance:

```
fruit = "oringe"
fruit = fruit[:2] + "a" + fruit[3:]
print( fruit )
```

The reasons for why strings are immutable are beyond the scope of this book. Just remember that if you want to modify a string you need to overwrite the entire string, and you cannot modify individual indices.

10.6 `string` methods

There is a collection of methods that are designed to operate on strings. All of these methods are applied to a string to perform some operation. Since strings are immutable, they never change the string they work on, but they always return a changed version of the string.

Like the **format()** method introduced in Chapter 5, all these methods are called using the syntax <string>.<method>(), i.e., you have to write the string that they work on before the method call, with a period in between. You will encounter this more often, and why this is implemented in this way will be explained later in the course, in the chapters about object orientation.

Most of these methods are not part of a specific module, but can be called without importing them. There is a `string` module that contains specific constants and methods that can be used in your programs, but the methods I discuss here can all be used without importing the `string` module.

10.6.1 strip()

`strip()` removes from a string leading and trailing spaces, including leading and trailing newlines and other characters that may be viewed as spaces. If instead of spaces, you want to remove different characters, you can add a string parameter that consists of all characters you want to be removed.

```
s = "    And now for something completely different \n    "
print( "["+s+"]" )
s = s.strip()
print( "["+s+"]" )
```

10.6.2 upper() and lower()

`upper()` creates a version of a string of which all letters are capitals. `lower()` is equivalent, but uses only lower case letters. Neither method uses parameters.

```
s = "The Meaning of Life"
print( s )
print( s.upper() )
print( s.lower() )
```

10.6.3 find()

`find()` can be used to search in a string for the starting index of a particular substring. As parameters it gets the substring, and optionally a starting index to search from, and an ending index. It returns the lowest index where the substring starts, or -1 if the substring is not found.

```
s = "Humpty Dumpty sat on the wall"
print( s.find( "sat" ) )
print( s.find( "t" ) )
print( s.find( "t", 12 ) )
print( s.find( "q" ) )
```

10.6.4 replace()

replace() replaces all occurrences of a substring with another substring. As parameters it gets the substring to look for, and the substring to replace it with. Optionally, it gets a parameter that indicates the maximum number of replacements to be made.

I must stress again that strings are immutable, so the replace() function is not actually changing the string. It returns a new string that is a copy of the string with the replacements made.

```
s = 'Humpty Dumpty sat on the wall'
print( s.replace( 'sat on', 'fell off' ) )
```

10.6.5 split()

split() splits a string up in words, based on a given character or substring which is used as separator. The separator is given as the parameter, and if no separator is given, the white space is used, i.e., you split a string in the actual words (though punctuation attached to words is considered part of the words). If there are multiple occurrences of the separator next to each other, the extra ones are ignored (i.e., with the white space as separator, it does not matter if there is a single white space between two words, or multiple).

The result of this split is a so-called "list" of words. Lists are discussed in a coming chapter, so for now I will not say much about them. I just indicate that if you want to access the separate words, you can use the **for** <word> **in** <list>: construction.

```
s = 'Humpty Dumpty      sat    on the wall   '
wordlist = s.split()
for word in wordlist:
    print( word )
```

A very useful property of splitting is that we can decode some basic file formats. For example, a comma separated value (CSV) file is a very simple format, of which the basic setup is that each line consists of values that are separated by a comma. These values can be split from each other using the split() method.[6]

```
csv = "2016,September,28,Data Processing,Tilburg University"
values = csv.split( ',' )
for value in values:
    print( value )
```

10.6.6 join()

join() is the opposite of split(). join() joins a list of words together, separated by a specific separator. This sounds like it would be a method of lists, but for historic reasons

[6] In actuality it will be a bit more convoluted as there might be commas in the fields that are stored in the CSV file, so it depends a bit on the contents of the file whether this simple approach will work. More on CSV files will be said in Chapter 26.

it is defined as a method for strings. Since all string methods are called using the format `<string>.<method>()`, there must be a string in front of the call to `join()`. That string is the separator that you want to use, while the parameter of the method is the list that you want to join together. The return value, as always, is the resulting string.

```
s = "Humpty;Dumpty;sat;on;the;wall"
wordlist = s.split( ';' )
s = " ".join( wordlist )
print( s )
```

10.6.7 Practice

Exercise In the string `"How much woot would a wootchuck chuck if a wootchuck could chuck woot."` the word `"wood"` is misspelled. Use `replace()` to replace all occurrences of this spelling error with the correct spelling.

Exercise Display the contents of the string `"Nobody expects the Spanish Inquisition!# In fact, those who do expect the Spanish Inquisition..."` up to, but not including, the hash mark (#). Use `find()` to get the index of the hash mark.

Exercise Write a program that prints a "cleaned" version of all the words in a string. Everything that is not a letter should be removed and be considered a separator. All the letters should be lower case. For example, the string `"I'm sorry, sir."` should produce four words, namely `"i"`, `"m"`, `"sorry"`, and `"sir"`. You can use the function for string cleaning which you wrote as an exercise above.

10.7 Character encoding

All systems use a particular way of encoding characters. The basic encoding that (almost) every system supports is the standard ASCII code. This is a 7-bits code, which can represent 128 different characters. Several of these (in particular those with the lowest-numbered encodings) are control characters that have a special function. Most of these special functions are only useful for archaic computer systems, but the tabulation, newline, and backspace characters are found amongst them. If you only use characters on a standard US keyboard, you are limited to ASCII characters.

Nowadays, many systems use Unicode. Unicode supports far more characters. There are different formats for storing characters in Unicode. The best-known is UTF-8, which uses one byte for each of the ASCII characters, but multiple bytes for all the other characters (a byte is a group of 8 bits, whereby each bit contains either a 1 or a zero). Other Unicode encodings use multiple bytes to store any character. Python, be default, works with UTF-8, which means that it also supports regular ASCII encodings.

10.7.1 ASCII

Below I display the ASCII table. The only characters I have left off are those which are control sequences. These have the numbers zero to 31, and 127. 32 is the space. I also

display the hexadecimal code for each character next to the decimal code. (If you wonder why I even bother listing those hexadecimal codes: they become relevant in a later chapter.)

As you can see, each character has a number attached to it. To find out what a character's number is in a program, you can use the **ord**() function. For instance, **ord**("A") returns the number of "A", which, as you can see, is 65. The counterpart of the **ord**() function is the **chr**() function. **chr**() gets a number as argument, and returns the character that belongs to that number. For instance, **chr**(65) is the letter "A".

```
DC HX      DC HX      DC HX      DC HX      DC HX      DC HX
32 20      48 30 0    64 40 @    80 50 P    96 60 `   112 70 p
33 21 !    49 31 1    65 41 A    81 51 Q    97 61 a   113 71 q
34 22 "    50 32 2    66 42 B    82 52 R    98 62 b   114 72 r
35 23 #    51 33 3    67 43 C    83 53 S    99 63 c   115 73 s
36 24 $    52 34 4    68 44 D    84 54 T   100 64 d   116 74 t
37 25 %    53 35 5    69 45 E    85 55 U   101 65 e   117 75 u
38 26 &    54 36 6    70 46 F    86 56 V   102 66 f   118 76 v
39 27 '    55 37 7    71 47 G    87 57 W   103 67 g   119 77 w
40 28 (    56 38 8    72 48 H    88 58 X   104 68 h   120 78 x
41 29 )    57 39 9    73 49 I    89 59 Y   105 69 i   121 79 y
42 2A *    58 3A :    74 4A J    90 5A Z   106 6A j   122 7A z
43 2B +    59 3B ;    75 4B K    91 5B [   107 6B k   123 7B {
44 2C ,    60 3C <    76 4C L    92 5C \   108 6C l   124 7C |
45 2D -    61 3D =    77 4D M    93 5D ]   109 6D m   125 7D }
46 2E .    62 3E >    78 4E N    94 5E ^   110 6E n   126 7E ~
47 2F /    63 3F ?    79 4F O    95 5F _   111 6F o
```

A comparison of strings which use only these characters use the numbers of the characters to determine which string is "smaller." For instance, the string "orange" is smaller than the string "ordinal", because the first character that differs between them is the third one, which is "a" for "orange" and "d" for "ordinal", and since the number for "a" is lower than the number for "d", the string "orange" is considered to be smaller than the string "ordinal". This is, basically, an alphabetic comparison. If characters occur in a string that are not letters, you can check in the ASCII table which is considered lower. Notice how all the digits are lower than letters.

```
print( ord( 'A' ) )
print( ord( 'a' ) )
print( chr( 65 ) )
print( chr( 97 ) )
print( "orange" < "ordinal" )
```

You can use these numbers that are associated with characters to do all kinds of neat calculations. For instance, if I want to know which the twelfth letter after "g" is, I can calculate that as follows:

```
print( "The 12th letter after g is", chr( ord( "g" )+12 ) )
```

For another example of what you can do with character codes, here is a program that generates the ASCII table as a matrix:

listing1005.py

```
print( ' ', end='' )
for i in range(16):
    if i < 10:
        print( ' '+chr( ord( '0' )+i ), end='' )
    else:
        print( ' '+chr( ord( 'A' )+i-10 ), end='' )
print()
for i in range( 2, 8 ):
    print( i, end='' )
    for j in range( 16 ):
        c = i*16+j
        print( ' '+chr( c ), end='' )
    print()
```

Note that I highly prefer you using the **ord**() and **chr**() functions if you want to juggle character encoding. If you want to refer to the character code of the letter "A", do not write 65, but write **ord**("A") instead. 65 is only meaningful to people who know ASCII encodings, and your programs should be meaningful to anybody. Moreover, while ASCII is a widely-used standard, there are still computers out there which use different encoding mechanisms, in which the code for "A" is not necessarily 65 (I am looking at you, IBM).

10.7.2 UTF-8

Python supports Unicode, in particular the most common Unicode encoding scheme UTF-8. This means that you can use all kinds of "weird" characters. I explained that in the naming of functions and variables you can use "letters," which you probably assumed meant "A" to "Z" and "a" to "z". The funny thing is that it depends on the language codes of your computer what is considered a letter. For instance, if your computer tells Python that the language is German, then you can also use characters with umlauts. I strongly discourage using such letters in variable and function names, by the way. Not only are they hard to type, but they also make your program less portable.

In UTF-8, the regular characters which you find on a keyboard are represented in strings exactly as you would expect. However, "special" characters can be incorporated too, but look quite different. Since Python supports UTF-8, you have to be careful when you copy texts from, for instance, a word processor document. Word processors have the disturbing habit of changing characters into other characters, like turning straight quotes into round quotes. If you copy such round quotes into your program, Python will accept the characters, but will not interpret them as, for instance, string boundaries.

If you want to display Unicode characters, you can do so by using Unicode encodings. You have to know the UTF-8 number of the character that you want to display. If you know that, you can use a code \u*xxxx*, where *xxxx* is a hexadecimal number, to incorporate a Unicode character in a string. For example, the code below displays the capitals of the Greek alphabet:[7]

[7]There is one weird character in this display, between the Rho and the Sigma, which is \u03A2, which is evidently not a legal Unicode character.

```
alpha = "\u0391"
for i in range( 25 ):
    print( chr( ord( alpha )+i ), end=" " )
```

In general, you will not need to worry too much about character encodings. I recommend that you restrict yourself to ASCII whenever possible. In cases where you have to deal with Unicode characters, things usually work correctly automatically, since the standard Python functionalities support Unicode. Occasionally I have run into translation problems from Unicode to ASCII, in particular where files were concerned. It will be a while before you run into problems like that, and I will bring it up again in Chapter 16 and later.

What you learned

In this chapter, you learned about:

- Strings
- Multi-line strings
- Accessing string characters with positive and negative indices
- Slices
- Immutability of strings
- `strip()`, `upper()`, `lower()`, `find()`, `replace()`, `split()`, and `join()`
- Escape sequences
- ASCII and UTF-8 encodings

Exercises

Exercise 10.1 Count how many of each vowel (a, e, i, o, u) there are in a text string, and print the count for each vowel with a single formatted string. Remember that vowels can be both lower and uppercase.

Exercise 10.2 Below is a text with several characters enclosed in square brackets [and]. Scan the text and print out all characters which are between square brackets.

exercise1002.py

```
text = """And sending tinted postcards of places they don't
realise they haven't even visited to 'All at nu[m]ber 22, weather
w[on]derful, our room is marked with an 'X'. Wish you were here.
Food very greasy but we've found a charming li[t]tle local place
hidden awa[y ]in the back streets where they serve Watney's Red
Barrel and cheese and onion cris[p]s and the accordionist pla[y]s
"Maybe i[t]'s because I'm a Londoner"' and spending four days on
the tarmac at Luton airport on a five-day package tour wit[h]
n[o]thing to eat but dried Watney's sa[n]dwiches..."""
```

Exercise 10.3 Print a line of all the capital letters `"A"` to `"Z"`. Below it, print a line of the letters that are 13 positions in the alphabet away from the letters that are above them. E.g., below the `"A"` you print an `"N"`, below the `"B"` you print an `"O"`, etcetera. You have to consider the alphabet to be circular, i.e., after the `"Z"`, it loops back to the `"A"` again.

Exercise 10.4 In the text below, count how often the word "wood" occurs (using program code, of course). Capitals and lower case letters may both be used, and you have to consider that the word "wood" should be a separate word, and not part of another word. Hint: If you did the exercises from this chapter, you already developed a function that "cleans" a text. Combining that function with the `split()` function more or less solves the problem for you.

exercise1004.py

```
text = """How much wood would a woodchuck chuck
If a woodchuck could chuck wood?
He would chuck, he would, as much as he could,
And chuck as much as a woodchuck would
If a woodchuck could chuck wood."""
```

Exercise 10.5 Write a program that takes a string and produces a new string that contains the exact characters that the first string contains, but in order of their ASCII-codes. For instance, the string `"Hello, world!"` should be turned into `" !,Hdellloorw"`. This is relatively easy to do with list functions, which will be introduced in a future chapter, but for now try to do it with string manipulation functions alone.

Exercise 10.6 Typical autocorrect functions are the following: (1) if a word starts with two capitals, followed by a lower-case letter, the second capital is made lower case; (2) if a sentence contains a word that is immediately followed by the same word, the second occurrence is removed; (3) if a sentence starts with a lower-case letter, that letter is turned into a capital; (4) if a word consists entirely of capitals, except for the first letter which is lower case, then the case of the letters in the word is reversed; and (5) if the sentence contains the name of a day (in English) which does not start with a capital, the first letter is turned into a capital. Write a program that takes a sentence and makes these autocorrections. Test it out on the string below.

exercise1006.py

```
sentence = "as it turned out our chance meeting with REverend \
aRTHUR BElling was was to change our whole way of life, and \
every sunday we'd hurry along to St lOONY up the Cream BUn \
and Jam..."
```

Chapter 11

Tuples

A tuple is a group of one or more values that are treated as a whole. This chapter explains how to recognize and use tuples.

11.1 Using tuples

A tuple is a group of one or more values, separated by commas. Normally, tuples are written with parentheses around them, but the parentheses are not actually necessary (except in circumstances where otherwise confusion would arise). For example:

```
t1 = ("apple", "orange")
print( type( t1 ) )
t2 = "banana", "cherry"
print( type( t2 ))
```

You can mix data types within tuples. You can even put tuples in tuples.

```
t1 = ("apple", 3, 1.4)
t2 = ("apple", 3, 1.4, ("banana", 5))
```

To find out how many elements a tuple contains, you can use the **len()** function.

```
t1 = ("apple", "orange")
t2 = ("apple", 3, 1.4)
t3 = ("apple", 3, 1.4, ("banana", 5))
print( len( t1 ) )
print( len( t2 ) )
print( len( t3 ) )
```

Note that in this example, the length of `t3` is 4, and not 5. The last element of `t3` is the tuple `("banana", 5)`, which counts as one element.

You can use a **for** loop to access individual elements of a tuple in sequence.

```
t1 = ("apple", 3, 1.4, ("banana", 5))
for element in t1:
    print( element )
```

You can also use the **max**() and **min**() functions to get the maximum respectively the minimum from a tuple of numbers. You can sum the elements of a tuple using the **sum**() function.

```
t1 = (327, 419, 101, 667, 925, 225)
print( max( t1 ) )
print( min( t1 ) )
print( sum( t1 ) )
```

You can test whether an element is part of a tuple by using the **in** operator.

```
t1 = ("apple", "banana", "cherry")
print( "banana" in t1 )
print( "orange" in t1 )
```

11.1.1 Tuple assignments

As you have seen, you can create a tuple by assigning comma-separated values to a variable. Parentheses around it are optional. What if you want to create a tuple with only one element?

```
t1 = ("apple")
print( type( t1 ) )
```

If you run this code, you will find that `t1` is of the class `str`, i.e., a string. Putting parentheses around the element does not work, as parentheses are optional. Python introduced a little trick to create a tuple with only one element, and that is that you indicate that it is a tuple by placing a comma after the value. This is rather unintuitive and I would even say "degenerate," but historically this was the solution that an early version of Python introduced, and for compatibility reasons it was not changed.

```
t1 = ("apple",)
print( type( t1 ) )
print( len( t1 ) )
```

Python allows you to place a tuple left of the assignment operator. This is an exception to the rule that only one variable can be placed left of an assignment. The values at the right side are copied one-by-one to the left side, left to right.

```
t1, t2 = "apple", "banana"
print( t1 )
print( t2 )
```

You can place parentheses around the values at the right side, and/or parentheses around the variables at the left side, which makes no difference.

If you place more variables at the left side than values at the right side, you get a runtime error. The same for placing fewer (unless you place just one, as shown above). However, you can create tuples at the right side by placing parentheses.

```
t1, t2 = ("apple", "banana"), "cherry"
print( t1 )
print( t2 )
```

11.1.2 Tuple indices

Just like with strings, you can access the individual elements of a tuple using indices. Where with strings the individual elements are characters, for tuples they are the values. For instance:

```
t1 = ("apple", "banana", "cherry", "durian")
print( t1[2] )
```

You can even use slices, with the same rules as for strings (if you do not remember, check Chapter 10 again). A slice of a tuple is another tuple. For example:

```
t1 = ("apple", "banana", "cherry", "durian", "orange")
print( t1[1:4] )
```

Since tuples are indexed, an alternative for a **for** loop to access the individual elements of a tuple is to loop over the indices.

listing1101.py

```
t1 = ("apple", "banana", "cherry", "durian", "orange")
i = 0
while i < len( t1 ):
    print( t1[i] )
    i += 1
```

Exercise Write a **for** loop that displays all the values of the elements of a tuple, and also displays their index.

11.1.3 Tuple comparisons

You can compare two tuples with each other by using the regular comparison operators. These operators first compare the first two elements of the tuples. If these are different, then the comparison will determine which one is "smaller" based on the rules for these data types, and result in **True** or **False**. If they are equal, the second elements will be compared, etcetera.

listing1102.py

```
t1 = ( "apple", "banana" )
t2 = ( "apple", "banana" )
t3 = ( "apple", "cherry" )
t4 = ( "apple", "banana", "cherry" )
print( t1 == t2 )
print( t1 < t3 )
print( t1 > t4 )
print( t3 > t4 )
```

11.1.4 Tuple return values

In Chapter 8, you learned that functions can return multiple values. If you code something like that, what actually happens is that the function is returning a tuple. To deal with such return values, you assign them to variables as explained under "tuple assignments" above.

11.2 Tuples are immutable

Just like strings, tuples are immutable. This means that you cannot assign a new value to one element of a tuple. The example below will produce a runtime error when run.

```
t1 = ("apple", "banana", "cherry", "durian")
t1[0] = "orange"
```

11.3 Applications of tuples

Tuples are not used often in Python code (except as return values of functions). A logical application of tuples would be to deal with values that always occur in small collections. However, object orientation (Chapter 20 and further) offers many tools and techniques to deal with such small collections, which means that programmers usually revert to object orientation when they need something like that.

For the moment, here is an example of the use of tuples in an application. Suppose that you have to write a program that deals with geometric figures in 2-dimensional space. A concept that you need is that of a point: a location in 2D space that is identified by two coordinates. Rather than write functions that always require a separate X-coordinate and a separate Y-coordinate, you can specify that coordinates are always communicated in the form of tuples.

listing1103.py

```
from math import sqrt

# Returns the distance between two points in 2-dimensional space.
# The points are the parameters of the function; each point is a
# tuple of two numeric values.
```

```
def distance( p1, p2 ):
    return sqrt( (p1[0] - p2[0])**2 + (p1[1] - p2[1])**2 )

point1 = (1,2)
point2 = (5,5)
print( "Distance between", point1, "and", point2, "is",
    distance( point1, point2 ) )
```

An advantage of using tuples to communicate coordinates is that it is relatively easy to write functions that can deal with coordinates in higher-dimensional spaces too.

listing1104.py

```
from math import sqrt

# Distance between two points in N-dimensional space.
# The points should have the same dimension, i.e., they are tuples
# of numeric values, and they should have the same length.
def distance( p1, p2 ):
    total = 0
    for i in range( len( p1 ) ):
        total += (p1[i] - p2[i])**2
    return sqrt( total )

# 1-dimensional space
point1 = (1,)
point2 = (5,)
print( "1D: Distance between", point1, "and", point2, "is",
    distance( point1, point2 ) )

# 2-dimensional space
point1 = (1,2)
point2 = (5,5)
print( "2D: Distance between", point1, "and", point2, "is",
    distance( point1, point2 ) )

# 3-dimensional space
point1 = (1,2,4)
point2 = (5,5,8)
print( "3D: Distance between", point1, "and", point2, "is",
    distance( point1, point2 ) )
```

What you learned

In this chapter, you learned about:

- Tuples
- Tuple assignments

- Tuple indices
- Immutability of tuples
- Applications of tuples

Exercises

Exercise 11.1 A complex number is a number of the form `a + b`i, whereby `a` and `b` are constants, and i is a special value that is defined as the square root of -1. Of course, you never try to actually calculate what the square root of -1 is, as that gives a runtime error; in complex numbers, you always let the i remain. For instance, the complex number `3 + 2`i cannot be simplified any further. Addition of two complex numbers `a + b`i and `c + d`i is defined as `(a + c) + (b + d)`i. Represent a complex number as a tuple of two numeric values, and create a function that calculates the addition of two complex numbers.[8]

Exercise 11.2 Multiplication of two complex numbers `a + b`i and `c + d`i is defined as `(a*c - b*d) + (a*d + b*c)`i. Write a function that calculates the multiplication of two complex numbers.

Exercise 11.3 Consider the definition of a new datatype. The new datatype is the `inttuple`. An `inttuple` is defined as being either an integer, or a tuple consisting of `inttuples`. You see an example of an `inttuple` in the code block below. Write a function that prints all the integer values stored in an `inttuple`. Hint: Since the `inttuple` is defined recursively, a recursive function is probably the right approach. If you skipped Chapter 9, you probably should skip this exercise too. Use the **`isinstance`**`()` function (explained in Chapter 8) to determine whether you are dealing with an integer or a tuple. If you do this correctly, for the `inttuple` given below, the function will print the numbers 1 to 20 sequentially.

exercise1103.py

```
inttuple = ( 1, 2, ( 3, 4 ), 5, ( ( 6, 7, 8, ( 9, 10 ), 11 ), 12,
    13 ), ( ( 14, 15, 16 ), ( 17, 18, 19, 20 ) ) )
```

[8] Actually, Python supports a separate data type `complex` that represents complex numbers, so there is not really a need to deal with complex numbers as tuples, but for the purpose of practicing with tuples this exercise works fine.

Chapter 12

Lists

Lists are ordered collections of data items, just like tuples. The major difference with tuples is that lists are mutable. This makes them a highly flexible data structure, that you will find many uses for.

12.1 List basics

A list is a collection of elements.

The elements of a list are *ordered*. Because they are ordered, you can access each of the elements of a list using an index, just like you can access the characters of a string, and just like you can access the elements of a tuple. Indices start at zero, just as with strings.

In Python, lists are recognizable from the fact that they enclose their elements in square brackets ([]). You can get the number of elements in a list by using the **`len`**`()` function. You can use a **`for`** loop to traverse the elements of a list. You can mix data types in a list. You can apply the **`max`**`()`, **`min`**`()` and **`sum`**`()` functions to a list. You can test for the existence of an element in a list using the **`in`** operator (or for the non-existence by using **`not in`**).

listing1201.py

```
fruitlist = ["apple", "banana", "cherry", 27, 3.14]
print( len( fruitlist ) )
for element in fruitlist:
    print( element )
print( fruitlist[2] )

numlist = [314, 315, 642, 246, 129, 999]
print( max( numlist ) )
print( min( numlist ) )
print( sum( numlist ) )
print( 100 in numlist )
print( 999 in numlist )
```

Exercise Write a **while** loop to print the elements of a list.

Apart from the square brackets, lists seem to be a lot like tuples. Yet there is a big difference...

12.2 Lists are mutable

Because lists are mutable, you can change the contents of a list.

To overwrite an element of a list, you can assign a new value to it.

```
fruitlist = ["apple", "banana", "cherry", "durian", "orange"]
print( fruitlist )
fruitlist[2] = "strawberry"
print( fruitlist )
```

You can also overwrite list slices by assigning a new list to the slice. The slice you remove need not be of equal length to the new list you insert.

```
fruitlist = ["apple", "banana", "cherry", "durian", "orange"]
print( fruitlist )
fruitlist[1:3] = ["raspberry", "strawberry", "blueberry"]
print( fruitlist )
```

You can insert new elements into a list by assigning them to an empty slice.

```
fruitlist = ["apple", "banana", "cherry", "durian", "orange"]
print( fruitlist )
fruitlist[1:1] = ["raspberry", "strawberry", "blueberry"]
print( fruitlist )
```

You can delete elements from a list by assigning an empty list to a slice.

```
fruitlist = ["apple", "banana", "cherry", "durian", "orange"]
print( fruitlist )
fruitlist[1:3] = []
print( fruitlist )
```

Using slices and assignments, you can adapt a list in any way that you like. However, it is easier to change lists using methods. There are many helpful methods available, which I am going to discuss below.

Exercise Change a list that contains only words (you can take one of the `fruitlists` above) by turning every word in the list into a word consisting of only capitals. At this point in the book, the way to do that is by using a **`while`** loop that uses a variable `i` that starts at 0 and runs up to **`len`**`(<list>)-1`. Use `i` as an index for this list.

12.3 Lists and operators

Lists support the use of the operators $+$ and $*$. These operators work similar as to how they work for strings.

You can add two lists together with the $+$ operator, the result of which is a list which contains the elements of both lists involved. Of course, you have to assign the result to a variable to store it.

You can multiply a list by a number to create a list that contains the elements of the original list, repeated as often as the number indicates. This can be a fast approach to create a list with all equal elements.

```
fruitlist = ["apple", "banana"] + ["cherry", "durian"]
print( fruitlist )
numlist = 10 * [0]
print( numlist )
```

Note: With the $+$ you can add a list to another list, but you cannot add a new element to a list, unless you turn that new element into a list with a single element by putting two square brackets around it. If you try to add something to a list that is not a list, Python will try to interpret it as a list – if it can do that (which it can, for instance, for a string, which it can consider a list of letters); it will then still do the addition but the result will not be what you want. For instance, the code below tries to add a "cherry" to a list, but only the second addition actually does what is intended.

listing1202.py

```
fruitlist = ["apple", "banana"]
fruitlist += "cherry"
print( fruitlist )

fruitlist = ["apple", "banana"]
fruitlist += ["cherry"]
print( fruitlist )
```

12.4 List methods

Python supports many methods to change lists or get information from them. You do not need to import a module to use them. Since they are methods, you call them using the syntax `<list>.<method>()`.

Important! Lists are mutable and these methods actually change the list! It is not as you are used to with string methods, where the methods create a new string, and return it, while the original string remains. Most list methods have an irrevocable effect on the list they work on. Usually they have no return value, and you do not need one either, as the purpose of the methods is to change the list.

12.4.1 `append()`

`append()` attaches an item at the end of a list. You call the method with the item you wish to add as argument.

```
fruitlist = ["apple", "banana", "cherry", "durian"]
print( fruitlist )
fruitlist.append( "orange" )
print( fruitlist )
```

An alternative for using the `append()` method is to add a list with one new element to the existing list with a +, and assign the resulting list to the original list variable. However, the `append()` method is preferable as it is more readable. `<list>.append(<element>)` is equivalent to `<list>[len(<list>):] = [<element>]`, or simply `<list> += [<element>]`.

12.4.2 `extend()`

`extend()` makes a list longer by appending the elements of another list at the end. You call the method with the list of which you want to add the elements as argument.

```
fruitlist = ["apple", "banana", "cherry", "durian"]
print( fruitlist )
fruitlist.extend( ["raspberry", "strawberry", "blueberry"] )
print( fruitlist )
```

Just as with the `append()` method, you can extend an existing list with a new list by simply using the + operator, and assigning the result to the original list variable. And just as with the `append()` method, the `extend()` method is preferable. `<list>.extend(<addlist>)` is equivalent to `<list>[len(<list>):] = <addlist>`.

12.4.3 `insert()`

`insert()` allows you to insert an element at a specific position in a list. It is called with two arguments, the first being the index of the location where you wish to insert the new element, and the second the new element itself. To insert an element at the front of the list, you can use index 0.

```
fruitlist = ["apple", "banana", "cherry", "durian"]
print( fruitlist )
fruitlist.insert( 2, "orange" )
print( fruitlist )
```

<list>.insert(<i>,<element>) is equivalent to <list>[<i>:<i>] = [<element>].

12.4.4 remove()

remove() allows you to remove an element from a list. The element you wish to remove is given as argument. If the element occurs in the list multiple times, only the first occurrence will be removed. If you try to remove an element that is not on the list, a runtime error is generated.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
print( fruitlist )
fruitlist.remove( "banana" )
print( fruitlist )
```

12.4.5 pop()

Like remove(), pop() removes an element from the list, but does so by index. It has one optional argument, which is the index of the element that you wish to remove. If you do not provide that argument, pop() removes the last element from the list. If the index is beyond the boundaries of the list, pop() generates a runtime error.

A major difference with remove() is that pop() actually has a return value, namely the element that gets removed. This allows you to quickly process all the elements of a list, while emptying the list at the same time.

```
fruitlist = ["apple", "banana", "cherry", "durian"]
print( fruitlist )
print( fruitlist.pop() )
print( fruitlist )
print( fruitlist.pop( 0 ) )
print( fruitlist )
```

12.4.6 del

del is neither a method nor a function, but since it is often metioned in one breath with remove() and pop(), I place it here. **del** is a keyword that allows you to delete a list element, or list slice, by index. It is similar to pop() in functionality, but does not have a return value. Also, pop() cannot be used on slices. To remove one element from a list, use **del** <list>[<index>]. To remove a slice, use **del** <list>[<index1>:<index2>].

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
del fruitlist[3]
print( fruitlist )
```

12.4.7 index()

index() returns the index of the first occurrence on the list of the element that is given to index() as argument. A runtime error is generated if the element is not found on the list.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
print( fruitlist.index( "banana" ) )
```

12.4.8 count()

count() returns an integer that indicates how often the element that is passed to it as an argument occurs in the list.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
print( fruitlist.count( "banana" ) )
```

12.4.9 sort()

sort() sorts the elements of the list, from low to high. If the elements of the list are strings, it does an alphabetical sort. If the elements are numbers, it does a numeric sort. If the elements are mixed, it generates a runtime error, unless certain arguments are given.

listing1203.py

```
fruitlist = ["apple", "strawberry", "banana", "raspberry",
    "cherry", "banana", "durian", "blueberry"]
fruitlist.sort()
print( fruitlist )

numlist = [314, 315, 642, 246, 129, 999]
numlist.sort()
print( numlist )
```

To do a reverse sort (for instance, sorting numerical items from high to low, or sorting strings alphabetically from "z" to "a"), you can add an argument reverse=<boolean>.

```
fruitlist = ["apple", "strawberry", "banana", "raspberry",
    "cherry", "banana", "durian", "blueberry"]
fruitlist.sort( reverse=True )
print( fruitlist )
```

Another argument that you can give `sort()` is a key. You have to provide this argument as `<list>.sort( key=<key> )`, whereby `<key>` is a function that takes one argument (the element that is to be sorted) and returns a value that is used as key. A typical use for the key argument is if you want to sort a list of strings, but want to do the sorting case-insensitively. So as key you want to use the elements, but in lower case, i.e., you want to apply the function **`str`**`.lower()` to the element. You call the `sort()` method as in the following example:

listing1204.py

```
fruitlist = ["apple", "Strawberry", "banana", "raspberry",
    "CHERRY", "banana", "durian", "blueberry"]
fruitlist.sort()
print( fruitlist )
fruitlist.sort( key=str.lower ) # case-insensitive sort
print( fruitlist )
```

Note that for the key argument, you do not place parentheses after the function name. This is not a function call, it is an argument that tells Python which function to use to generate the key. You can write your own function to be used as key. For example, in the code below, `numlist` is sorted with the digits reversed:

listing1205.py

```
def revertdigits( item ):
    return (item%10)*100 + (int(item/10)%10)*10 + int(item/100)

numlist = [314, 315, 642, 246, 129, 999]
numlist.sort( key=revertdigits )
print( numlist )
```

Here is another example, that sorts a list of strings by length, then alphabetical order:

listing1206.py

```
def len_alphabetical( element ):
    return len( element ), element

fruitlist = ["apple", "strawberry", "banana", "raspberry",
    "cherry", "banana", "durian", "blueberry"]
fruitlist.sort( key=len_alphabetical )
print( fruitlist )
```

Note that the `len_alphabetical()` function returns a tuple. When two tuples are compared, first the first elements of both tuples are compared, and if they are equal, the second elements are compared. You can use this knowledge to create a key function which sorts a mixed list, e.g., a list which consists of both strings and numbers. Just let the key function return a tuple of which the first item indicates the type of the element represented by a number, and the second the element itself.

listing1206a.py

```
def mixed_key( element ):
    if isinstance( element, str ):
        return 1, element
    return 0, element

mixedlist = ["apple", 0, "strawberry", 5, "banana", 2, \
"raspberry", 9, "cherry", "banana", 7, 7, 6, "blueberry"]
mixedlist.sort( key=mixed_key )
print( mixedlist )
```

At this point I can give a typical example of the use of "anonymous functions," which I introduced in Chapter 8. Using an anonymous function to specify the key for the `sort()` method keeps the code for the key next to where you call the `sort()`, instead of elsewhere in the program. This may improve readability.

listing1207.py

```
fruitlist = ["apple", "strawberry", "banana", "raspberry",
    "cherry", "banana", "durian", "blueberry"]
fruitlist.sort( key=lambda x: (len(x),x) )
print( fruitlist )
```

12.4.10 `reverse()`

`reverse()` simply puts the elements of the list in reverse order.

```
fruitlist = ["apple","strawberry","banana","raspberry","durian"]
fruitlist.reverse()
print( fruitlist )
```

12.4.11 Practice

Exercise Write a program that asks the user to enter some data, for instance the names of their friends. When the user wants to stop providing inputs, he just presses `Enter`. The program then displays an alphabetically sorted list of the data items entered. Do not just print the list, but print each item separately, on a different line.

Exercise Sort a list of numbers using their absolute values; use the **`abs()`** function as key.

Exercise Count how often each letter occurs in a string (case-insensitively). You can ignore every character that is not a letter. Store the counts in a list of 26 items that all start at zero. Print the resulting counts. As index you can use **`ord`**`(letter) - `**`ord`**`("a")`, where letter is a lower case letter (the **`ord()`** function is explained in Chapter 10).

12.5 Aliasing

If you assign a variable that contains a list to another variable, you might expect that you create a copy of the list in the second variable. But you are not doing that. You are actually creating an *alias* for the list, i.e., a new variable that is referring to the same list. This means that the new variable can be treated as a list, but any change that you make to the list it refers to, is visible in the original list variable, and vice versa. They are not different lists.

listing1208.py

```
fruitlist = ["apple", "banana", "cherry", "durian"]
newfruitlist = fruitlist
print( fruitlist )
print( newfruitlist )
newfruitlist[2] = "orange"
print( fruitlist )
print( newfruitlist )
```

Every variable in Python has an identification number. You can see it with the **id**() function. The ID number indicates which memory spot the variable refers to. For an alias of a list, the ID is the same as for the original list.

listing1209.py

```
fruitlist = ["apple", "banana", "cherry", "durian"]
newfruitlist = fruitlist
print( id( fruitlist ) )
print( id( newfruitlist ) )
```

If you want to create a copy of a list, you can do so using a little trick. Instead of using `<newlist> = <oldlist>`, you use the command `<newlist> = <oldlist>[:]`.

listing1210.py

```
fruitlist = ["apple", "banana", "cherry", "durian"]
newfruitlist = fruitlist
verynewfruitlist = fruitlist[:]

print( id( fruitlist ) )
print( id( newfruitlist ) )
print( id( verynewfruitlist ) )

fruitlist[2] = "orange"
print( fruitlist )
print( newfruitlist )
print( verynewfruitlist )
```

12.5.1 is

The keyword **is** is introduced to compare the identities of two variables.

listing1211.py

```
fruitlist = ["apple", "banana", "cherry", "durian"]
newfruitlist = fruitlist
verynewfruitlist = fruitlist[:]

print( fruitlist is newfruitlist )
print( fruitlist is verynewfruitlist )
print( newfruitlist is verynewfruitlist )
```

As you can see, the keyword **is** manages to determine that `fruitlist` and `newfruitlist` are aliases, but that `verynewfruitlist` is not the same list. If you compare them with the == operator, the results are not the same as comparing them with **is**:

listing1212.py

```
fruitlist = ["apple", "banana", "cherry", "durian"]
newfruitlist = fruitlist
verynewfruitlist = fruitlist[:]

print( fruitlist == newfruitlist )
print( fruitlist == verynewfruitlist )
print( newfruitlist == verynewfruitlist )
```

The == operator actually compares the contents of the lists, so it returns **True** for all comparisons. For data types for which == is not defined, it executes an identity comparison, but for lists it has been defined as a comparison of the contents. I will return to this topic when discussing "operator overloading" in Chapter 21. .

12.5.2 Shallow vs. deep copies

If (some of) the items of your list are lists themselves (or other mutable data structures which are introduced in the next chapters), you may get problems if you copy the list using the `<newlist> = <oldlist>[:]` syntax. The reason is that such a copy is a "shallow copy," which means that it copies each of the elements of the list with a regular assignment, which entails that the items in the list that are lists themselves become aliases of the items on the original list.

listing1213.py

```
numlist = [ 1, 2, [3, 4] ]
copylist = numlist[:]

numlist[0] = 5
numlist[2][0] = 6
print( numlist )
print( copylist )
```

In the code above, you can see that the assignment `numlist[0] = 5` only has an effect on `numlist`, as `copylist` contains a copy of `numlist`. However, since this is a shallow copy,

the assignment to `numlist[2][0]` has an effect on both lists, as the sublist `[3, 4]` is stored in `copylist` as an alias.

If you want to create a "deep copy" of a list (i.e., a copy that also contains true copies of all mutable substructures of the list, which in turn contain true copies of all their mutable substructures, etcetera), then you can use the `copy` module for that. The `deepcopy()` function from the `copy` module allows you to create deep copies of any mutable data structure.

listing1214.py

```
from copy import deepcopy

numlist = [ 1, 2, [3, 4] ]
copylist = deepcopy( numlist )

numlist[0] = 5
numlist[2][0] = 6
print( numlist )
print( copylist )
```

Note that the `copy` module also contains a function `copy()` that makes shallow copies. If you wonder why that function is included as you can easily create shallow copies of lists with the `<newlist> = <oldlist>[:]` command: the `copy` module not only works for lists, but for any mutable data structure. Not for all such data structures there exist shortcuts to create shallow copies.

12.5.3 Passing lists as arguments

When you pass a list as an argument to a function, this is a "pass by reference." The parameter that the function has access to will be an alias for the list that you pass. This means that a function that you pass a list to, can actually change the contents of the list.

This is important, so I repeat it: when you pass a mutable data structure to a function, this is a "pass by reference," meaning that the data structure is passed as an alias and the function can change the contents of the data structure.

You have to know whether a function that you pass a list to will or will not change the list. If you do not want the function to change the list, and you do not know if it will, you best pass a deep copy of the list to the function.

listing1215.py

```
def changelist( x ):
    if len( x ) > 0:
        x[0] = "CHANGE!"

fruitlist = ["apple", "banana", "cherry", "durian"]
changelist( fruitlist )
print( fruitlist )
```

The reason that a list is "passed by reference" and not "by value" is that technically, every argument that is passed to a function must be stored in the computer in a specific block of

memory that is part of the processor. This is called the "stack," and it is pretty limited in size. Since lists can be really long, allowing a program to place a list on the stack would cause all kinds of annoying runtime errors. In Python, as in most other programming languages, for the most part only basic data types (such as integers, floats, and strings) are passed by value.

12.6 Nested lists

The elements of a list may be lists themselves (which also may contains lists, etcetera). This is a good way to create a matrix in a program. For instance, you can create a Tic-Tac-Toe board, where a dash (-) represents an empty cell, as follows:

```
board = [ ["-", "-", "-"], ["-", "-", "-"], ["-", "-", "-"] ]
```

The first row of the board is represented by `board[0]`, the second row by `board[1]`, and the third row by `board[2]`. If you want to access the first cell of the first row, that is `board[0][0]`, the second cell is `board[0][1]` and the third cell is `board[0][2]`. For example, the following code places an "X" in the middle of the board, and an "O" in the upper right corner. It also displays the board in a nice way (with markers for rows and columns around it).

listing1216.py

```
def display_board( b ):
    print( "  1 2 3" )
    for row in range( 3 ):
        print( row+1, end=" ")
        for col in range( 3 ):
            print( b[row][col], end=" " )
        print()

board = [ ["-", "-", "-"], ["-", "-", "-"], ["-", "-", "-"] ]
board[1][1] = "X"
board[0][2] = "O"
display_board( board )
```

12.7 List casting

You can type cast a sequence of elements to a list using the **`list()`** function. The code below turns a tuple into a list.

```
t1 = ( "apple", "banana", "cherry" )
print( t1 )
print( type( t1 ) )
fruitlist = list( t1 )
print( fruitlist )
print( type( fruitlist ) )
```

This is sometimes necessary, in particular when you have an "iterator" available and you want to use the elements in a list format. An iterator is a function that generates a sequence (more on iterators is given in Chapter 23). An example of an iterator that I already discussed is the **range()** function. The **range()** function generates a sequence of numbers. If you want to use these numbers as a list, you can use list casting.

```
numlist = range( 1, 11 )
print( numlist )
numlist = list( range( 1, 11 ) )
print( numlist )
```

You can turn a string into a list of its characters by using a list casting on the string.

12.8 List comprehensions

List comprehensions are a concise way to create lists. They are typical for Python, but you do not find them in many other programming languages. They are not actually needed, as you can use functions to achieve the same effect, but as they are often used in examples (especially by people who want to show off their Python abilities to create short statements that have extensive effects), I thought it prudent to discuss them. If you are never going to use them in your own code, that is fine as they are completely optional. But you should be able to recognize them in other people's code.

Suppose that you want to create a list consisting of the squares of the numbers 1 to 25. A function that creates such a list is:

```
def squareslist():
    squares = []
    for i in range( 1, 26 ):
        squares.append( i*i )
    return squares

sl = squareslist()
print( sl )
```

In Python, you can create that list with one single statement, namely as follows:

```
sl = [ x*x for x in range( 1, 26 ) ]
print( sl )
```

Now suppose that you want to create this list, but want to leave out (for some reason) the squares of any numbers that end in 5. That would add at least two lines to the function above, but with list comprehensions you can still do it with that single line:

```
sl = [ x*x for x in range( 1, 26 ) if x%10 != 5]
print( sl )
```

A list comprehension consists of an expression in square brackets, followed by a **for** clause, followed by zero or more **for** and/or **if** clauses. The result is a list that contains the

elements that result from evaluating the expression for the combination of the **for** and **if** clauses.

The results can become quite complex. For instance, here is a list comprehension that creates a list of tuples with three integers between 1 and 4, whereby the three integers are all different:

listing1217.py

```
triplelist = [ (x,y,z) for x in range( 1, 5 )
    for y in range( 1, 5 ) for z in range( 1, 5 )
    if x != y if x != z if y != z]
print( triplelist )
```

If you find list comprehensions hard to use, remember that there is absolutely no reason to use them except for keeping code concise, and that keeping code readable and understandable is far more important than keeping it concise.

What you learned

In this chapter, you learned about:

- Lists
- Mutability of lists
- Using + and * with lists
- List methods `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `index()`, `count()`, `sort()`, and `reverse()`
- **del** with lists
- Aliasing
- The keyword **is**
- Creating list copies
- Creating deep copies of lists using `deepcopy()`
- Using lists as arguments
- Nested lists
- List casting
- List comprehensions

Exercises

Exercise 12.1 A magic 8-ball, when asked a question, provides a random answer from a list. The code below contains a list of possible answers. Create a magic 8-ball program that asks a question, then gives a random answer.

exercise1201.py

```
answers = [ "It is certain", "It is decidedly so", "Without a \
doubt", "Yes, definitely", "You may rely on it", "As I see it, \
yes", "Most likely", "Outlook good", "Yes", "Signs point to yes",
"Reply hazy try again", "Ask again later", "Better not tell you \
now", "Cannot predict now", "Concentrate and ask again", "Don't \
count on it", "My reply is no", "My sources say no", "Outlook \
not so good", "Very doubtful" ]
```

Exercise 12.2 A playing card consists of a suit (`"Hearts"`, `"Spades"`, `"Clubs"`, `"Diamonds"`) and a value (`2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `"Jack"`, `"Queen"`, `"King"`, `"Ace"`). Create a list of all possible playing cards, which is a deck. Then create a function that shuffles the deck, producing a random order.

Exercise 12.3 A first-in-first-out (FIFO) structure, also called a "queue," is a list that gets new elements added at the end, while elements from the front are removed and processed. Write a program that processes a queue. In a loop, ask the user for input. If the user just presses the `Enter` key, the program ends. If the user enters anything else, except for a single question mark (?), the program considers what the user entered a new element and appends it to the queue. If the user enters a single question mark, the program pops the first element from the queue and displays it. You have to take into account that the user might type a question mark even if the queue is empty.

Exercise 12.4 Count how often each letter occurs in a string (case-insensitively). You can ignore every character that is not a letter. Print the letters with their counts, in order from highest count to lowest count.

Exercise 12.5 The sieve of Eratosthenes is a method to find all prime numbers between 1 and a given number using a list. This works as follows: Fill the list with the sequence of numbers from 1 to the highest number. Set the value of 1 to zero, as 1 is not prime. Now loop over the list. Find the next number on the list that is not zero, which, at the start, is the number 2. Now set all multiples of this number to zero. Then find the next number on the list that is not zero, which is 3. Set all multiples of this number to zero. Then the next number, which is 5 (because 4 has already been set to zero), and do the same thing again. Process all the numbers of the list in this way. When you have finished, the only numbers left on the list are primes. Use this method to determine all the primes between 1 and 100.

Exercise 12.6 Write a Tic-Tac-Toe program that allows two people to play the game against each other. In turn, ask each player which row and column they want to play. Make sure that the program checks if that row/column combination is empty. When a player has won, end the game. When the whole board is full and there is no winner, announce a draw.

This is a fairly long program to write (60 lines or so). It will definitely help to use some functions. I recommend that you create a function `display_board()` that gets the board as parameter and displays it, a function `getRowCol()` that asks for a row or a column (depending on a parameter) and checks whether the user entered a legal value, and a function

`winner()` that gets the board as argument and checks if there is a winner. Keep track of who the current player is using a global variable `player` that you can pass to a function as an argument if the function needs it. I also use a function `opponent()`, that takes the player as argument and returns the opponent. I use that to switch players after each move.

The main program will be something along the lines of (in pseudo-code):

```
display board
while True:
    ask for row
    ask for column
    if row/column combination already occupied:
        display error message
        continue
    place player marker on row/column combination
    display board
    if there is a winner:
        announce winner
        break
    if the board is full:
        announce draw
        break
    switch players
```

Exercise 12.7 Create a program that is a simplified version of the game "Battleship." The computer creates (in memory) a grid that is 4 cells wide and 3 cells high. The rows of the grid are numbered 1 to 3, and the columns of the grid are labeled A to D. The computer hides a battleship in three random cells in the grid. Each battleship occupies exactly one cell. Battleships are not allowed to touch each other horizontally or vertically. Make sure that the program places the battleships randomly, so not pre-configured.

The computer asks the player to "shoot" at cells of the grid. The player does so by entering the column letter and row number of the cell which he wants to shoot at (e.g., `"D3"`). If the cell which the player shoots at contains nothing, the computer responds with "Miss!" If the cell contains a battleship, the computer responds with "You sunk my battleship!" and removes the battleship from the cell (i.e., a second shot at the same cell is a miss). As soon as the player hits the last battleship, the computer responds with displaying how many shots the player needed to shoot down all three battleships, and the program ends.

To help with debugging the game, at the start the computer should display the grid with periods marking empty cells and `X`s marking cells with battleships.

Hint: If you have troubles with this exercise, start by using a board which has the battleships already placed. Once the rest of the code works, add a function that places the battleships at random, at first without checking if they are touching one another. Once that works, add code that disallows battleships touching each other.

Exercise 12.8 The "subset sum" problem asks the question whether a list of integers contains a subset of integers that, when summed, gives zero as answer. For instance, for the list `[1, 4, -3, -5, 7]` the answer is "yes," as $1 + 4 - 5 = 0$. However, for the list `[1, 4, -3, 7]` the answer is "no," as there is no subset of integers that adds up to zero.

Write a program that solves the "subset sum" problem for a list of integers. If there is a solution, print it; if not, report that there is no solution.

Hint: This problem is tackled best using recursion. If you skipped Chapter 9, you better skip this exercise too.

Chapter 13

Dictionaries

Strings, tuples and lists are ordered data structures, which entails that they can be indexed. Not all data is naturally ordered, which is why Python offers dictionaries as a way to structure unordered data.

13.1 Basics of dictionaries

Dictionaries are unordered collections of elements. To identify an element, you have to know the element's "key."

Basically, dictionaries store "key-value pairs." Any immutable data type can function as a key. A very common type to use as key is the string.

You create dictionaries using curly brackets `{}`, similar to how you create lists using square brackets. An empty dictionary you create by assigning `{}` to a variable. You can create a dictionary with contents by describing every element of the dictionary between the curly brackets using the syntax `<key>:<value>`, and commas between the elements.

Here a dictionary fruitbasket is created, that contains three key-value pairs, namely the key `"apple"` with value 3, the key `"banana"` with value 5, and the key `"cherry"` with value 50.

```
fruitbasket = { "apple":3, "banana":5, "cherry":50 }
```

To access the value belonging to a specific key, you use the same syntax as you would use for a list, except that you write the key in the place where you would write the index for a list.

```
fruitbasket = { "apple":3, "banana":5, "cherry":50 }
print( fruitbasket["banana"] )
```

You can use a **`for`** loop to traverse a dictionary. The variable in the **`for`** loop gets access to all the keys.

listing1301.py

```
fruitbasket = { "apple":3, "banana":5, "cherry":50 }
for key in fruitbasket:
    print( "{}:{}".format( key, fruitbasket[key] ))
```

Trying to access a dictionary element using a key that is not available in the dictionary will lead to a runtime error. However, adding a new element to a dictionary you can do by simply assigning a value to the dictionary item identified by the new key. For instance, adding a `"mango"` to the `fruitbasket` you can do as follows:

```
fruitbasket = { "apple":3, "banana":5, "cherry":50 }
print( fruitbasket )
fruitbasket["mango"] = 1
print( fruitbasket )
```

Overwriting a dictionary item works in exactly the same way as creating a new dictionary item: you just assign a value to it.

Deleting an item from a dictionary you do using the **`del`** keyword, just as you can use with lists.

```
fruitbasket = { "apple":3, "banana":5, "cherry":50 }
print( fruitbasket )
del fruitbasket["banana"]
print( fruitbasket )
```

You can determine the number of key-value pairs in a dictionary by using the **`len`**`()` function.

By the way, do you understand how the ordering of a dictionary works when looking at the display of a dictionary? Think about it.

The answer is: there is no ordering. That was what I said at the start: dictionaries are unordered. In principle I cannot even tell you what ordering you see on your screen when you run the code above, because it might differ between computers, operating systems, and versions of Python. There is a certain structure to the ordering of the items, but nothing that you can (or should desire to) predict. By adding enough items, the ordering might even suddenly change completely.

Since dictionaries are unordered, many of the concepts that are applicable to lists, do not work on dictionaries. For instance, you cannot refer to “slices” of a dictionary, and neither can you “sort” or “reverse” a dictionary. So dictionaries are quite limited, but they do have their uses.

13.2 Dictionary methods

This section describes the dictionary methods that are most often used.

13.2.1 `copy()`

Just like lists, if you assign a variable that contains a dictionary to another variable, you are not creating a copy of the dictionary; you are actually creating an alias (if you do not remember what an alias is, see Chapter 12). You cannot use the trick which is used for lists to create a copy, as it uses a slice-syntax, and dictionaries do not support slices. Therefore, there is a method `copy()` that returns a copy of a dictionary.

listing1302.py

```
fruitbasket = { "apple":3, "banana":5, "cherry":50 }
fruitbasketalias = fruitbasket
fruitbasketcopy = fruitbasket.copy()

print( id( fruitbasket ) )
print( id( fruitbasketalias ) )
print( id( fruitbasketcopy ) )
```

Note that this method makes a shallow copy of the dictionary (see Chapter 12 if you do not remember the difference between shallow and deep copies). If you want to make a deep copy, use the `deepcopy()` function from the `copy` module.

13.2.2 `keys()`, `values()`, and `items()`

The method `keys()` provides an iterator that lists all the keys of a dictionary. The method `values()` provides an iterator that lists all the values of a dictionary. The method `items()` provides an iterator that lists all the key-value pairs of a dictionary as tuples.

I specifically say that these methods returns an iterator and not a list. If you want to turn them into lists, you have to use list casting (see Chapter 12).

```
fruitbasket = { "apple":3, "banana":5, "cherry":50 }
print( list( fruitbasket.keys() ) )
print( list( fruitbasket.values() ) )
print( list( fruitbasket.items() ) )
```

At this point you might be wondering when you can use an iterator. You mainly use iterators for **`for`** loops (though you can also use them as arguments for the functions **`max`**`()`, **`min`**`()` and **`sum`**`()`).

```
fruitbasket = { "apple":3, "banana":5, "cherry":50, "durian":0, "
   mango":2 }
for key in fruitbasket.keys():
    print( "{}:{}".format( key, fruitbasket[key] ) )
print( sum( fruitbasket.values() ) )
```

Since this code provides an unpredictable order for the keys, you might want to sort them before looping over them. Since `keys()` does not provide a list, it cannot be sorted directly, but you can turn the result into a list using list casting. After doing that, you can sort.

listing1303.py

```
fruitbasket = { "apple":3, "banana":5, "cherry":50, "durian":0, "
    mango":2 }
keylist = list( fruitbasket.keys() )
keylist.sort()
for key in keylist:
    print( "{}:{}".format( key, fruitbasket[key] ) )
```

`keylist = list( fruitbasket.key() ).sort()` does not work, as you cannot apply the `sort()` method directly to the list casting. You must first create the list, then sort it. Neither can you write `for key in keylist.sort()`, as the `sort()` method has no return value.

If you wonder why Python seems to prefer iterators instead of lists: the answer is that iterators are more general and use much less memory. They are "lazy" methods, as they only provide an item when it is requested.

13.2.3 `get()`

The `get()` method can be used to get a value from a dictionary even when you do not know if the key for which you seek the value exists. You call the `get()` method with the key you are looking for, and it will return the corresponding value when the key exists in the dictionary, or the special value **`None`** when the key does not exist in the dictionary. If you want to return a specific value instead of **`None`** if the key does not exist, you can add that value as a second argument to the method.

listing1304.py

```
fruitbasket = { "apple":3, "banana":5, "cherry":50, "durian":0, "
    mango":2 }

apple = fruitbasket.get( "apple" )
if apple:
    print( "apple is in the basket" )
else:
    print( "no apples in the basket")

orange = fruitbasket.get( "orange" )
if orange:
    print( "orange is in the basket" )
else:
    print( "no oranges in the basket")

banana = fruitbasket.get( "banana", 0 )
print( "number of bananas in the basket:", banana )

strawberry = fruitbasket.get( "strawberry", 0 )
print( "number of strawberries in the basket:", strawberry )
```

Run and study the example above closely, as what it demonstrates about the `get()` method is very useful. Suppose that you store a collection of items with corresponding quantities,

for instance, the contents of a fruit basket with the keys being the names of the fruits and the values being the quantities. When you query the `fruitbasket` using the `get()` method with a second parameter zero, you can look for any fruit in the basket without the need to check first if the fruit exists in the basket, because if you ask for a fruit that is not there, the `get()` method returns zero, which is exactly what you want to hear.

13.2.4 Practice

Exercise The code below contains a list of words. Build a dictionary that contains all these words as keys, and their quantities as values. Print the words with their quantities.

listing1305.py

```
wordlist = ["apple","durian","banana","durian","apple","cherry",
    "cherry","mango","apple","apple","cherry","durian","banana",
    "apple","apple","apple","apple","banana","apple"]
```

Exercise The code block below contains a string that is a list of words, separated by commas. Build a dictionary that contains all these words as keys, and how often they occur as values. Then print the words with their quantities.

listing1306.py

```
text = "apple,durian,banana,durian,apple,cherry,cherry,mango," + \
    "apple,apple,cherry,durian,banana,apple,apple,apple," + \
    "apple,banana,apple"
```

Exercise The code block below contains a very small dictionary that contains the translations of English words to Dutch. Write a program that uses this dictionary to create a word-for-word translation of the given sentence. A word for which you cannot find a translation, you can leave "as is." The dictionary is supposed to be used case-insensitively, but your translation may consist of all lower case words. It is nice if you leave punctuation in the translation, but if you take it out, that is acceptable (as leaving punctuation in is quite a bit of work, and does not really have anything to do with dictionaries – besides, leaving punctuation in is much easier to do once you have learned about regular expressions).

listing1307.py

```
english_dutch = { "last":"laatst", "week":"week", "the":"de",
    "royal":"koninklijk", "festival":"feest", "hall":"hal", "saw":
    "zaag", "first":"eerst", "performance":"optreden", "of":"van",
    "a":"een", "new":"nieuw", "symphony":"symphonie", "by":"bij",
    "one":"een", "world":"wereld", "leading":"leidend", "modern":
    "modern", "composer":"componist", "composers":"componisten",
    "two":"twee", "shed":"schuur", "sheds":"schuren" }

sentence = "Last week The Royal Festival Hall saw the first \
performance of a new symphony by one of the world's leading \
modern composers, Arthur \"Two-Sheds\" Jackson."
```

13.3 Keys

As I said, any immutable data type can be a dictionary key. This means that strings, integers, and floats can all be used as keys. You may remember that tuples are also immutable, which entails that you can use tuples as keys. This can occasionally be useful.

A very straightforward example of tuples being useful as keys is a dictionary in which you want to store information associated with points in two-dimensional space (a discussion of which was given in Chapter 11). There is no good way in which you can store the identification of a point in a single number or string. It is not impossible (for instance, you could store the number-pair as their string-representations, concatenated with a comma in between) but it becomes ambiguous and convoluted (for instance, the string-keys `"2,3"`, `"2, 3"`, `"+2,+3"`, and `"02,03"` would all be representing the same tuple but different keys).

13.4 Storing complicated values

Until now I only considered the case in which a dictionary stores a single value of a simple data type. However, it is possible to store much more complex values in dictionaries. Values can be arbitrary Python objects. For example, you can store a list with each key. Below a dictionary is used to store the students who are following a course. The course is identified by its course number, while the students are identified by their student numbers.

listing1308.py

```
courses = {
    '880254':['u123456', 'u383213', 'u234178'],
    '822177':['u123456', 'u223416', 'u234178'],
    '822164':['u123456', 'u223416', 'u383213', 'u234178']}

for c in courses:
    print( c )
    for s in courses[c]:
        print( s, end=" " )
    print()
```

Suppose that you do not only want to store the student numbers for a course number, but also the name of the course, the ECTS value of the course, and for each student number also the grade. You can do that (for example) by storing the value for a course number as a dictionary, with three keys, namely `"name"`, `"ects"`, and `"students"`. The value for `"name"` is the course name as a string, the value for `"ects"` is the ECTS as an integer, and the value for `"students"` is another dictionary, which contains student numbers as keys and grades as values.

listing1309.py

```
courses = {
    '880254': { "name":"RS: Data Processing", "ects":3,
        "students":{'u123456':8, 'u383213':7.5, 'u234178':6} },
    '822177': { "name":"Understanding Intelligence", "ects":6,
        "students":{'u123456':5, 'u223416':7, 'u234178':9} },
```

```
    '822164': { "name":"Computer Games", "ects":6,
        "students":{'u383213':6, 'u234178':4} } }

for c in courses:
    print( "{}: {} ({})".format( c, courses[c]["name"],
        courses[c]["ects"] ) )
    for s in courses[c]["students"]:
        print( "{}: {}".format( s, courses[c]["students"][s] ) )
    print()
```

Data structures can become a lot more complex than this if you want. However, if you are really considering designing Python programs for data structures like this, you should at least investigate object orientation first (Chapter 20 and onward) and probably do a separate course on databases.

13.5 Lookup speed

Lists and dictionaries are the two most-used data structures in Python. While often it is clear when you should use which data structure, it is helpful if you know a little bit about how Python processes these data structures in case you have a choice.

Suppose that you read a large bunch of numbers from a file. The numbers are all different and can be anything. You later need to compare the numbers on another list to the numbers that you read from the file.

Should you use a list or a dictionary to store the numbers that you read from the file? Since they are just numbers, without extra data, a list seems to be the best option. There is, however, a problem if you use a list here. Check out the following code, in which a list of 10000 numbers is created, and after that some code checks for 10000 different numbers whether they are on the list (which none of them are).

listing1310.py

```
from datetime import datetime

numlist = []
for i in range( 10000 ):
    numlist.append( i )

start = datetime.now()
count = 0
for i in range( 10000, 20000 ):
    if i in numlist:
        count += 1
end = datetime.now()

print( "{}.{} seconds needed to find {} numbers".format(
    (end - start).seconds, (end - start).microseconds,count ) )
```

Here is the code for doing the same thing with a dictionary, where I simply store the value 1 with each number.

listing1311.py

```
from datetime import datetime

numdict = {}
for i in range( 10000 ):
    numdict[i] = 1

start = datetime.now()
count = 0
for i in range( 10000, 20000 ):
    if i in numdict:
        count += 1
end = datetime.now()

print( "{}.{} seconds needed to find {} numbers".format(
    (end - start).seconds, (end - start).microseconds,count ) )
```

You will notice that for a dictionary, the code gives an answer almost immediately, while for a list it takes quite some time for the code to provide an answer.

The reason is that I use the **in** operator to check whether a number is in the list, or in the dictionary. For a list this means that Python searches through the list, sequentially, until it reaches the number or reaches the end of the list. In this case, it means that Python checks 10000 times 10000 numbers (as it cannot find any of them), which is 100 million numbers.

For a dictionary, the process of finding a key is much faster. Python can quickly decide whether or not a key is in a dictionary.[9] Usually, the checking of just a handful of numbers suffices. Therefore, the code is much, much faster for a dictionary.

You might think that a couple of seconds for the list search is still negligible, but the search time increases quadratically with the size of the data. Depending on the problem, using a dictionary might be highly preferable over using a list.

On the other hand, lists take less memory than dictionaries, and if you can directly access a list item via its index, lists are faster than dictionaries. For instance, in the problem above, if the list is sorted you can find numbers on it in a smarter way than using the **in** operator (checking about 14 indices would suffice) – in that case, a list may be faster again.

From this, you should remember that a list is fast if you can access its elements directly via their index, while a dictionary is a much better choice if the main way to find something is by scanning items. The **in** operator seems easy and reads well, but if you use it to seek something in a long list, you better think again.

What you learned

In this chapter, you learned about:

[9] Technically, Python stores the keys for the dictionary in a so-called "hash table." I will not explain the details here, and just tell you that a hash table allows for very fast look-up of keys at the cost of some memory.

- Dictionaries
- Dictionary keys and values
- Dictionary methods `copy()`, `keys()`, `values()`, `items()`, and `get()`
- Complicated dictionaries
- Speed differences between lists and dictionaries

Excercises

Exercise 13.1 Write a program that takes a text (for instance the one given below), splits it into words (where everything that is not a letter is considered a word boundary), and case-insensitively builds a dictionary that stores for every word how often it occurs in the text. Then print all the words with their quantities in alphabetical order.

exercise1301.py

```
text = """How much wood would a woodchuck chuck
If a woodchuck could chuck wood?
He would chuck, he would, as much as he could,
And chuck as much as a woodchuck would
If a woodchuck could chuck wood."""
```

Exercise 13.2 The code block below shows a list of movies. For each movie it also shows a list of ratings. Convert this code in such a way that it stores all this data in one dictionary, then use the dictionary to print the average rating for each movie, rounded to one decimal.

exercise1302.py

```
movies = ["Monty Python and the Holy Grail",
          "Monty Python's Life of Brian",
          "Monty Python's Meaning of Life",
          "And Now For Something Completely Different"]

grail_ratings = [ 9, 10, 9.5, 8.5, 3, 7.5,8 ]
brian_ratings = [ 10, 10, 0, 9, 1, 8, 7.5, 8, 6, 9 ]
life_ratings = [ 7, 6, 5 ]
different_ratings = [ 6, 5, 6, 6 ]
```

Exercise 13.3 A library contains books. Books have a writer, identified by last name and first name. Books also have a title. Books also have a location number that identifies where they can be found in the library. Librarians want to be able to locate a specific book if they know writer and title, and they want to be able to list all the books that they have of a specific writer. What data structure would you use to store the books?

Chapter 14

Sets

Sets are unordered data structures, which contain only unique elements. Few programming languages support sets natively, but Python is one of them. Sets are not used often, but occasionally they provide a nice solution to a problem that you are trying to solve, when you need to ensure that you only have unique solution items.

14.1 Basics of sets

Sets are unordered collections of elements. You cannot access specific elements using an index or a key. The only way to access items in a set is by using a **`for`** loop, or by testing for the existence of elements in the set using the **`in`** operator.

You have to think of sets in the mathematical sense. In mathematics, a set is a collection of elements, all unique, and elements can be part of a specific set, or not part of the set. You use special set operators to combine sets in different ways.

Python uses dictionaries to implement sets; specifically, it implements the elements of a set as dictionary keys. Thus, only immutable data types can be set elements. Sets themselves, however, are mutable.

Since Python uses dictionaries to implement sets, you might think that you can create an empty set by assigning `{}` to a variable. That, however, does not work as it creates an empty dictionary, not an empty set. Instead, you create an empty set by assigning a call to the function **`set()`** to a variable.

To create a set with some elements already in it, you can assign the elements to the variable between curly brackets. Alternatively, you can call the **`set()`** function with a list of the elements as argument.

```
fruitset = { "apple", "banana", "cherry" }
print( fruitset )
```

If you want to create a set consisting of the different characters in a string, you can call **`set()`** with the string as argument.

```
helloset = set( "hello world" )
print( helloset )
```

You can use a **for** loop to traverse a set. The variable in the **for** loop gets access to all the set elements. There is no way to determine in which order you get to see the elements. Sorting them is not possible as long as they form a set. You can, however, use list casting on a set to create a list of its elements, which can then be sorted.

listing1401.py

```
fruitset = { "apple", "banana", "cherry", "durian", "mango" }
for element in fruitset:
    print( element )
print()

fruitlist = list( fruitset )
fruitlist.sort()
for element in fruitlist:
    print( element )
```

You can determine the number of elements in a set using the **len**() function.

14.2 Set methods

To manipulate the contents of sets, the following methods are supported. This is not a complete list of set methods, but these are the most common ones.

14.2.1 `add()` and `update()`

Adding new items to a set you can do using the `add()` method, to add one new element that you provide as an argument. If you want to add multiple new elements at once, you can use the `update()` method, which you provide with a list of the new elements as argument. You can also use `update()` with a tuple as argument, and you can even use it with a string as argument. If you use it with a string, it will consider each letter of the string as a separate element to add.

Since sets can only contain unique elements, any duplicate element that you try to add will be ignored.

listing1402.py

```
fruitset = { "apple", "banana", "cherry", "durian", "mango" }
print( fruitset )
fruitset.add( "apple" )
fruitset.add( "elderberry" )
print( fruitset )
fruitset.update( ["apple","apple","apple","strawberry",
    "strawberry","apple","mango"] )
print( fruitset )
```

14.2.2 `remove()`, `discard()`, and `clear()`

To remove elements from a set, you can use the `remove()` or `discard()` method. Both get the element to remove as argument. The difference between the two methods is that `remove()` will cause a runtime error if the element is not part of the set, while `discard()` will ignore such errors.

```
fruitset = { "apple", "banana", "cherry", "durian", "mango" }
print( fruitset )

fruitset.remove( "apple" )
print( fruitset )
```

`clear()` removes all elements of the set at once.

14.2.3 `pop()`

Calling the `pop()` method will remove an element from the set and return it. You cannot predict which element will be removed, as sets are unordered.

```
fruitset = { "apple", "banana", "cherry", "durian", "mango" }
while len( fruitset ) > 0:
    print( fruitset.pop() )
```

14.2.4 `copy()`

Just like lists and dictionaries, if you assign a variable that contains a set to another variable, you are creating an alias. Like with dictionaries (and probably because sets are implemented as dictionaries), you use the method `copy()` to create a copy of a set.

14.2.5 `union()`

The union of two sets is a set which contains elements of both of them. You can use the `union()` method for one set, with as argument a second set, to return the union of both sets involved. This does not change the sets themselves. Alternatively, you can use the special operator | (pipeline) to create the union of two sets. Note: you might suspect that you can also use the + operator to combine two sets, but + is not defined for sets, and neither is ∗.

listing1403.py

```
fruit1 = { "apple", "banana", "cherry" }
fruit2 = { "banana", "cherry", "durian" }
fruitunion = fruit1.union( fruit2 )
print( fruitunion )

fruitunion = fruit1 | fruit2
print( fruitunion )
```

14.2.6 intersection()

The intersection of two sets is a set which contains only the elements that they both have. You can use the intersection() method for one set, with as argument a second set, to return the intersection of the sets involved. This does not change the sets themselves. Alternatively, you can use the special operator & (ampersand) to create the intersection of two sets.

listing1404.py

```
fruit1 = { "apple", "banana", "cherry" }
fruit2 = { "banana", "cherry", "durian" }
fruitintersection = fruit1.intersection( fruit2 )
print( fruitintersection )

fruitintersection = fruit1 & fruit2
print( fruitintersection )
```

14.2.7 difference()

The difference of two sets is a set which contains only the elements that the first set has that are not also in the second set. You can use the difference() method for one set, with as argument a second set, to return the difference whereby the elements of the argument set are removed from the first set. This does not change the sets themselves. Alternatively, you can use the special operator - (minus) to create the difference of two sets.

listing1405.py

```
fruit1 = { "apple", "banana", "cherry" }
fruit2 = { "banana", "cherry", "durian" }
fruitdifference = fruit1.difference( fruit2 )
print( fruitdifference )

fruitdifference = fruit1 - fruit2
print( fruitdifference )

fruitdifference = fruit2 - fruit1
print( fruitdifference )
```

14.2.8 isdisjoint(), issubset(), and issuperset()

The methods isdisjoint(), issubset(), and issuperset() are all called as methods of one set, with a second set as argument. All return **True** or **False**. isdisjoint() returns **True** if the two sets share no elements. issubset() returns **True** if all the elements of the first set are also found in the argument set. issuperset() returns **True** if all the elements of the argument set are also found in the first set. Note that a set is both a subset and a superset of itself.

listing1406.py

```
fruit1 = { "apple", "banana", "cherry" }
fruit2 = { "banana", "cherry" }

print( fruit1.isdisjoint( fruit2 ) )
print( fruit1.issubset( fruit2 ) )
print( fruit2.issubset( fruit1 ) )
print( fruit1.issubset( fruit1 ) )
print( fruit1.issuperset( fruit2 ) )
print( fruit2.issuperset( fruit1 ) )
print( fruit1.issuperset( fruit1 ) )
```

14.2.9 Practice

Exercise There is also a set method `symmetric_difference()` which returns a set that contains all the elements that are in the union of two sets, except those that are found in both sets. For example, if set 1 contains A, B, and C, and set 2 contains B, C, and D, the symmetric difference of sets 1 and 2 contains A and D. Can you implement the `symmetric_difference()` method by using only some of the methods found above?

Exercise In the chapter on iterations you were asked to write code that determines all the letters that two words have in common, whereby each letter should only be reported once. Using sets, you can do this very efficiently. Please write the appropriate code.

14.3 Frozensets

Python supports a variant on the set type, namely the **`frozenset`**. You create a **`frozenset`** by using the **`frozenset`**`()` function. The elements of a **`frozenset`**, once assigned, cannot be changed. You therefore have to create the **`frozenset`** immediately when you call the **`frozenset`**`()` function, because it is impossible to add or remove elements later. I.e., **`frozenset`**s are immutable.

All the regular set methods work for **`frozenset`**s, except for those that try to change the set. Trying to use such a method for a **`frozenset`** will lead to a syntax error.

```
fruit1 = frozenset( ["apple", "banana", "cherry"] )
fruit2 = frozenset( ["banana", "cherry", "durian"] )

print( fruit1.union( fruit2 ) )
```

What you learned

In this chapter, you learned about:

- Sets
- `add()`, `update()`, `remove()`, `discard()`, `clear()`, `pop()`, `copy()`, `union()`, `intersection()`, `difference()`, `isdisjoint()`, `issubset()`, and `issuperset()`
- Frozensets

Exercises

Exercise 14.1 A famous syllogism says: *All men are mortal. Socrates is a man. Therefore Socrates is mortal.* In the code block below you see some sets. The first is the set of all things (I know a few are missing, but for the sake of argument). The second is the set of all men (assuming that the first set indeed contains all things). The third set contains everything that is mortal (again, assuming...). Using set operators and methods, show that indeed (a) all men are mortal, (b) Socrates is a man, and (c) Socrates is mortal. Also shows that (d) there are mortal things that are not men, and (e) there are things that are not mortal.

exercise1401.py

```
allthings = {"Socrates", "Plato", "Eratosthenes", "Zeus", "Hera",
    "Athens", "Acropolis", "Cat", "Dog"}
men = {"Socrates", "Plato", "Eratosthenes"}
mortalthings = {"Socrates","Plato","Eratosthenes","Cat","Dog"}
```

Exercise 14.2 Write a program that first produces three sets of numbers between 1 and 1000, the first all those numbers that are divisible by 3, the second all those numbers that are divisible by 7, and the third all those numbers that are divisible by 11. It is easiest to do that with list comprehension, but it is not necessary. Now produce sets of all the numbers between 1 and 1000 that (a) are divisible by 3, 7, and 11, (b) are divisible by 3 and 7, but not by 11, (c) that are not divisible by 3, 7, or 11. The shortest solution has only one line of code for each of the six sets.

Chapter 15

Operating System

Until now, we considered Python programs as self-contained functionalities. Python programs, however, run on a computer, and occasionally the program must deal with computer intricacies. This will start to play a major role from Chapter 16 onward, when I will discuss file handling. To deal with the computer, Python offers a series of standard functionalities in the `os` module, whereby `os` is a common abbreviation for "operating system." This chapter explains the most important functions from the `os` module.

15.1 Basics of operating systems

A computer consists of hardware, while programs consist of software. The software uses facilities offered by the hardware. While in the early days of computer programming, programmers accessed hardware directly (for instance, to make a pixel visible on a computer screen, a programmer placed a value in a specific memory address that was directly coupled to the screen – an approach called "poking"), nowadays hardware is so complex and diverse that this is no longer a viable approach. Let alone the fact that if you want to write a program that runs on multiple computers, you cannot afford to access hardware directly as hardware differs from computer to computer.

Therefore, programs access hardware functionalities through an "operating system." An operating system can be seen as a layer between programs and hardware, that offers programs high-level functions to get the hardware to work. Typical operating systems in use nowadays on personal computers are Microsoft's "Windows," Apple's "Mac OS," and the open-source OS "Linux" (though there are many more). Each of these exists in multiple variants, often differentiated by numbers or "builds," and sometimes (in the case of Linux) by a company name. Regardless, they all offer functionalities that allow accessing hardware.

The problem is that while all of them offer such functionalities, the functionalities are not named in consistent ways, and have different parameterizations. This means that if you want to write Python programs that access hardware by directly "talking" to the operating system, your program is not portable to other operating systems. This is where the `os` module comes in. The `os` module offers functions that you can use to access the hardware

with, regardless of the operating system. Basically, the `os` module has a different implementation for each operating system, but your program does not need to know that, as the functions are always named the same, and have the same parameters.

That does not mean that you can be completely oblivious of OS intricacies. For instance, when you access a file, on Windows you might need to include a "drive letter," which Mac OS does not support. Another example is that security and file access are much more flexible on Linux than on either Windows or Mac OS, so accessing files on Linux might generate different kinds of warnings and errors than on other operating systems. There are quite a few functions that only have an effect for particular operating systems. Still, the `os` module is a fine compromise between portability and OS-dependent effectiveness.

15.2 Command prompt

When you are working with a mouse-driven user interface (UI), which is standard for Windows and Mac OS, and is used by many Linux users too, you are actually interacting with a visual representation of the system, specifically, of the file system. Programs and documents are represented by "icons," which have a name. They are grouped by "folders," which are actually "directories" of the file system. You can create new folders, delete documents, rename programs, change security settings, etcetera. All these actions you can also execute by directly typing commands, in an environment that is often called the "command prompt" or "command shell."

Most Linux users are familiar with a command shell, but for many Windows and Mac users this is not something that they are aware of. Both Windows and Mac actually have a program that allows you to work in the command shell. On Windows, you find the "command prompt" as one of the "accessories" or "system tools." On Macs, this is called the "Terminal." If you start that program, you get confronted by a window with a black background and a blinking prompt. Here you can type commands that the system will execute for you.

The commands that you can give depend on the system that you are using. This book is not meant to teach you how to use it, but I want to tell you at least that you can run Python programs directly from the command prompt by typing the command:

```
python <programname>.py
```

As long as Python can be found on your system, and the program is actually found in the current working directory (i.e., the place in the computer's file system where you currently are), or you have specified the complete path for the program, then it will run the program. This can actually be quite handy if you have written a program that processes files and you want to process many files in a "batch." Again, this goes a bit too far for this book, but you might get to a point in your career where this is extremely useful.

The commands that you can give are things like "change the current working directory," "make a new directory," "remove an empty directory," "list all the files in the directory," "delete a file," etcetera. Again, it depends on the operating system what exactly the commands are that you need to give to achieve these things.

Exercise On your system, find the command shell and run the program. On Windows, type "dir" to see the files in the current directory. On Macs and Linux, this command is usually "ls." After doing this, you can close the command shell again.

15.3 File system

A computer's file system consists of a tree-like structured organization of directories and files.

There is one "root" directory, which is the main access point for all other directories. The root directory is identified by a slash (/) or backslash (\), depending on the operating system. Under Windows it is a backslash, under Mac OS and Linux it is a forward slash. However, Windows now also supports the forward slash. I recommend using the forward slash in most cases, as in strings the backslash indicates a special symbol, so if you want to use a backslash in a string as a directory separator, you have to use a double backslash. This tends to be confusing, which is why I recommend using the forward slash.

"Under" the root-directory there are multiple other directories, each identified by a name, and usually also multiple files, each identified by a name. Under each directory there may be more directories and files.

Each operating system has certain restrictions on what file and directory names can be used, but in general most characters are supported. It is convention that regular files have an extension, which is placed at the end of the file name, and separated from the filename with a period. The extension identifies what kind of file it is, for instance, an executable program (`.exe`), a flat text file (`.txt`), or a Python file (`.py`). It is also convention that directory names do not have such an extension. However, this is not a rule, and you may certainly encounter files without, and directories with an extension. Note that in the visual environment, extensions for files are often hidden, but they are there – you just do not see them.

To uniquely identify a file, you need to know its exact "path" from the root to the file, following the directories. The path name for the file is `/<directory>/<directory>/.../<filename>`. Under Windows, a drive letter can be placed in front of this path, making it `<drive>:/<directory>/<directory>/.../<filename>`. For instance, if under Windows, on the "C" drive, under the root there is a directory "Python34," under which there is a directory "Lib," in which you can find a file "os.py," the path for that file is `C:/Python34/Lib/os.py`. Under Windows, this path is case insensitive, so you can use only lower case letters if you like. That is not the case for all operating systems, though.

When you are working in the file system (and you always are working in the file system, even if you do not realize that), there is a "current directory," which is identified by a period (`.`). If you want to access a file in the current directory, you do not need to know the complete path; it is enough to know the file name. One directory "higher" than the current directory (i.e., the "parent" directory) is identified by a double period (`..`). The parent directory of the root is the root itself.

Finally, it should be noted that most operating systems support a method that allows you to access files, without knowing the path, even if those files are not in the current directory. Under Windows, for instance, you can set a `PATH` environment variable that contains a string that lists all the directories that Windows will search when you use a filename that is for a file that is not in the current directory. How to adapt such an environment variable is not part of this book, though.

15.4 os functions

The os module supports many functions that allow you to affect the file system. I will mention only a few of them, as many of them are actually a bit dangerous to use (you can easily delete files that you wanted to keep) and you do not need them anyway. If you are really interested in manipulating the file system, you can read up on the dozens of other functions that os supports.

15.4.1 getcwd()

getcwd() returns the current working directory as a string.

```
from os import getcwd
print( getcwd() )
```

15.4.2 chdir()

chdir() changes the current working directory. The new directory is provided as a string argument.

```
from os import getcwd, chdir

home = getcwd()
print( home )
chdir( ".." )
print( getcwd() )
chdir( home )
print( getcwd() )
```

15.4.3 listdir()

listdir() returns a list of all the files and directories in the directory that is given as argument. The names are given in arbitrary order. Notice that they do not include the full path name.

```
from os import listdir

flist = listdir( "." )
for name in flist:
print( name )
```

15.4.4 system()

system() gets a string argument that is a command, that Python executes on the command line. You can use it to do anything that the operating system supports, including running other programs. There are better ways to execute other programs, though (look for functions that start with "exec").

What you learned

In this chapter, you learned about:

- Operating systems
- Command prompt
- File systems
- Functions `getcwd()`, `chdir()`, `listdir()`, and `system()`

Exercises

Exercise 15.1 Write a program that lists all the files and directories in the current directory, displaying them with their full path names.

Chapter 16

Text Files

One of the most important uses of Python for data processing is the reading, changing, and writing of text files. Data is often stored in text files, because text files can be easily transferred between different programs. There are multiple standardized formats for text files, such as "comma-separated values" (CSV) files. Python supports particular text file formats through modules, some of which will be discussed later. This chapter focusses on opening, reading, writing, and closing of any text file, regardless of format.

16.1 Flat text files

When programmers refer to "text files" or "flat text files," they mean files in which all characters are meant to be read as regular characters, like you would type on a keyboard. For instance, Python program files are flat text files, as are HTML files. Word processor documents, however, are not flat text files, and neither are images. If you want to know whether a file is a text file or not, you can try to open it in a text editor (such as the editor for the IDLE environment, which comes with Python). If you see only readable text, the file is likely to be a text file. Otherwise, it is a so-called "binary file" (binary files are discussed in Chapter 18).

Text files consist of lines of text. At the end of a line, there is a "newline" symbol, which in Python is the character `"\n"`. Different operating systems use slightly different ways of storing this character in a text file: some Windows programs store it as "carriage return plus line feed" (`"\r\n"`), while on Linux it is always stored as a single `"\n"`. As long as you access a file from Python as a regular text file, Python will convert the characters that it reads to the standard `"\n"`, and vice versa when it writes. So you do not need to worry about such differences (except when you need to transfer text files between operating systems).

16.1.1 File handles and pointers

When you work with a file in a program, you have to open the file. Opening a file provides a so-called "file handle." A file handle can be seen as an access point to the file. It contains a "pointer" that indicates a particular place in the file. That pointer is used when you read

from or write to the file. For instance, when you read from the file, it starts reading at the pointer, and moves the pointer forward in the file.

When you open a file, the pointer is placed at a particular spot in the file, depending on how you opened the file. If you opened the file for reading only, the pointer is placed at the start of the file. The same is true when you open the file for both reading and writing. If you open the file for "appending" (i.e., to place new data at the file's end), the file pointer is positioned at the end of the file. Lastly, if you open a file for writing only, actually the file is completely emptied and the file pointer is placed at the start of the, now empty, file. To create a new file (i.e., a file with a name that does not exist yet), you open it for "writing only."

After opening the file, the file handle is the only access point for the file. All actions you perform on the file, you perform as methods for the file handle.

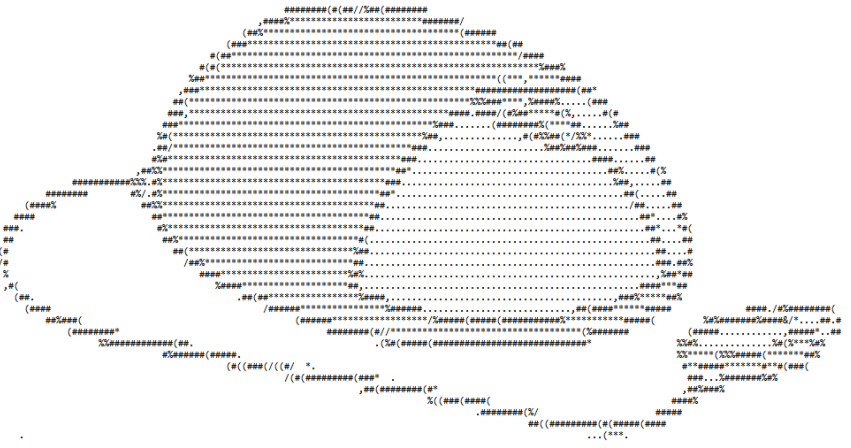
Note that any operating system only allows a limited number of files to be open simultaneously. Therefore you should close files that you no longer need to work with.

16.1.2 Moving the file pointer

The file pointer, that indicates where in a file you are working, is moved automatically. For instance, when you read 10 characters from a file, the file pointer indicates which is the first of those 10 characters, and, while reading, moves up 10 characters, so that its new position is 10 characters further in the file than before. When you deal with text files, the automatic movements of the file pointer are exactly what you want. You can position the file pointer manually using specific methods, but such methods are, in general, only used when dealing with binary files. Therefore, in this chapter I will ignore these methods, but they will be discussed in the chapter on binary files.

16.1.3 Buffering

When you make changes to files, these often are not stored in the files immediately. Instead, the operating system "buffers" the changes in memory, and "flushes" the buffers to the actual files when it sees a need for that. You can force the flushing of the buffers by closing a file. Buffers are also flushed when your program ends normally.



However, when your program crashes (for instance because of a runtime error), buffers might not be flushed, and your files will not be updated to the point where the crash took place. So you cannot take the file contents into account when trying to debug a program.

16.1.4 File processing programs

Most programs that deal with text files follow a process that, in a loop, reads contents of a file, processes those contents in some way, then writes the contents to another file. For instance, a program might read lines from a text file, and for each line sort the words, then write the sorted words to another text file. This is hardly any different from a program that asks the user the provide, in a loop, a line of text, then sorts the words in the line, and displays them using the **`print`**`()` function. While students tend to find it easy to write the version of the program that gets user input and displays output, my experience has taught me that many students find it very hard to use file input and output for the same purpose.

I never found out why exactly students think it is so much harder to do it with files, though I can imagine that you feel you have little control over your own program when working with files. When you provide input to a program manually, and see it displaying outputs, you always know more or less what lines of your code Python is processing, and you can make up tests on the fly. If you work with files, you have to prepare your files beforehand, then run the program and wait until it finishes before you can examine the contents of the output files.

While working with files might give a sense of lack of control, during development of the program you can always include **`print`**`()` statements to get an insight in what the program is doing. For instance, when it reads a line, you can print that line, and when it writes a line, you can also print that line. That way, your insight in the inner workings of the program is no different regardless whether you use manual inputs and screen outputs, or file inputs and outputs.

16.2 Reading text files

To read the contents of a file, you must first open it, then read the contents, then close it.

16.2.1 Opening a file using `open()`

To open a file, you use the **`open`**`()` function.

The **`open`**`()` function gets two arguments, of which the second one is optional. The first argument is the name of the file. If the file is not in the current directory, you have to include the complete path to the file so that Python can find it. The second argument is the "mode." The mode indicates how you want to treat the file. The default mode (which is picked when you do not supply the second argument) is opening the file as a text file for reading only. How you set other modes is discussed later.

The **`open`**`()` function returns a file handle, which you use for all the remaining functionalities.

Rather than writing `<handle> = `**`open`**`( <filename> )`, you will often see Python programs that write this as **`open`**`( <filename> )` **`as`** `<handle>`. These two ways of writing

code are equivalent. I myself prefer the first, as that is the way it is done in most programming languages. However, the second method has an advantage that I discuss below, when talking about closing a file.

16.2.2 Reading a file using `read()`

The simplest way to read the contents of a file is using the `read()` method, without arguments, on the file handle. This returns a string that contains the complete contents of the file. `read()` can get an argument, but I will discuss that in the chapter on binary files.

Reading from a file moves the file pointer to right after the part of the file that was read. This means that if you use the `read()` method without arguments, the file pointer is moved to the end of the file. This entails that if you would try to `read()` from it a second time, nothing would be read, as there is nothing to be read after the spot where the file pointer is. `read()` then returns an empty string.

16.2.3 Closing a file using `close()`

To close a file, you use the `close()` method on the file handle. After that, the handle is no longer associated with the file. Every file that you open, you should close at some point in your program.

So, a complete program that opens a file, reads the complete contents, prints them, and closes the file again, is as follows:

listing1601.py

```
fp = open( "pc_rose.txt" )
print( fp.read() )
fp.close()
```

If everything that you need to do with a file is done in a single block, you can write this block as follows:

```
with open( <filename> ) as <handle>:
    <statements>
```

This syntactic construction has the advantage that the file will be closed automatically after the block `<statements>` ends, so you do not need to include an explicit `close()` call. This construction is typically Python; you do not see it in many other programming languages.

16.2.4 Displaying the contents of a file

Now the first few functions and methods for dealing with text files have been introduced, I can show some code that reads the contents of a file.

listing1602.py

```
with open( "pc_rose.txt" ) as fp:
    buffer = fp.read()
print( buffer )
```

This code assumes that a file is available with the name "pc_rose.txt," and that it is located in the same directory as the program. Appendix E explains how to get it. If the file is unavailable, you get a runtime error. How to deal with such errors will be explained in the next chapter.

Exercise In the code above, change the file name "pc_rose.txt" to something that does not exist. Run the program and observe the error that you get.

Exercise In the code above, change the file name to "pc_woodchuck.txt" (if you have that file). Run the program and observe the output.

16.2.5 Reading lines using `readline()`

To read a text file line by line, you can use the `readline()` method. The `readline()` method reads characters starting at the file pointer up to and including the next newline character, and returns them as a string. You can recognize that you have reached the end of the file by the fact that no characters are read anymore, i.e., the string that is returned is empty.

listing1603.py

```
fp = open( "pc_rose.txt" )
while True:
    buffer = fp.readline()
    if buffer == "":
        break
    print( buffer )
fp.close()
```

Notice that the output of the code above has an empty line between each of the lines displayed. Where is that extra line coming from? Think about it.

The extra line is there because the `readline()` method returns a string of the characters read, up to and including the newline character. So when the buffer is printed, it prints a newline character too. And since the **`print()`** function also moves to a new line after it is executed, there is an empty line printed after each line of text.

Exercise Write a program that reads the lines from "pc_rose.txt," and displays only those lines that contain the word `"name"`.

16.2.6 Reading lines using `readlines()`

A corollary to the `readline()` method is the `readlines()` method. `readlines()` reads all the lines in the file, and returns them as a list of strings. The strings include the newline characters.

listing1604.py

```
fp = open( "pc_rose.txt" )
buffer = fp.readlines()
```

```
for line in buffer:
    print( line, end="" )
fp.close()
```

Note that the output of the code above does not have the empty lines between the lines of text, as the **print()** function includes the "end="" argument, which entails that **print()** itself does not go to the next line after printing.

16.2.7 When to use which file-reading method

Both the `read()` and `readlines()` method read a whole file at once. Obviously, for small files this is acceptable, but for long files you might not have enough memory to store the file contents efficiently. In such circumstances (or when you do not know the file size), you should read a file line by line with the `readline()` method.

It is often a good idea, during code development, to process only the first few lines of a file. That way you limit the amount of time that the program needs to process a file, and limit its output, which makes debugging easier. For instance, the code below process the first 5 lines of a file.

listing1605.py

```
fp = open( "pc_jabberwocky.txt" )
count = 0
while count < 5:
    buffer = fp.readline()
    if buffer == "":
        break
    print( buffer, end="" )
    count += 1
fp.close()
```

Once the program is finished and debugged, I can remove the references to `count` and change the loop to **while True**, to process the whole file.

Exercise Adapt the code above to count how often the word "jabberwock" (with any capitalization) occurs in the first 5 lines. Print only the number of occurrences of that word. Once it works, remove the `count` so that you count the number of occurrences of the word in the text as a whole.

16.3 Writing text files

Writing a text file is similar to reading. You open the file, write to it, and close it.

16.3.1 Opening a file for writing

To open a file for writing, and writing only, you give the value "w" as the second argument to the **open()** function. If the file does not exist yet, it will create it. If it does exist, it will delete its contents.

Let me repeat that: **when you open a file for writing and it already exists, its contents are deleted!** There is no warning message saying "are you sure?" The file is simply emptied. So you have to be very, very careful when opening a file for writing. Usually I ask students to write their programs in such a way that they first check if a file exists before opening it for writing, and give an error message when it already exists. Functions for checking if a file exists are discussed later in this chapter.

16.3.2 Writing using `write()`

To write something to a text file, you use the `write()` method with as argument a string that you want to write to the file. The example code below asks you to enter some strings, and writes them to a file. It stops asking for inputs when you enter an empty line. It then opens the file, reads the contents, and displays them. Run the code, enter at least two lines of text, and see what happens.

listing1606.py

```
fp = open( "pc_writetest.tmp", "w" )
while True:
    text = input( "Please enter a line of text: " )
    if text == "":
        break
    fp.write( text )
fp.close()

fp = open( "pc_writetest.tmp" )
buffer = fp.read()
fp.close()

print( buffer )
```

If you did what I asked, you have noticed that all the text that you entered is in the file, but it all is on one line. There are no newlines in between. The reason is that you have to explicitly write newline characters when you want newlines in your file. When you get input from the keyboard using **`input()`**, while you stop entering input using the `Enter` key, that does not result in a newline character in the string that **`input()`** returns. So you have to add that to the string that you write manually.

Exercise Adapt the code above so that every line of text that you enter, is a separate line in the file that you write.

16.3.3 Writing using `writelines()`

You can write a list of strings at once, by using the `writelines()` method that gets the list as argument. Each of the strings in the list must end in a newline character if you want those newline characters in the output file. `writelines()` is the opposite of `readlines()`; if you use the list that `readlines()` returns as argument for `writelines()`, the contents of the output file will be exactly the same as the contents of the input file.

Note that there is no `writeline()` method. `writeline()` would be exactly the same method as `write()`, so it is not needed.

16.3.4 Practice

Exercise Write a program that reads the contents of "pc_rose.txt," and writes exactly the same contents to the file "pc_writetest.tmp." Then open the file "pc_writetest.tmp" and display the contents. You can easily construct this program by cobbling together some of the code given above.

Exercise Write a program that reads the contents of "pc_rose.txt," reverses each of the lines, and writes the reversed lines to the file "pc_writetest.tmp." Then open the file "pc_writetest.tmp" and display the contents.

16.4 Appending to text files

"Appending" refers to writing at the end of an existing file. When you open a file for appending, the contents are not erased, but the file pointer is placed at the end of the file, where you can then write new data. You open a file in "append" mode by using `"a"` as the mode argument when opening the file.

The code below first displays the contents of "pc_writetest.tmp" (which should exist by now). It then asks the user for lines which are appended to the file. Finally, it displays the contents of the new file. I took the liberty of creating this little program in a slightly-better structured manner than before, using a constant for the filename that is repeated three times in the program, and using a function to display the file contents as this functionality is needed twice.

listing1607.py

```
FILENAME = "pc_writetest.tmp"

def displaycontents( filename ):
    fp = open( filename )
    print( fp.read() )
    fp.close()

displaycontents( FILENAME )

fp = open( FILENAME, "a" )
while True:
    text = input( "Please enter a line of text: " )
    if text == "":
        break
    fp.write( text+"\n" )
fp.close()

displaycontents( FILENAME )
```

16.5 os.path methods

At this point you know everything you need to handle text files in Python. However, there are several handy functions that make your life easier when dealing with files. These are collected in the os.path module. As per usual, I am not going to list all of them, but I will list the ones that you will use the most.

In these functions, the term "path" refers to a filename or a directory name, complete with parent directories (and drive letter). The parent directories (and drive letter) do not need to be there explicitly, but even if they are not, implicitly they still are as each file and each directory is located in a particular place in the file system.

16.5.1 exists()

The function exists() gets a path as argument, and returns **True** if that path exists, and **False** if it does not.

```
from os.path import exists

if exists( "pc_rose.txt" ):
    print( "Rose exists" )
else:
    print( "Rose does not exist" )

if exists( "pc_tulip.txt" ):
    print( "Tulip exists" )
else:
    print( "Tulip does not exist" )
```

16.5.2 isfile()

isfile() tests if the path that is supplied as argument is a file. If it is, it returns **True**. If it is not, it returns **False**. If the path does not exist, the function also returns **False**.

```
from os.path import isfile

if isfile( "pc_rose.txt" ):
    print( "Rose is a file" )
else:
    print( "Rose is not a file" )
```

16.5.3 isdir()

isdir() tests if the path that is supplied as argument is a directory. If it is, it returns **True**. If it is not, it returns **False**. If the path does not exist, the function also returns **False**.

```
from os.path import isdir

if isdir( "pc_rose.txt" ):
    print( "Rose is a directory" )
else:
    print( "Rose is not a directory" )
```

16.5.4 join()

join() takes one or more parts of a path as argument, and concatenates them reasonably intelligently to a legal name for a path, which it returns. This means that it will add and remove slashes as needed. join() is particularly handy in combination with listdir() (see Chapter 15, and the example below).

The reason that join() is handy with listdir(), is that listdir() results in a list of file names that do not include the directory names. Usually, when you ask for a list of file names, you intend to open them at some point. But to open a file that is not in the current directory, you need to know the complete path name that leads to the file. When you apply listdir(), you know where you are looking for files, so you know the elements of the path name. To construct the complete path name for each file, you need to concatenate the elements of the path name to the file name. Rather than trying to decide where you need to add slashes, and which kind of slashes they need to be, you can leave all of that to the join() function.

The code below looks for all the files in the current directory, and lists them including their complete path name. See how join() is used to construct that path name from the current directory, and the file name.

```
from os import listdir, getcwd
from os.path import join

filelist = listdir( "." )
for name in filelist:
    pathname = join( getcwd(), name )
    print( pathname )
```

16.5.5 basename()

basename() extracts the filename from a path, and returns it.

```
from os.path import basename

print( basename( "/System/Home/readme.txt" ) )
```

16.5.6 dirname()

dirname() extracts the directory name from a path, and returns it.

```
from os.path import dirname

print( dirname( "/System/Home/readme.txt" ) )
```

16.5.7 getsize()

getsize() gets the size of the file that is supplied as argument, and returns it as an integer (representing a number of bytes). The file must exist, otherwise you get a runtime error.

```
from os.path import getsize

numbytes = getsize( "pc_rose.txt" )
print( numbytes )
```

Exercise Write a program that adds up the sizes of all the files in the current directory, and prints the result.

16.6 File encoding

Text files use an "encoding," i.e., a system that prescribes how characters in the files are supposed to be interpreted. This encoding may differ between operating systems. You can see the preferred encoding that your system uses with a call to sys.getfilesystemencoding().

```
from sys import getfilesystemencoding

print( getfilesystemencoding() )
```

If you read a text file which uses a different encoding than your file system prefers, you may get a UnicodeDecodeError. Whether or not you get this error for a particular file, is related to your operating system. An annoying consequence of that is that when you port Python code that reads a file to another system, a file that could be read by your code previously may cause your code to crash after the port.[10]

An easy way to get around this problem is by adding an extra parameter when opening a file, which indicates the encoding mechanism that you want to use when reading the file. You do this by adding a parameter encoding=<encodingname>, where <encodingname> is a string that can have a variety of values, for which some typical ones are:

[10] I have to make note of some Python behavior that seems bizarre when you first encounter it: you may get this error when your file contains characters in an encoding that is not supported by your system in lines that you are not even trying to read! E.g., suppose that there is such an erroneous character on line 10 of your file, but you are only trying to read the first 5 lines before closing the file again – your program may still crash! I suspect that this is related to the buffering of data: rather than reading exactly what you ask Python to read, Python reads data in bigger chunks, so that the program is faster when you actually want to go through the whole file. So, by trying to be smart, Python may saddle you up with problems that you did not expect could arise. It is good to be aware of such issues.

- `ascii`: 7-bits encoding, characters with values in the range 00-7F
- `latin-1`: 8-bits encoding, characters with values in the range 00-FF
- `mbcs`: 2-byte encoding, that is currently getting replaced by UTF-8
- `utf-8`: variable bytes encoding

Typically, text files are created with `ascii` or `latin-1` encoding. Since `ascii` is incorporated in `latin-1`, you can safely open any text file by specifying `latin-1` as encoding. It is possible that for the characters beyond the `ascii` range, you get different characters than the person who created the file wanted you to see – that depends on the encoding mechanism that your file system uses. But at least the `UnicodeDecodeError` is avoided. So, when you try to read the contents of a file and get a `UnicodeDecodeError`, you may try to open it using **`open`**`( <file>, encoding="latin-1" )`. Usually that will solve the problem.

Note that while `utf-8` supports a much wider range of characters than `latin-1`, you may still get the `UnicodeDecodeError` when you read a text file that uses `latin-1` encoding on a system that uses `utf-8` encoding, as `utf-8` has no corresponding characters with values in the (hexadecimal) range 80-FF.

If you want to see which special characters are supported with values in the range 80-FF on your system, run the code below. The numerical value of a character in the table can be derived by calculating $16 * row + col$, whereby `row` and `col` are the hexadecimal row and column number, respectively. I do not display the characters in the range 80-9F, as these are normally not filled in.

listing1608.py

```
for i in range(16):
    if i < 10:
        print( ' '+chr( ord( '0' )+i ), end='' )
    else:
        print( ' '+chr( ord( 'A' )+i-10 ), end='' )
print()
for i in range( 10, 16 ):
    print( chr( ord( 'A' )+i-10 ), end='' )
    for j in range( 16 ):
        c = i*16+j
        print( ' '+chr( c ), end='' )
    print()
```

More details on UTF-8 encoding will be given in Chapter 19, but for dealing with text files, the information above suffices.

What you learned

In this chapter, you learned about:

- Text files
- File pointers

- Opening and closing files with **`open`**`()` and `close()`
- Reading files with `read()`, `readline()`, and `readlines()`
- Writing files with `write()` and `writelines()`
- Appending to files
- `os.path` methods `exists()`, `isfile()`, `isdir()`, `join()`, `basename()`, `dirname()`, and `getsize()`
- Dealing with text files with different encoding mechanisms

Exercises

Exercise 16.1 Write a program that reads the contents of the file "pc_woodchuck.txt," splits it into words (where everything that is not a letter is considered a word boundary), and case-insensitively builds a dictionary that stores for every word how often it occurs in the text. Then print all the words with their quantities in alphabetical order.

Exercise 16.2 Do the same thing as you did for the previous exercise, but now process the text line by line. This is something that you would have to do if you had to process a very long text.

Exercise 16.3 Write a program that processes the contents of "pc_woodchuck.txt," line by line. It creates an output file in the current working directory called "pc_woodchuck.tmp," which has the same contents as "pc_woodchuck.txt," except that all the vowels are removed (case-insensitively). At the end, display how many characters you read, and how many characters you wrote.

Exercise 16.4 Write a program that determines how many words of 2 or more letters the files "pc_woodchuck.txt," "pc_jabberwocky.txt" and "pc_rose.txt" have in common. You have to treat the words case-insensitively, and, as always, any character that is not a letter can be treated as a word boundary. If your program is correct, you will find three such words.

Exercise 16.5 For the three files that you used in the previous exercise, count for each of them how often each letter (case-insensitively) occurs. Calculate for each letter and each file the fraction `<number of occurrences of letter in the file>/<total number of letters in the file>`. Write an outputfile (any name, as long as you can safely overwrite it) with extension `.csv`, that contains 26 lines, each line formatted as follows: `"<letter>",<fraction for first file>,<fraction for second file>,<fraction for third file>`. The first line should have letter `a`, the second letter `b`, etcetera. The fractions should be stored with 5 decimals. Finally, display the contents of the outputfile. As the outputfile is a CSV file, you should also be able to load it in a spreadsheet program.

A quick check to see if you did things correctly is that all of the fractions must be between zero and 1, and the fraction for "e" should be highest for both "pc_jabberwocky.txt" and

"pc_rose.txt" (but not so much for "pc_woodchuck.txt"). If you would use longer files which are all in the same language, you would also find that the fractions are usually more or less in each others' neighborhood.

Chapter 17

Exceptions

Sometimes runtime errors occur not because you made a programming mistake, but because your program encountered a situation that you could not foresee when you wrote it. This is particularly relevant when you deal with files: for instance, when you access a file on a USB-stick, and the user pulls out the USB-stick during the file processing, obviously an error occurs that you cannot really account for in your code. Every runtime error raises a so-called "exception," and you can "capture" such exceptions if you want to deal with them in your program, rather than make the program end abruptly.

17.1 Errors and exceptions

When you try to run a Python program, Python first does a quick check to see if all the statements in the program meet the basic Python syntax requirements. If they do not, Python announces a "syntax error" and will not run the program.

If no syntax errors are encountered, Python will run the program, but may encounter statements that generate errors while trying to execute them. Such statements cause a "runtime error." You have seen runtime errors many times while writing code (don't try to deny it).

In general, you try to fix runtime errors by extending or changing your code. For instance, the following program causes a runtime error when you enter a zero as input:

```
from pcinput import getInteger

num = getInteger( "Please enter a number: " )
print( "3 divided by {} is {}".format( num, 3/num ) )
print( "Goodbye!" )
```

Python tells you what kind of error it is, namely a `ZeroDivisionError`. To fix it, you can change the program:

```
from pcinput import getInteger

num = getInteger( "Please enter a number: " )
```

```
if num == 0:
    print( "Dividing by zero is not allowed" )
else:
    print( "3 divided by {} is {}".format( num, 3/num ) )
print( "Goodbye!" )
```

`ZeroDivisionError` is actually the name of an "exception" that Python "raises" (generates). If you do not handle such an exception in your program, Python interrupts the program's execution and splashes its error message on the screen. However, this entails that you actually can handle exceptions in your program and simply continue running it.

While in the code above you should ensure that no exception is raised on dividing by zero – because you can foresee that this might happen – it occasionally happens that you have to accept that exceptions might be raised as you cannot foresee all the circumstances that your program might have to deal with. This is especially relevant when your program depends on elements outside its direct control, such as files and user behavior.

17.2 Exception handling

To handle exceptions explicitly in your program, you use a **`try ... except`** clause. There are different ways of applying this clause.

17.2.1 `try` ... `except`

The most basic form of the **`try ... except`** clause has the following syntax:

```
try:
    <statements>
except:
    <exception handling>
```

When the `<statements>` between **`try:`** and **`except:`** are executed and raise an exception, Python immediately jumps to the `<exception handling>` statements and executes those, after which the program continues as normal, at the first line below the `<exception handling>` statements. If no exceptions are raised during the execution of the `<statements>`, the `<exception handling>` statements are skipped.

Using exception handling, the code at the start of this chapter can be written as follows to avoid runtime errors:

listing1701.py

```
from pcinput import getInteger

num = getInteger( "Please enter a number: " )
try:
    print( "3 divided by {} is {}".format( num, 3/num ) )
except:
    print( "Division by zero is not allowed" )
print( "Goodbye!" )
```

Multiple statements may be part of a single **try ... except** clause. For instance, the following code raises an exception both when a user enters zero and when a user enters 3. Both exceptions are handled by the same **try ... except** clause.

listing1702.py

```
from pcinput import getInteger

num = getInteger( "Please enter a number: " )
try:
    print( "3 divided by {} is {}".format( num, 3/num ) )
    print( "3 divided by {}-3 is {}".format( num, 3/(num-3) ) )
except:
    print( "Division by zero is not allowed" )
print( "Goodbye!" )
```

This is slightly ugly, not only because these errors should have been avoided rather than handled via exceptions, but also because when an exception occurs, it is unclear which of the statements caused it (though in this case, if you enter 3, you can see that the first of the two statements under the **try** executed correctly). However, this is just a demonstration, and there certainly can be situations where you say "I do not care where an exception occurs in this sequence of statements, but if anything happens, I want to do this."

17.2.2 Handling specific exceptions

Examine the code below. This code can cause at least two different exceptions. Which?

```
print( 3 / int( input( "Please enter a number: " ) ) )
```

The two different exceptions that this code can generate are the `ZeroDivisionError` when you enter a zero, and the `ValueError` when you enter something that is not an integer. Try it if you did not try it already.

You can handle both these errors with a single **try ... except** clause, but you can distinguish them by specifying multiple **except**s. Each **except** can be followed by one of the specific exceptions, and the code below it will only be executed if that specific exception is raised.

listing1703.py

```
try:
    print( 3 / int( input( "Please enter a number: " ) ) )
except ZeroDivisionError:
    print( "Dividing by zero is not allowed" )
except ValueError:
    print( "You have not entered an integer" )
print( "Goodbye!" )
```

If you want to capture "all remaining exceptions," you add an **except** without a specific exception at the end. Only one of the **except** clauses will be executed, namely the first one encountered that applies. It is a lot like an **if ... elif ... elif ... else** clause.

listing1704.py

```
try:
    print( 3 / int( input( "Please enter a number: " ) ) )
except ZeroDivisionError:
    print( "Dividing by zero is not allowed" )
except ValueError:
    print( "You have not entered an integer" )
except:
    print( "Something unforeseen went wrong" )
print( "Goodbye!" )
```

Here is a list of some specific exceptions that are raised often:

- `ZeroDivisionError`: Trying to divide by zero
- `IndexError`: Trying to access a list or tuple element with an out-of-bounds index
- `KeyError`: Trying to access a dictionary element with an unknown key
- `IOError`: Any error that occurs while trying to access a file
- `FileNotFoundError`: Trying to open a file that does not exist for reading
- `ValueError`: Error while trying to type cast a value to another value
- `TypeError`: Using a value of a type that is not supported for an operation

Exercise The code below can generate several exceptions. These are now handled by a single **`try ... except`** clause. Extend this code by handling all exceptions that may occur explicitly (there are at least three different kinds of exceptions that can be raised). Note: Let me stress again that I rather have you avoid exceptions occurring than handling them, but in this case I want you to practice with exception handling.

listing1705.py

```
fruitlist = ["apple", "banana", "cherry"]
try:
    num = input( "Please enter a number: " )
    if "." in num:
        num = float( num )
    else:
        num = int( num )
    print( fruitlist[num] )
except:
    print( "Something went wrong" )
```

17.2.3 Adding an `else` clause

At the end of a **`try ... except`** clause you can add an **`else`** clause. The statements with that **`else`** will be executed if no exception occurs. For instance, in the code block below, the calculated value for `num` will only be printed if no exception is raised.

listing1706.py

```
try:
    num = 3 / int( input( "Please enter a number: " ) )
except ZeroDivisionError:
    print( "Dividing by zero is not allowed" )
except ValueError:
    print( "You have not entered an integer" )
except:
    print( "Something unforeseen went wrong" )
else:
    print( num )
print( "Goodbye" )
```

In general, I prefer not to use an **else** clause with an exception, as it feels like the code under the **except** clauses should be code that is only executed in abnormal circumstances. But if you really want you can use it.

17.2.4 Adding a `finally` clause

You can add an extra branch to the **try** clause, which is **finally**:. The **finally** clause has statements which are executed regardless of the manner in which the **try** clause is exited. If it ends normally, it is executed, but if you get a runtime error, it is executed too. You can use such a **finally** clause to, for instance, make sure that a file gets closed before code is harshly interrupted.

listing1707.py

```
try:
    fp = open( "pc_rose.txt" )
    print( "File opened" )
    print( fp.read() )
finally:
    fp.close()
    print( "File closed" )
```

17.2.5 Accessing exception information

You can get extra information on the exception by adding and **as** clause to the **except** statement, using the syntax **except** <exception> **as** <variable>. When an <exception> occurs, the variable is filled with an "exception object," that may provide more information on the exception. The problem is that there is no standardized way to get the information out: it depends on the kind of exception what the variable contains.

The variable will always have a tuple of arguments (even if there is only one), that were provided to the exception when it was raised. You can examine these arguments by means of the attribute <variable>.args. A `ValueError` gets a tuple with only one value, namely a string.

listing1708.py

```
try:
    print( int( "NotAnInteger" ) )
except ValueError as ex:
    print( ex.args )
```

If you run the code below, you see that an `IOError` gets a tuple of two values: an integer and a string. The integer that is provided for the `IOError` is actually quite informative, as it explains what went wrong.

listing1709.py

```
try:
    fp = open( "NotAFile" )
    fp.close()
except IOError as ex:
    print( ex.args )
```

17.2.6 General advice on using exception handling

Never capture an exception and then just ignore it. In particular, you should not use a general **`try ... except`** clause and then do nothing with the exception. If you think you can ignore a certain exception, make sure that you capture that specific exception, and comment in your program why you think you can ignore it. Basically, all exceptions should either be handled responsibly or should just make the program crash.

17.3 File handling exceptions

Any problem with accessing files, whether it is the inability to find a file, a problem with reading or writing a file, or trying to open a protected system file or even a directory, leads to an `IOError` exception. Since problems with file access are quite common and usually at least partly outside the realm of control of the program, it is a good idea to handle `IOError` exceptions in your programs when you can.

Since many different things can go wrong with files, the `args` tuple explained above might be used to provide better information on what you have to do to handle the problem. For instance, if your program asked the user to supply a filename, and when you open that file you get an `IOError`, if the error number (the first element of the tuple) indicates that the file does not exist (2), then an appropriate response might be to simply report this to the user and ask for a new name.

The error numbers are defined in the `errno` module, which you can import in your program. The module offers constants that you can use instead of the actual numbers, which is the convention. The most common error numbers are:

- `errno.ENOENT`: No such file or directory. You get this when you try to access a file that does not exist.

- `errno.EACCESS`: Permission denied. You can get this in varied circumstances, such as when you try to read from a closed file, when you try to open a read-only file for writing, or when you try to open a directory as if it is a file.
- `errno.ENOSPC`: No more space left on device. You get this when you try to write a file and there is no room for it, for instance when you try to write to a USB-stick that is full.

There is a big list of such error numbers which you can easily find in the reference manuals. You might not understand what all of them refer to, and actually many of them are archaic and no longer occur on modern computer systems. The best thing to do when you develop your program, is to try to capture `IOErrors`, and when you do encounter an `IOError`, print the arguments so that you know the number and the error message. You can then look up which `errno` constant that message belongs to, and respond to it in your program if you can do that in a sensible way.

However, just as with other kinds of exceptions, it is better to avoid them than to capture them. There is no reason that you should ever encounter a "file does not exist" error, as you can test whether a file exists with the `exists()` and `isfile()` functions from the `os.path` module.

listing1710.py

```
import errno

try:
    fp = open( "NotAFile" )
    fp.close()
except IOError as ex:
    if ex.args[0] == errno.ENOENT:
        print( "File not found!" )
    else:
        print( ex.args[0], ex.args[1] )
```

Note: Exception `FileNotFoundError` is a "subclass" (see Chapter 22) of `IOError`, which, in Python 3, is actually an alias for yet another exception, namely `OSError`. This means that capturing `FileNotFoundError` is equivalent with capturing `IOError` (or `OSError`) **`as`** `ex` and testing whether `ex.args[0]` holds `errno.ENOENT`.

17.4 Raising exceptions

You are allowed to raise exceptions yourself. For that, you use the keyword **`raise`**, and follow that with one of the known exceptions (potentially, you could create your own, new exceptions if you like, but you need to have studied Chapters 20 and 22 before you are ready for it). You can give the exception arguments of any kind that you like.

You might wonder why you would want to raise your own exceptions. The answer is that when you create a module, when an error occurs (for instance because the main program that uses the module passes arguments to a function that are of an incorrect type), it is bad form to print an error message. It is much better to just raise an exception, and let the main program handle the exception. Here is an example of raising an exception:

listing1711.py

```
def getStringLenMax10( prompt ):
    s = input( prompt )
    if len( s ) > 10:
        raise ValueError( "Length exceeds 10", len( s ) )
    return s

print( getStringLenMax10( "Use 10 characters or less: " ) )
```

When you run this code, you see that if you enter a string of more than 10 characters, a `ValueError` exception is raised. It has two arguments, which you can see displayed as a tuple when Python splashes the exception on the screen. You can handle this exception just as you would handle exceptions that Python itself produces.

The **`raise`** keyword has a second function: if you are in an **`except`** clause, and rather than handle the exception there, you want to pass it on to the "next level" of the program, you can just write the keyword **`raise`**, and the exception will be "re-raised." This can be useful when you want to do a bit of extra handling before the program "crashes" or the exception is handled elsewhere. For instance:

listing1712.py

```
fp = open( "pc_rose.txt ")
try:
    buffer = fp.read()
    print( buffer )
except IOError:
    fp.close()
    raise
fp.close()
```

This code probably runs fine, but if an `IOError` occurs when reading the file, the file gets closed before the exception is re-raised.

What you learned

In this chapter, you learned about:

- Exception handling
- **`except`**, **`else`**, and **`finally`**
- `ZeroDivisionError`, `IndexError`, `KeyError`, `IOError`, `ValueError`, `TypeError`, and `FileNotFoundError`
- Getting exception information
- File handling exceptions
- Raising and re-raising exceptions

Exercises

Exercise 17.1 Which exceptions can the code below raise? Extend the code to handle all of them in a reasonable manner.

exercise1701.py

```
numlist = [ 100, 101, 0, "103", 104 ]

i1 = int( input( "Give an index: " ) )
print( "100 /", numlist[i1], "=", 100 / numlist[i1] )
```

Chapter 18

Binary Files

"Binary files" is the term used to refer to all files that are not text files. Executable programs are binary files, as are image files, movies, word processor documents, and many other file types. It is not common to use Python to process binary files (usually binary files are not handled by general-purpose programming languages, but by special-purpose programs), but it is possible. This chapter will explain how to deal with binary files.

18.1 Opening and closing binary files

The handling of binary files is quite similar to the handling of text files. You have to **`open`**`()` a file when you want to access its contents, `close()` it when you are finished, `read()` from the file and `write()` to the file.

When you open a binary file, you have to indicate to Python that you want to handle this file in "binary mode." You do this by adding a letter `"b"` to the mode argument. For instance, to open a file in "binary read" mode, the mode argument should be `"rb"`. You can also open a file both for reading and writing; reading and writing you indicate with mode `"r+"`, so reading and writing in binary mode is `"r+b"` (while it is also possible to open text files in `"r+"` mode, I did not indicate it in Chapter 16, as it seldom makes sense to open text files in this mode). Just as with text files, if you open a binary file in write-mode, with `"wb"`, the file gets emptied. The mode `"w+b"` will open a file for both reading and writing, but also empties the file to start with.

When you open a file for both reading and writing, if the file pointer is not at the end of the file, when you write you actually overwrite.

You can open any file in binary mode, even text files. However, when you open text files in binary mode, you treat them like binary files, which means that Python does not do the automatic conversion of newline characters.

Closing a binary file is no different from closing a text file.

```
fp = open( "pc_rose.txt", "rb" )
fp.close()
```

Note: The code above has no output – if it does have output, that is a runtime error, meaning that "pc_rose.txt" is not available.

18.2 Reading a binary file

As binary files do not know the concept of "lines," the only way to read from a binary file is to use the `read()` method. If you use `read()` without argument, it reads the whole file (starting at the file pointer). If you give the method an integer as parameter, that integer indicates the number of bytes that are read from the file (starting at the file pointer, and reading at maximum until the end of the file).

A "byte," if you do not know, is an 8-bit character, i.e., a number between zero and 255, which is stored in the smallest possible memory unit that a computer supports. The regular characters on a keyboard are each stored in a single byte, and the characters in a string are also each a byte, though limited to a specific range of numbers.

18.2.1 Byte strings

Here we enter one of the more obscure parts of the Python language. When you read from a binary file, the `read()` method does not return a regular string – it returns a "byte string." There are some subtle differences between regular strings and byte strings. To show you these differences, I first have to tell you that you can indicate that a string is a byte string by placing a letter `b` in front of it. So `"Hello, world"`! is a string, while `b"Hello, world"`! is a byte string.

```
hw1 = "Hello, world!"
hw2 = b"Hello, world!"

print( hw1 )
print( hw2 )
```

The difference between a string and a byte string is that a byte string can contain characters that a string cannot. For instance, if you remember the discussion on the ASCII table, you may recall that each character has a number associated with it. You saw, for instance, that "A" has the number 65, and the space has the number 32. The space was the lowest numbered character that I showed, and you might wonder which characters are associated with numbers 0 to 31. The answer is: these are control codes, and are not legal characters that you can put in a string (barring a few exceptions). You can try to put them in a string using an "escape code": the escape sequence `\x` can be followed by a two-character hexadecimal code that represents the character with the specified number. For example, the hexadecimal code for a space is `\x20`, i.e., `"Hello, world!"` is the same as `"Hello,\x20world!"` (this was discussed in Chapter 10).

```
hw1 = "Hello,\x20world!"
print( hw1 )
```

But what if you try to put illegal characters in a string that way? They are ignored:

```
print( "Hello,\x00\x01\x02world!" )
```

The problem is that such characters can occur in binary files, so you must be able to read them from binary files. Since byte strings can contain such characters, reading from binary files results in byte strings.

```
print( b"Hello,\x00\x01\x02world!" )
```

Characters from a byte string you can access using indices, just like you do with regular strings. The difference here is that with regular strings you get letters, while with byte strings you get numbers. The numbers are the codes for the letters, which you would also get when you use the **ord**() function on the corresponding letter.

listing1801.py

```
hw1 = "Hello, world!"
hw2 = b"Hello, world!"

for c in hw1:
    print( c, end=" " )
print()
for c in hw1:
    print( ord( c ), end=" " )
print()
for c in hw2:
    print( c, end=" " )
```

Since bytes are numbers between 0 and 255, you might want to convert a number to a single-character byte string, or a list of numbers to a multi-character byte string. You can do so using a **bytes** casting on a list of those numbers. Note that if you want to convert a single character, you still have to use a list, but a list with just one element. Do not forget to put the list brackets around that element, because if you do not, the result will not be what you expect.

listing1802.py

```
bs = bytes( [72,101,108,108,111,44,32,119,111,114,108,100,33] )
print( bs )
bch = bytes( [72] )
print( bch )
wrong = bytes( 72 )
print( wrong )
```

Can you convert from a byte string to a regular string? You might think that string casting works, but unfortunately it does not:

```
hw1 = b"Hello, world!"
hw2 = str( hw1 )
print( hw2 )
```

The reason that it does not, is that when a string is in the format of a byte string, it uses an encoding scheme, according to the Unicode standard (discussed in Chapter 16). You have to "decode" the byte string according to a certain decoding scheme, which usually is `"utf-8"`, as that is the most common Unicode format. You decode using the `decode()` method, with the encoding scheme as a string parameter. You can also go from a string to a byte string by encoding using the `encode()` method, again with the encoding scheme as string parameter.

listing1803.py

```
hw1 = b"Hello, world!"
hw2 = hw1.decode( "utf-8" )
print( hw2 )
hw3 = hw2.encode( "utf-8" )
print( hw3 )
```

In general you have little reason to read text files in binary mode, at least not if you just want to access the text, and so you do not have to worry about encoding and decoding byte strings. The exception is when you have to deal with a text file that uses Unicode characters. Such a file cannot be treated as a text file, and you have to open it in binary mode.

18.2.2 Binary reading demonstration

To demonstrate how reading a binary file works, I now open the file "pc_rose.txt," and read ten times ten bytes from it.

listing1804.py

```
fp = open( "pc_rose.txt", "rb" )
for i in range( 10 ):
    buffer = fp.read( 10 )
    print( buffer )
fp.close()
```

When you run the code, you see the ten byte strings being displayed. You may also notice that there are certain control characters visible, such as \r and \n. The \r you would not see if you read this file as a text file, because Python converts it, together with the following \n, to a single \n. Moreover, in a regular string you would not see the \n, because it is a newline character which tells Python to move to the next line.

If instead of a text file, you open an actual binary file, you probably will not be able to make much sense of the output when you display it.

18.3 Writing a binary file

You write to a binary file using the `write()` method. The difference with writing to text files is that you have to supply a byte string as argument, rather than a regular string. The following code creates a binary file with some text in it.

listing1805.py

```
from os.path import getsize

FILENAME = "pc_binarytest.tmp"
fp = open( FILENAME, "wb" )
fp.write( b"And now for something completely different...\x0A\
\x00\x00\x00\x00\xD4\xE8\xE5\xA0\xD3\xF0\xE1\xEE\xE9\xF3\xE8\xA0\
\xC9\xEE\xF1\xF5\xE9\xF3\xE9\xF4\xE9\xEF\xEE\x00\x00\x00" )
fp.close()
print( getsize( FILENAME ), "bytes written" )
```

Run the code above to create the binary file. The code below opens it in text mode (you can do that, as Python cannot know that it actually is a binary file), reads the contents, and prints the contents. You will see some readable text and some unreadable characters.

listing1806.py

```
FILENAME = "pc_binarytest.tmp"
fp = open( FILENAME, encoding="latin-1" )
while True:
    buffer = fp.readline()
    if buffer == "":
        break
    print( buffer )
fp.close()
```

Exercise Change the code above to open the file in binary mode and print the contents.

18.4 Positioning the file pointer

The file "pc_binarytest.tmp" actually contains a few secret words, which you cannot recognize when printing the file. I am going to use them as an illustration on how to move the file pointer.

The file pointer indicates where in the file you start reading or writing. You can move the file pointer with the `seek()` method. `seek()` gets two integer arguments, of which the second one is optional. The first argument is a relative byte position. The second is the position relative to which you want to move the file pointer.

The second argument can be 0, 1, or 2. 0 means "relative to the beginning of the file," 1 means "relative to the current file pointer position," and 2 means "relative to the end of the file." If you do not specify a second argument, it is assumed to be 0. In the `os` module there are constants for this argument: `os.SEEK_SET` is 0, `os.SEEK_CUR` is 1, and `os.SEEK_END` is 2.

The first parameter indicates how many bytes you move from the indicated position. When starting at the beginning of the file, it should be a positive number; when starting at the end of the file, it should be a negative number; when starting somewhere in the middle of the file, it can be positive or negative. For instance, the statement `fp.seek(5)` is equivalent to `fp.seek(5,0)`, which moves the file pointer 5 bytes up from the start of the file, placing it at the sixth byte.

Should you wish to know at which position the file pointer is currently placed, you can use the `tell()` method. Both `seek()` and `tell()` can be called for text files too, but are not very useful then.

Now, the secret words are found starting at position 50, and run for a length of 23 bytes. The encoding is such that if you subtract 128 from byte values, you get the ordinals for the letters. So, here is how you get the words out of the file:

listing1807.py

```
fp = open( "pc_binarytest.tmp", "rb" )
print( "1. Current position of the file pointer is", fp.tell() )
fp.seek( 50 )
print( "2. Current position of the file pointer is", fp.tell() )
buffer = fp.read( 23 )
print( "3. Current position of the file pointer is", fp.tell() )
fp.close()

print( buffer )
s = ""
for c in buffer:
    s += chr( c-128 )
print( "The secret words are:", s )
```

The `seek()` method is particularly useful when you open a file in "reading and writing" mode (`"r+b"`). It allows you to move through the file, reading where you need to read, and (over)writing where you need to (over)write.

Exercise Open the file "pc_binarytest.tmp" in binary "reading and writing" mode, and overwrite the encoded secret words with their decoded translation. Once you have closed the file, open it again in text mode, read the contents, and display them. If you did it all correctly, you should see two readable lines. Should you mess up the file in some way, you can always recreate it.

What you learned

In this chapter, you learned about:

- Using binary files for reading, writing, and reading plus writing
- Binary `read()` and `write()`
- Byte strings
- Conversion between strings and byte strings using `encode()` and `decode()`
- `seek()` and `tell()` methods

Exercises

Exercise 18.1 Create a simple file encryption program. Open a file and read it in binary mode. For each byte, if it is smaller than 128, add 128; if it is bigger than or equal to 128, subtract 128. Overwrite the byte with new value. Test the program on a copy of a text file (make sure it is a copy, because you will destroy the file). Check the contents of the encrypted file: they should be a mess. However, when you run the program again, the original file should be restored. If it isn't, you have a bug in your program. Aren't you glad you were only working on a copy?

Exercise 18.2 The fourteen most common letters in the English language are: `etaoinshrdlcum`. Write a text compression program based on this fact. The compression program stores these letters in half-bytes. A half-byte can take the numbers zero to 15. If you only use the numbers 1 to 15, each number can represent one of these fourteen most common letters, and you can use the number 15 for the space. So you can store two of these letters (or space) in a byte (the value for the whole byte would be 16 times the value for the first letter, plus the second letter). If in the text you encounter a letter that is not amongst these fifteen, you indicate that by storing a zero-half-byte, followed by a whole byte that represents the unencoded letter. Of course, in this setup it is possible that the full byte is actually divided over two bytes, namely the second half-byte of one byte, and the first half-byte of the other byte.

Hint: an easy approach is to build a list of half-bytes. For the most common letters, you store their index-value for the string `"etaoinshrdlcum "` plus 1 (which is a value in the range 1 to 15; notice that the last character in the string is the space). For the other characters, you store three half bytes, namely zero, followed by the ordinal value of the character divided by 16 (rounded down), followed by that ordinal value modulo 16. Once the half-byte-list is finished, you can turn it into a byte list by taking pairs of half-bytes, multiplying the first by 16 and adding the second. Create a byte string using **`bytes`**`()` casting.

half bytes	0	4	8	1	B	B	4	0	2	C	F	0	7	7	4	9	B	A	0	2	1	0
letters	H		e		l	l	o		,			w			o	r	l	d	!			
bytes	\x04		\x81		\xBB		@		,		\xF0		w		I		\xBA		\x02		\x10	

Figure 18.1: Compression example.

For testing: the string `"Hello, world!"`, which is 13 characters in length, will become the 11-character byte string `b'\x04\x81\xbb@,\xf0wI\xba\x02\x10'` if you follow the procedure outlined above (which assigns `e` the value 1, `t` the value 2, etcetera).

A note on the translation of `"Hello, world!"` to the given byte string (see Figure 18.1): You may remember that a hexadecimal representation of a byte consists of two hexadecimal digits, i.e., each digit is a half-byte. Using that information, you can see how the translation has been done. The first byte is `\x04`, i.e., the first half-byte is zero. That means that the first character is given literally, i.e., it consists of the second half-byte of `\x04`, and the first half-byte of the next byte, which is `\x81`. That is the byte `\x48`. If you look up the hexadecimal code 48 in the ASCII table (given in the chapter on strings), you see that that is the character `H`. The following half-byte is the second half-byte of `\x81`, i.e., it is 1. Since this is not zero, it is one of the most common characters, namely the first one, which is `e`. So now you see how `"Hello, world!"` is compressed as the byte string provided. The byte string does contain a few characters that are not displayed as their hexadecimal code; if you really want to know which hexadecimal code they represent, look them up in the ASCII table.

By the way, despite the long description, the whole program needs less than 30 lines of code, including comments, empty lines, and testing statements.

Exercise 18.3 As a collary to the previous exercise, write a decompression program for the produced strings.

Hint: Just do the opposite of what you did in the previous exercise: rebuild the half-byte-list. That list is then easily converted back to the original string.

Exercise 18.4 This chapter is about binary files, and the previous two exercises were not, at least, not directly. There simply is not much that you can exercise with where binary files are concerned; the main problems are with handling byte values, which is what the previous two exercises were concerned with. But to round off what these two exercises did, let's now use what you developed in them to compress files.

Write a program that asks for an input file, that must exist, and an output file, that should not exist. Then it asks whether you want to compress or decompress. If you choose compress, the input file is compressed using the method developed above, and written as the output file. If you choose decompress, the input file is decompressed under the assumption that it was compressed with the method developed above, and written as the output file. So you should be able to get the original file again by first compressing and then decompressing.

You best read the whole file in memory before (de)compressing, so that you do not get into problems when a byte string ends in half a byte instead of a full byte after compression. You also best treat both the input file and the output file as binary files.

Chapter 19

Bitwise Operators

Chapter 18 discussed dealing with binary files. When binary files are used, you are no longer working with characters and numbers; rather, you are working with bytes. To manipulate information on the level of bytes, Python offers a number of so-called "bitwise operators." You will not need these often, but when you delve into binary file manipulation, they might come in handy.

19.1 Bits and bytes

A bit is the smallest size data unit that a computer can handle. A single bit can have only two different values, namely 1 and zero.

While "prehistoric" computers were indeed programmed by directly dealing with single ones and zeroes, very quickly computers were introduced that handled groups of bits. The smallest unit in that respect is the "byte," which consists of 8 bits. Today, the concept of a byte still permeates most computer languages, even though computers have been enhanced to use larger collections of bytes as smallest data units (notably, most computers today either deal with 32-bits or 64-bits data units).

19.1.1 Binary counting

A byte consists of 8 bits, which you can display as a sequence of ones and zeroes, e.g., `11010010`. As such, a byte can be used to represent a number in binary code. If a byte is used to represent a positive number, that number can be calculated by multiplying the right-most bit by 1, the bit next to that by 2, the bit next to that by 4, etcetera, and adding up all those values. For instance, the sequence `11010010` is `1*128+1*64+0*32+1*16+0*8+0*4+1*2+0*1`, which is 210. Note that this is similar to calculating the value of decimal numbers, where the rightmost digit is multiplied by 1, the digit next to that by 10, the digit next to that by 100, etcetera, and adding up all those resulting values. It is also similar to hexadecimal counting, which was discussed in Chapter 10.

When bits are numbered, by convention numbering starts at zero at the rightmost end, and numbers are increased when counting to the left, i.e., the rightmost bit has number 0, the

bit next to that has number 1, the bit next to that has number 2, etcetera. The reason is that the rightmost bit represents the value 2^0 (which, in case you forgot, equals 1), the bit next to it the value 2^1, the bit next to that 2^2, etcetera.

Byte	1	1	0	1	0	0	1	0	
Number of bit	7	6	5	4	3	2	1	0	
Represented value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
Byte value	2^7	$+ 2^6$	$+ 0$	$+ 2^4$	$+ 0$	$+ 0$	$+ 2^1$	$+ 0$	$= 210$

Exercise Write some code that calculates the decimal number represented by a binary string of 8 ones and zeroes. The nicest solution uses a loop, a multiplier, and a total. The total starts at 0. The multiplier (which is the represented value in the example above) starts at 1, and every time the loop is traversed it is multiplied by 2. The loop processes the string from right to left (or the reversed string from left to right), and if the character encountered is a "1," it adds the multiplier to a total. This will end up with the number represented by the string as the total.

The lowest number that can be expressed by a byte is 00000000, which equals zero. The highest is 11111111, which equals 255. Thus, there are 256 different values that can be expressed by one byte.

19.1.2 Character encoding

The most basic character encoding mechanism is ASCII. The ASCII table was shown in Chapter 10, including hexadecimal codes. You may have noticed that the codes used ran from (hexadecimal) 20 to 7E. The codes below 20 are used for special sequences (such as the newline character). The code 7F usually represent the `Del` key. No other codes are in use, which means that all ASCII characters can be represented by 7 bits, or the 8-bit sequences 00000000 to 01111111.

While computers use bytes as basic data unit, the ASCII character set does not use 128 of all the values that can be stored in a byte. All these unused bytes have a 1 as their leftmost bit. Naturally, some character encodings were introduced that assign a character to all 256 different values that a byte can take. A typical one is `latin-1`, which is discussed in Chapter 16. Unfortunately, not all encoding mechanisms assign the same characters to the numbers between 128 and 255. However, all encoding mechanisms in use today at least have the basic ASCII characters for the values 0 to 127.

Python is based on Unicode encoding. Specifically, it uses UTF-8 as character encoding mechanism (discussed in Chapters 10 and 16). UTF-8 encoding works as follows:

- A byte that has a zero as leftmost bit is an ASCII character.
- A byte that has a 1 as leftmost bit is the start of a sequence of multiple bytes that represent one character. The sequence consists of a leading byte (the leftmost byte) and one or more continuation bytes.
- For a multibyte sequence, the leading byte has, from left to right, several bits with value 1, followed by a bit with value zero, followed by the remaining bits. The length of the total multibyte sequence is as many bytes as there are bits with value 1 to the left of the leftmost zero. E.g., if the leading byte has value $1110xxxx$ (where each x

is some bit value), the whole sequence is three bytes long. This includes the leading byte. The minimum sequence length is two bytes, and the maximum sequence length is six bytes (the leading byte will then be 1111110x).

- Each continuation byte has 10 as the two leftmost bits.
- In practice, UTF-8 encoding is restricted to at most 4-byte sequences, and some of the 4-byte sequences have been excluded.

This means that UTF-8 can express a great many different characters. However, it also means that, due to the way characters are encoded, some bit patterns do not express UTF-8 characters. While any bit pattern expresses a legal string with `latin-1` encoding, it is possible to construct a bit pattern that does not express a legal UTF-8 encoding. This may cause those annoying `UnicodeDecodeErrors` when reading files.

19.1.3 Number encoding

The way that numbers are encoded as bit patterns is somewhat tricky, and in general you do not need to bother with it. You should know that positive integers are always encoded as multi-byte patterns, that have a zero as their leftmost bit. The rest of the pattern is like you would expect, and as explained above.

Negative numbers, however, are encoded rather differently. They use the so-called "two's complement" system. When a negative number is encoded, first the absolute value of that number (i.e., the positive version) is taken. From this number, all the bits are "flipped," i.e., every 1 becomes a zero and every zero becomes a 1. Finally, 1 is numerically added to the result. The bit pattern of a negative number therefore always has a 1 as its leftmost bit.

For example, to encode -1, first the bit pattern of 1 is taken, which is ...00000001. All the bits are flipped, which gives ...11111110. Finally, 1 is added the the result, which gives ...11111111. Thus, -1 is encoded as a sequence of only 1s.

As for floating point numbers, these use scientific notation, whereby part of the multi-byte pattern is used as exponent.

The reason that I am explaining all of this, is to indicate that if you want to handle bit patterns in a Python program, and you want to treat these patterns as numbers, you best work only with positive integers, as the bit patterns of those are easily understood.

19.2 Manipulating bits

Python offers a variety of operators that allow the manipulation of data items at the level of bits. These are the following:

```
<<      shift left
>>      shift right
&       bitwise and
|       bitwise or
~       bitwise not
^       bitwise exclusive or
```

They are used as follows.

19.2.1 Shifting bits

When you have a data item, you can use the << and >> to shift its bits to the left or right. `x<<y` shifts the bits of `x` by `y` places to the left, bringing in zeroes from the right. `x>>y` shifts the bits of `x` by `y` places to the right, copying the leftmost bit of `x` at the left while shifting, and losing the bits of `x` at the right. `x` and `y` must both be numbers.

For example, the exclamation mark `!` has decimal code 33, which is written as 00100001 in binary. Shifting this pattern one place to the left gives 01000010, i.e., 66 in decimal, which is the code for the capital B. You can reverse this by shifting the pattern of B one place to the right.

```
code = "!"
print( chr(ord(code)<<1) )
code = "B"
print( chr(ord(code)>>1) )
```

You might have noticed that shifting a number one place to the left amounts to doubling the number, while shifting it one place to the right amounts to halving it (while rounding down). Indeed, you can double the value expressed by a bit pattern by placing a zero to the right of it – and you can halve it (using integer division) by removing the rightmost bit.

```
print( "345 quadrupled makes", 345<<2 )
print( "345 divided by 8 makes", 345>>3 )
```

19.2.2 Bitwise `and`

The bitwise `and` operator (`&`) takes two bit patterns, and produces a new pattern that is all zeroes, except for those places where both bit patterns had a 1, which will then also have a 1 in the output pattern. For instance, if the input patterns are the number 11 (00001011) and the number 6 (00000110), then the bitwise `and` operator produces the pattern 00000010, which is the number 2.

```
print( 11 & 6 )
```

Exercise The bitwise `and` is an easy way to take (positive) numbers modulo a power of 2. For instance, if you want to take a number modulo 16, this is the same as performing the bitwise `and` on the number with 15, which is 00001111. Check that the value of 345 modulo 32 is the same as taking `345 & 31`.

19.2.3 Bitwise `or`

The bitwise `or` operator (`|`) takes two bit patterns, and produces a new pattern that is all ones, except for those places where both bit patterns had a 0, which will then also have a 0 in the output pattern. For instance, if the input patterns are the number 11 (00001011) and the number 6 (00000110), then the bitwise `or` operator produces the pattern 00001111, which is the number 15.

```
print( 11 | 6 )
```

Exercise To set a single bit in a pattern to the value 1 (this is usually called "setting a bit"), you can use the bitwise `or` and a pattern that consists of only zeroes, except for a 1 in the spot where you want to set the bit. An easy way to create a bit pattern with only one bit set, is to start with the number 1, and use the shift-left operator to shift that bit to the left as far as you need. Now take a number and set the bit with index 7 (i.e., the eight bit from the right) to 1.

19.2.4 Bitwise `not`

The bitwise `not` operator (~) is placed in front of a bit pattern, and then produces a new pattern that has all the bits of the original pattern "flipped," i.e., each zero becomes a 1 and each 1 becomes a zero. For instance, if the input pattern is the number 11 (00001011), then the bitwise `not` produces the pattern 11110100, which is the number -12. If you wonder why it is -12 and not -11: this is the result of the two's complement encoding, which I explained above. Don't worry too much about it.

```
print( ~11 )
```

Exercise To clear a single bit in a pattern (i.e., setting it to the value zero), you can use the bitwise `and` and a pattern that consists of only 1s, except for a zero in the spot where you want to clear the bit. An easy way to create a bit pattern consisting on only ones, except for a zero in the intended spot, is to start with the number 1, and use the shift-left operator to shift that bit to the left as far as you need. Then invert the pattern with the bitwise `not` operator. Now take a number and clear the bit with index 3 (i.e., the fourth bit from the right).

19.2.5 Bitwise `xor`

The bitwise `exclusive or`, or "`xor`," operator (`^`) takes two bit patterns, and produces a new pattern that has a zero in all places where the two bit patterns have the same bit, and a 1 in all places where the two bit patterns have different bits. For instance, if the input patterns are the number 11 (00001011) and the number 6 (00000110), then the bitwise `xor` operator produces the pattern 00001101, which is the number 13.

```
print( 11 ^ 6 )
```

Exercise The bitwise `xor` operator provides an easy way to encrypt numbers. Take a bit pattern, and call it the "mask." Apply the mask to a number using the `xor`. This gives a new number, which is the encrypted number. Somebody who does not know the mask, can't tell what the original number was. However, someone who does know the mask, can easily get the original number back, by applying the mask once more. Try this.

19.2.6 Precedence of bitwise operators

Warning: the precedence of bitwise operators is *not* that they are handled before other operators. Make sure that you use parentheses to order the operators when you use bitwise operators in a calculation. For instance, you might think that `1<<1 + 2<<1` is the same as `1*2 + 2*2`, but in actuality it is evaluated as `(1<<(1+2))<<1`, or `1*8*2`.

```
print( 1<<1 + 2<<1 )
print( (1<<1) + (2<<1) )
```

19.3 Usefulness of bitwise operations

Anything that you can do with bitwise operators, you can also do with general calculations, with the advantage of general calculations that they can do much more than bitwise operators. So what is the use of bitwise operators?

Bitwise operations are incredibly fast. Much, much faster than regular calculations. So should you use them when making calculations, when it is opportune to do so? The answer is no, for two reasons:

- Python is already smart enough to recognize that some calculations can be executed using bitwise operators, so it will make the conversion for you.
- If you really want a fast program, you should not use Python at all.

Another use that is often mentioned, is that they facilitate storing boolean values in a small storage space. For instance, if I have eight booleans that I want to store, I can use a tuple of eight booleans, which amounts to at least eight bytes of space, or encode all eight of them in one byte using bitwise operators. However, in today's computers space is of little concern, so only if you are talking about huge, *huge* data collections you might get worried about space.

So what is the use of bitwise operators then? They are actually of fairly little use, unless you have to create programs that need to work "close to the machine." Occasionally you have to deal with data structures that are most naturally handled using bitwise operators. They may also help when you need to manipulate the content of binary files.

To give an example: colors are usually encoded as three bytes, for the red, green, and blue channel. A color number is thus a three-byte number. Bitwise operators are a natural way to distinguish the separate color channels from a color number. Here is a function that does that:

listing1901.py

```
def getRGB( color ):
    blue = color & 255
    green = (color >> 8) & 255
    red = (color >> 16) & 255
    return red, green, blue

r, g, b = getRGB( 223567 )
print( "red={}, green={}, blue={}".format( r, g, b ) )
```

For someone who knows about color encoding, such a function reads well.

What you learned

In this chapter, you learned about:

- Binary counting
- Character and number encoding
- Bitwise operators <<, >>, &, |, ~, and ^

Exercises

Exercise 19.1 Encode a string using the bitwise exclusive or (xor) and the pattern 00101010 as mask. Display the resulting string. Then decode it, and display the decoded string, which should be the same as the original string.

Exercise 19.2 Write a function that gets an integer, a boolean, and a number. The integer is used to store booleans. Each bit in the integer represents **`True`** or **`False`**. The bits of the integer are numbered as the convention indicates, with the rightmost bit having number zero, the bit next to that number 1, etcetera. If the boolean parameter is **`True`**, the function sets the bit corresponding to the number parameter in the integer to 1. If the boolean parameter is **`False`**, the function clears the bit corresponding to the number parameter in the integer (i.e., it sets it to 0). The function then returns the integer.

Also write a function that gets an integer and a number as parameters, and returns **`True`** if the bit corresponding to the number is set to 1, and **`False`** otherwise.

To test the functions, it helps to create an extra function that displays the bits in the number. The display function can make use of the function that gets the bit values.

Chapter 20

Object Orientation

The chapters until this point covered an approach to programming that is often referred to as "structured programming" or "imperative programming," wherein a program is considered a sequence of statements, decisions, and loops. You can solve any programming problem with a structured programming approach. However, in the last decades several other programming "paradigms" have been coined up, which help designing and implementing large-scale programs. One of the most successful paradigms is "object orientation," and most modern programming languages support the object oriented paradigm. Python is, in fact, an object oriented language.

While object orientation tends to provide a natural way to look at problems and solutions, designing an object oriented program can be quite hard. The reason that it is hard, is that you have to really think about your approach to a problem in all of its aspects, before you start coding. For bigger problems, this can be daunting, especially when you lack experience with programming. However, for bigger problems you have to spend a lot of time designing your solution anyway, and an object oriented approach may be quite helpful in creating it. Moreover, you will find that most modules provide object oriented implementations, and that object orientation can be helpful for many smaller problems too.

Since object orientation is a broad topic, several chapters will be spent on it, of which this one is the first. It discusses the basics of object orientation, leaving the more specialized (and powereful!) aspects of object orientation for later chapters.

20.1 The object oriented world

While I am typing this, I am sitting at my kitchen table. Next to me is a bowl of fruit. There are some apples in the bowl. While these apples share certain features, they have their differences too. They share their name, their price, and their age, but they all have (slightly) different weights. There are also some oranges in the bowl. Like the apples, they are fruits, but they have a lot of differences with apples: different names, different colors, different trees that they grow from. Still, they share some things with apples that all fruits share, and make them different from, for instance, the table I am sitting at. I can eat a fruit, i.e., I can eat apples and I can eat oranges. I am not going to try to eat a table.

When I try to model my world in a computer program, I have to model objects: objects such as apples, oranges, and tables. Some of these objects have a lot in common, for instance, each apple shares a lot of features with every other apple. It behooves me to define a class "apple" which contains the features that all apples share, and only fill in the few features in which apples differ from each other for each individual apple object. The same holds for oranges, they should get their own class "orange." And while "apples" and "oranges" are quite different, they still share some features that entail that I would like to put them in the same class: the class "fruit." Every object that belongs to the class "fruit" at least has the property that I can eat it. Which means that each individual apple object not only belongs to the class "apple," but also belongs to the class "fruit" – just like the "oranges."

Come to think of it: I can eat more things than only apples and oranges. I can eat cakes too. And mushrooms. And bread. And licorice. So maybe I need another class, which the class "fruit" also belongs to. The class "food," perhaps?

What this leads to, is that if I try to model the world, or part of the world, I need to model objects – and rather than modeling each separate object, I am better off defining classes of objects, as that means I can make statements about certain groups of objects in general. I can talk about the relationships between classes, and I can define functions that work on classes; for instance, I can define a functionality "eat" that works on every object that is part of the class "food," which removes the object from the world and assigns its "nutrients" to the object that does the "eating." Since I can "eat" objects that belong to the class "food," I can eat "fruit." And since I can eat "fruit," I can eat any "apple" object.

A computer program is, in essence, a model of a part of the world. As such, there are many programs that benefit from the ability to deal with objects, classes, relationships, and functionalities (methods) that work on objects.

20.1.1 Students, teachers, and courses

Many programs deal with persons. The student administration deals with students, who are persons. These students follow courses, which are taught by teachers, who are also persons. Undoubtedly, the student administration stores information on students and teachers, and probably the programmer who created the software for the student administration was smart enough to create a single interface that allows entering person data.

What data do all persons share, as far as the student administration is concerned? Well, probably all persons have a first and a last name. They have an address. They also have an age and a gender. They all get assigned an administration number, so that for the administration they have at least one thing that makes them unique. These data elements are all "properties" or "attributes" of "persons."

I mentioned the properties first name, last name, address, age, gender, and administration number. One of these is actually more like a function than a property. Do you see which one?

The answer is "age." Age is calculated from date of birth and current date. While you can consider age a property, it is a property that should be calculated each time that it is needed. You cannot store it as a value, as tomorrow it might be different from today, without anything changing but the date. Therefore, if I design a class `Person` that models a person, I best make "date of birth" an attribute of the person, while "age" is a method of the person. Remember that methods are functions that belong to a certain data type: if a

data type `Person` is defined, `date_of_birth` is an attribute of that data type, while `age()` is a method of that data type that returns the person's age as an integer.

Students and teachers are both persons. They share the properties of the `Person` class. Yet there are differences. Teachers, for instance, get paid a salary, while students do not. Students, on the other hand, earn grades in courses, which teachers do not; they teach the courses. From this follow two obvious observations:

- While students and teachers are both persons, they have clear differences; besides a class `Person` I need a class `Student` and a class `Teacher`, both of which are derived from the class `Person`.
- "Courses" seem to be an inherent part of the student administration world, so a class `Course` might be needed too.

Once `Course` has become a class in the student administration world, relationships become visible. Students have relationships to multiple courses, and teachers do too, though in a different capacity. Students "enroll" in courses. It looks like an `enroll()` method is needed, that allows a student to get into a relationship with a course. The question is: is `enroll()` a method of `Student`, that gets a course as argument, or is it a method of `Course`, that gets a student as argument? What do you think?

The answer is: "it depends." It depends on how you envision the student administration world. To me, it feels more natural to make `enroll()` a method of a course, as I view a course as a collection of students. However, in principle there is nothing against seeing a student as an entity who encompasses a collection of courses. You might also decide to make `enroll()` a method of each of them, or think of yet another class that contains the `enroll()` method that has both the student and the course as arguments.

This illustrates the difficulty of the object oriented view on program design: by designing the classes that form the world model that the program works with, choices need to be made that may have a big impact on how the program works. Weak choices may lead to difficulties in implementation. You need to spend considerable time on designing the object oriented model that underlies the program, and try to anticipate all the consequences of your choices. This is hard even for experienced designers. However, a solid object oriented model makes programs easy to read, understand and maintain. The object oriented paradigm is often worth the hassle.

20.1.2 Classes, objects, and hierarchies

In the object oriented world, every distinguishable entity belongs to a "class." A class is a general model for a specific group of entities. It describes all the attributes that these entities have, and it describes the methods that the class offers the outside world to influence it.

A class, by itself, is not an entity. An entity that belongs to the class, is an "object." The terminology is that an object is an "instance" of a particular class. While the class describes its attributes, an object that is an instance of the class has values for these attributes. While the class describes the methods that it supports, to execute such a method one needs an object that is an instance of the class to call the method with.

A class is a data type, an object is a value.

Classes may exist in hierarchies. A general, high-level class may describe properties and methods that are shared by different subclasses. Each subclass may add properties, add methods, and even change properties and methods (though in general cannot – and should not – completely remove them). Each subclass may have further subclasses.

For instance, the class `Apple` may be a subclass of the class `Fruit`, which may be a subclass of the class `Food`. This means that where in a program an object of the class `Food` is needed, you can supply an object that is an instance of the class `Food`, but also an object that is an instance of the class `Fruit`, or an object that is an instance of the class `Apple`. This does not work the other way around, though. When, for instance, a function in a program was designed for instances of `Apple`, you cannot use it with instances of `Fruit`, or other subclasses of `Fruit`. While an `Apple` is `Fruit`, `Fruit` is not an `Apple`, and `Apple`s aren't `Oranges`.

Such a hierarchy is implemented using "inheritance," which is the topic of Chapter 22.

20.1.3 Classes and data types in Python

Most object oriented programming languages have some basic data types, and allow you to create classes, i.e., new data types. This was the case for Python up to Python 2. Since Python 3, every data type is a class.

You can recognize some of this by the way that many functionalities of the basic data types are implemented as methods. Remember that a method is always called as `<variable>.<method>()`, contrary to functions that work on a variable, which are called as `<function>( <variable> )`. The fact that when you want to create a lower case version of a string, you effectuate that as `<string>.lower()` already indicates that the string is an instance of a class.

But not only strings are class instances: integers and floats are too. They even have methods, though these are seldom used explicitly. Some methods are used implicitly, e.g., when you add two numbers together with $+$, that is actually a method call. This will be discussed in Chapter 21.

20.2 Classes and objects

Now the basic philosophies of object orientation are out of the way, I am going to discuss how to use object orientation in Python. It starts with creating new classes using the keyword **`class`**.

20.2.1 `class`

A class can be considered a new data type. Once a class is created, you can assign instances of the class to variables. To start simple, I am going to create a class that represents a point in 2D space. I name this class `Point` (the naming of classes is restricted to the same requirements as the naming of variables, and it is convention to let the names of classes start with a capital). Creating this class in Python is incredibly easy:

```
class Point:
    pass
```

The keyword **pass** in the class definition means "do nothing." This keyword can be used wherever you need to place a statement, but you have nothing yet to place there. You cannot just leave it empty or give a comment and nothing else. But as soon as statements are added, you no longer need **pass**.

To create an object that is an instance of the class, I assign to a variable the name of the class, with parentheses after it, as if it is a function call (you can have arguments between the parenthesis, which will be discussed a bit later in this chapter).

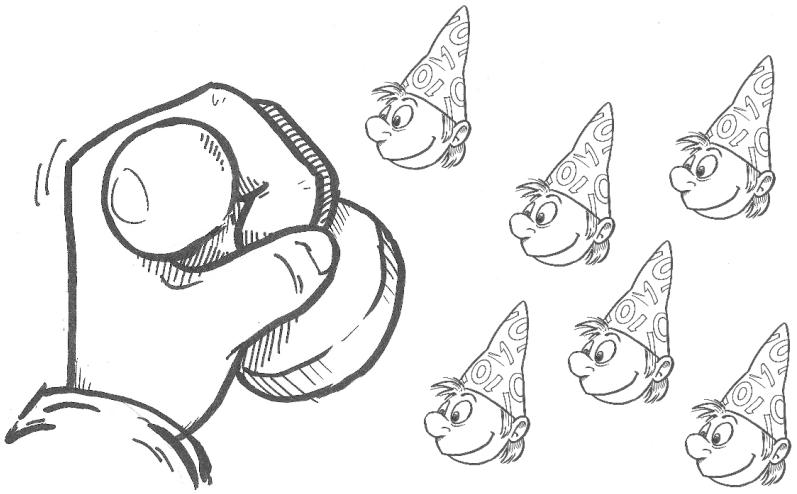
```
class Point:
    pass

p = Point()
print( type( p ) )
```

Of course, a point is more than just an object. A point has an x and a y coordinate. Since Python is a soft-typed language, you need to assign values to attributes to create them. This is done in a special initialization method in the class.

20.2.2 `__init__()`

The initialization method of a class has the name `__init__` (that's two underscores, followed by the word `init`, followed by two more underscores). Even if the `__init__()` method is not defined explicitly for the class, it still exists. You use the `__init__()` method to initialize everything that you want to initialize upon creation of an instance of the class.

In the case of `Point`, `__init__()` should assure that any `Point` object has an `x` and a `y` coordinate. This is implemented as follows:



listing2001.py

```
class Point:
    def __init__( self ):
        self.x = 0.0
        self.y = 0.0

p = Point()
print( "({}, {})".format( p.x, p.y ) )
```

Study the code above closely. You see that `__init__()` is defined just as you would define a function, inside the class definition.

`__init__()` gets one parameter, which is called `self`. Every method that you define, always gets at least one parameter, which will get filled with a reference to the object for which the method is called. By convention, this first parameter is always called `self`. That is not mandatory, but everybody always does it like this. If you forget to include that first parameter, you will get a runtime error. If you forget to include the first parameter `self` but you do have other parameters, Python will fill the first of the parameters that you do list with a reference to the object, and you will probably also get a runtime error (as you did not expect that that would happen).

In the `__init__()` method for `Point`, the object that is created gets two attributes, which are variables that are part of the object. They are called `x` and `y`, and since they are part of the object, you refer to them as `self.x` and `self.y`. They both get initial value 0.0, which makes them floats.

To refer to these attributes when the object has been created, you use the syntax `<object>.<attribute>`, as you can see on the last line of the code, where the object that has just been created is used in a **`print`**`()` statement.

You might wonder if you can only create attributes for an object in the `__init__()` method. The answer is: no, you can create attributes in other methods too, and even outside the class definition.

```
class Point:
    def __init__( self ):
        self.x = 0.0
        self.y = 0.0

p = Point()
p.z = 0.0
print( "({}, {}, {})".format( p.x, p.y, p.z ) )
```

Most Python programmers (including me) would consider what happens in the code above bad form. It is good practice to create all the attributes that you need exclusively in the `__init__()` method (though you can change their values elsewhere), so that you know that every instance of the class has them, and no instances have more.

If you do need a version of the class with extra attributes, you can use “inheritance” to create new classes based on existing ones, which do have these extra attributes. Inheritance

will be discussed in a later chapter. For now, make sure that you create classes with all their attributes defined in the `__init__()` method.

Like any method, `__init__()` can get arguments. You can use such arguments to initialize (some of) the attributes. For instance, if I want to create an instance of `Point` while immediately specifying the values for the `x` and `y` coordinates, I can use the following class definition:

listing2002.py

```
class Point:
    def __init__( self, x, y ):
        self.x = x
        self.y = y

p = Point( 3.5, 5.0 )
print( "({}, {})".format( p.x, p.y ) )
```

`__init__()` is now defined with three parameters. The first is still `self`, as it always has to be there. The second and third are called `x` and `y`. I could have called them anything I like (within the boundaries of variable naming), but I went for `x` and `y` as these are the most logical names. I assign `x` to `self.x`, and `y` to `self.y`.

I call the creation of a point now with values for the `x` and `y` coordinates as arguments. The first argument will be passed to the method in the second parameter, and the second in the third parameter, as the first parameter will be used to pass the reference to the object itself.

If you want to make it optional for the programmer to pass such values, you can give the parameters default values using an assignment in the parameter specification, as follows:

listing2003.py

```
class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y

p1 = Point()
print( "({}, {})".format( p1.x, p1.y ) )

p2 = Point( 3.5, 5.0 )
print( "({}, {})".format( p2.x, p2.y ) )
```

Exercise Create a list of all the points with integer coordinates, with both their `x` and `y` coordinates ranging from 0 to 3.

20.2.3 `__repr__()` and `__str__()`

In the code above, I print the point attributes. What happens if I try to print the point itself?

listing2004.py

```
class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y

p = Point( 3.5, 5.0 )
print( p )
```

Try it, and you will agree that the result is not very informative. When I print a point, I want to see the coordinates. Python offers another predefined method for that, which is `__repr__()`. `__repr__()` should return a string, which contains what you want to see when an object is displayed.

listing2005.py

```
class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

p = Point( 3.5, 5.0 )
print( p )
```

That looks much better.

Python offers yet another standard method for creating a string version of an object, namely `__str__()`. `__str__()` is the same as `__repr__()`, but it is only used when the object is being printed or passed to a **`format`**`()` method. If `__str__()` is not defined, it is the same as `__repr__()` (but not vice versa). If `__str__()` is defined, you can ensure that something different is shown when **`print`**`()` is used, than what is shown in other places.

You now might think: "what other places?" The main "other place" where objects are displayed is in the command shell, when you just type the name of the variable that contains an object.

It is commonly understood that in the `__repr__()` method you are supposed to return a string that contains each and every bit of information that is needed to recreate an object, while in `__str__()` you can just return a string that contains a nicely formatted representation of the most important information that you want to see in the program. Very often, these two are the same.

Many programmers ignore `__repr__()` altogether and only define `__str__()`. I think this is the wrong way around: you should always define `__repr__()`, while `__str__()` is optional. If you use `__repr__()`, make sure that you indeed return all details of an object. If you leave things out, it is better to just use `__str__()`.

Exercise Expand the `Point` class with a `color` attribute. A `color` is represented by a number between 0 and $2^{24} - 1$. Make sure the color is used both in the `__init__()` method and in the `__repr__()` method.

20.3 Methods

I already introduced to you the three methods `__init__()`, `__repr__()`, and `__str__()`. These are predefined methods that every class has. As they were defined by the Python developers, they have the eccentric names that start and end with a double underscore. There are several more of such methods, which I will discuss in later chapters.

You can also define your own methods for a class. Such methods get names similar to names you give to functions, and tend to follow the same conventions: starting with a lower case letter, and if there are different words either have underscores between them or capitalize the first letter of the second and later words. The prefix `is` is used for methods that provide a **`True`**/**`False`** statement about the object, the prefix `get` is used to get a value from an object, and the prefix `set` is used to set a value for an object.

For instance, for a point I can create a method `distance_from_origin()`, which calculates the distance from the point (0,0) to the given point.

listing2006.py

```
from math import sqrt

class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )
    def distance_from_origin( self ):
        return sqrt( self.x*self.x + self.y*self.y )

p = Point( 3.5, 5.0 )
print( p.distance_from_origin() )
```

You may also create methods that change the object in some way. For instance, the "translation" of points over a distance is defined as a specific shift in the horizontal and in the vertical direction. A method `translate()` gets two arguments (beyond the `self` reference), which are the horizontal and vertical shifts.

listing2007.py

```
from math import sqrt

class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )
    def translate( self, shift_x, shift_y ):
        self.x += shift_x
        self.y += shift_y
```

```
p = Point( 3.5, 5.0 )
p.translate( -3, 7 )
print( p )
```

As you can see, I did not specify a return value (I did not need it), but the new `translate()` method made changes to the point coordinates.

Exercise Enhance the `Point` class with a method that turns a point into its polar opposite, i.e., invert the signs of its coordinates, e.g., (3,4) becomes (-3,-4) and (-1,2) becomes (1,-2).

20.4 Nesting objects

Objects can be part of other objects. For instance, a rectangle can be defined as a point that indicates its top-left corner, a width, and a height. As such, the class `Rectangle` can be defined as follows:

listing2008.py

```
class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

class Rectangle:
    def __init__( self, point, width, height ):
        self.point = point
        self.width = width
        self.height = height
    def __repr__( self ):
        return "[{},w={},h={}]".format( self.point, self.width,
            self.height )

p = Point( 3.5, 5.0 )
r = Rectangle( p, 4.0, 2.0 )
print( r )
```

In this definition, the `Rectangle` object contains a `Point` object.

Exercise Create a different version of the `Rectangle` class, that instead of the top-left corner point, width, and height, gets the top-left corner point and the lower-right corner point.

20.4.1 Copies and references

Below is a copy of the code above, expanded with a few extra lines that make a change to the `Point p`.

listing2009.py

```
class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

class Rectangle:
    def __init__( self, point, width, height ):
        self.point = point
        self.width = width
        self.height = height
    def __repr__( self ):
        return "[{},w={},h={}]".format( self.point, self.width,
            self.height )

p = Point( 3.5, 5.0 )
r = Rectangle( p, 4.0, 2.0 )
print( r )

p.x = 1.0
p.y = 1.0
print( r )
```

When you run this code, you see that by changing `p`, the `Rectangle r` is also changed. The point that it contains, is actually a reference to the point that was passed to the `__init__()` method. Like lists, dictionaries, and sets, all the objects that are instances of classes that you define, are "passed by reference" to functions and methods. Therefore, `Rectangle r` gets created with a reference to `p`. In this way you can represent relationships between objects.

You do not always want this. In fact, it is unlikely that you would want a `Rectangle` object to have a relationship with the point that is indicated as its upper left corner. How can you solve that? You can solve it by creating a copy of the object. You can do this using the `copy` module. As discussed before, the `copy()` function of the `copy` module creates a shallow copy; if you want a deep copy, you have to use the `deepcopy()` function. For `Points` this is not needed, as there is no difference between shallow and deep copies of instances of this class.

listing2010.py

```
from copy import copy

class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )
```

```
class Rectangle:
    def __init__( self, point, width, height ):
        self.point = copy( point )
        self.width = width
        self.height = height
    def __repr__( self ):
        return "[{},w={},h={}]".format( self.point, self.width,
            self.height )

p = Point( 3.5, 5.0 )
r = Rectangle( p, 4.0, 2.0 )
print( r )

p.x = 1.0
p.y = 1.0
print( r )
```

20.5 Memory management

The following discussion could have occurred in many earlier places in the book, but I postponed it until now because it tends to first come up when dealing with object orientation and the creation of large objects. However, it is rather technical, and the central message is going to be "don't worry about it," so I did not want to touch upon it earlier. The topic is: what happens in the memory of the computer when you create a large number of big objects?

For instance, the following code creates 10,000 instances of the class `Point`, which it places in a list. You can safely run this code.

listing2011.py

```
class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y

pointlist = []

for i in range( 100 ):
    for j in range( 100 ):
        p = Point( i, j )
        pointlist.append( p )

print( "There are", len( pointlist ), "points in the list" )
```

10,000 is not "large" and `Point` does not produce a "big" object, but you can imagine that something similar is done with many more instances of a bigger object. If you wonder how much memory this program needs, the answer is that that is hard to determine in the code,

but it will be in the neighborhood of several megabytes. Modern computers sport several gigabytes of memory. While you are not allowed to use all those gigabytes, there is room to spare, so you will not run into problems with the code above.

However, what happens with larger lists and bigger objects? At some point, your computer will run out of memory, and what happens then is up to the operating system to resolve. Some operating systems start "swapping" memory by storing part of it on a hard disc, which makes the program very slow. Other operating systems just give an error message. This is why database systems have been developed to deal with large data collections, which cannot fit in memory.

But what about the code below? It creates no less than one million points, and thus seems to need hundreds of megabytes of memory. Still, you can safely run this code.

listing2012.py

```
class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y

totalx = 0
totaly = 0

for i in range( 1000 ):
    for j in range( 1000 ):
        p = Point( i, j )
        totalx += p.x
        totaly += p.y

print( "The totals of x and y are", totalx, "and", totaly )
```

The reason that this code can be run safely is that it does not need hundreds of megabytes of memory. While it creates one million points, each of these points is only needed for a brief time (namely to add its x and y coordinates to some totals). The line `p = Point( i, j )` creates a point and assigns it to variable `p`, but variable `p` is overwritten in the next cycle through the loop. At that moment there are no more references in the program to the previous point, and Python can safely remove it from memory if it needs to free up some memory.

This is a process called "garbage collection." It entails that when a program needs more memory, it checks whether it is currently occupying memory which is no longer accessible because there are no more references to the data which is held in that memory. If that is the case, it re-uses the "garbage" memory. This is a fully automatic process, which you do not need to worry about.

"Garbage collection" is an intelligent process. Imagine, for instance, that you have two objects, A and B, of which A contains a reference to B, and B contains a reference to A. Your program contains a reference to A, but not to B. Neither A nor B will be considered garbage in this case, since B can still be found via A. However, if your program gets rid of the reference to A, both A and B will be considered garbage, even though there are still

references to both of them, but none of these references can be accessed by the program anymore.

This is why the message of this section is "don't worry about memory management," as long as you are aware that loading huge data structures in memory to be accessible all at once (for instance into a gigantic list) may lead to memory problems. In which case you may have to diverge to dealing with data within files or databases, which tends to be a relatively slow process.

What you learned

In this chapter, you learned about:

- Classes and objects
- The keyword **`class`**
- Creating objects
- `__init__()`, `__repr__()`, and `__str__()`
- Methods
- Nesting objects
- Relationships via aliases
- Memory management

Exercises

Exercise 20.1 Create a version of the `Rectangle` class that is safe by assuring that both width and height are positive values (how you do that is up to you). Expand it with methods that calculate its surface area and its circumference. Also provide a method that returns the bottom-right corner of the rectangle as a `Point`. Finally, create a method that gets a second `Rectangle` object as parameter, and returns the overlapping area of the two rectangles as a new Rectangle object (the last one is much harder than the other ones).

Exercise 20.2 A student has a last name, a first name, a date of birth (either a year, month, and day, or a `datetime` object if you took the liberty of studying the `datetime` module already), and an administration number. A course has a name and a number. Students can enroll in courses. Create a class `Student` and a class `Course`. Create several students and several courses. Enroll each student in some of the courses. Display a list of students, showing their number, first name, last name, and age, and per student which courses he or she is enrolled in.

Chapter 21

Operator Overloading

Operator overloading is a powerful feature of object orientation that allows you to integrate your new classes into programs in a natural way. Operator overloading is always based on the definition of some special methods, that have the typical `__<name>__()` structure.

21.1 The idea behind operator overloading

When you write Python programs with basic data types, without thinking you use operators to add, subtract, multiply, and divide values, to compare values, and to apply all kinds of standard functionalities. Such interactions are not defined by default for classes you define yourself, but Python allows you to specify what should happen when one applies such an interaction to instances of your class. This is called "operator overloading."

For instance, suppose that you define a class that represents quaternions.[11] You know that adding and multiplying quaternions are well-defined operations. Therefore, you might want to define what happens when you combine two of your quaternions with a + operator. Python allows you to specify that. In fact, Python allows you to specify what the + operator does for any of your new classes.

Isn't that great? You can define a class `Student`, and then define that if you add two students together with a + operator, that their ages are added up. Wonderful, isn't it? No, it isn't. It obviously makes no sense to add up two students. You might start thinking about what a natural interpretation of adding up two students would entail, but the answer is that everything that you can come up with is far-fetched. You should not define an addition operator for classes which have no natural addition defined. This is one of the dangers of operator overloading: if you apply it without thinking, you get nonsensical programs.

Still, operator overloading has powerful applications. In the rest of this chapter I will introduce some of the most common applications of operator overloading.

By the way, operator overloading is a typical example of "polymorphism," a concept that allows a function to have different results depending on the type of its arguments. Polymorphism is often hailed as one of the powerful features of object orientation.

[11] Quaternions are an extension of complex numbers. They consist of 4-dimensional numbers, with a real factor and three imaginary factors called `i`, `j`, and `k`, with specific definitions for the multiplication of these factors. Details are not important for this book, they are just an example of numbers that are not native to Python.

21.2 Comparisons

In Chapter 20 I discussed that objects can be aliases of each other, but that you can also make actual copies. What happens if you try to compare them?

listing2101.py

```
class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

p1 = Point( 3, 4 )
p2 = Point( 3, 4 )
p3 = p1
print( p1 is p2 )
print( p1 is p3 )
print( p1 == p2 )
print( p1 == p3 )
```

The keyword **is** compares object identities. Since p3 is an alias for p1, p1 **is** p3 returns **True**, while p1 **is** p2 returns **False**. However, since p1 and p2 refer to the same point in 2D space, it would be nice if p1 == p2 would return **True**, i.e., that the == would do a value comparison (as you would expect). It does not. That is not surprising, as Python does not know how to compare the values of Points, and therefore the == does the only comparison that Python knows how to do, namely an identity comparison. However, you can instruct Python how to compare two points using the == operator, by defining an __eq__() method:

listing2102.py

```
class Point:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )
    def __eq__( self, p ):
        return self.x == p.x and self.y == p.y

p1 = Point( 3, 4 )
p2 = Point( 3, 4 )
p3 = p1
print( p1 is p2 )
print( p1 is p3 )
print( p1 == p2 )
print( p1 == p3 )
```

The __eq__() method tells Python, in this case, what to do when two objects of the type

`Point` are compared with ==. It returns **`True`** when their `x` and `y` coordinates are equal, **`False`** otherwise. In this example, the interpretation of the comparison operator == is "overloaded" by defining the `__eq__()` method.

You can also overload the other comparison operators !=, >, >=, <, and <=:

- `__eq__()` for equality (==)
- `__ne__()` for inequality (!=)
- `__gt__()` for greater than (>)
- `__ge__()` for greater than or equal to (>=)
- `__lt__()` for less than (<)
- `__le__()` for less than or equal to (<=).

If you specify the `__eq__()` method but not the `__ne__()` method, the `__ne__()` method will automatically return the opposite of what the `__eq__()` method returns. None of the other methods have such an automatic interpretation.

You are not limited to comparing only objects that are instances of the same class. For instance, when I define a class `Quaternion` that implements a quaternion, I might want to compare a quaternion with an integer or a float. That is possible:

listing2103.py

```
class Quaternion:
    def __init__( self, a, b, c, d ):
        self.a = a
        self.b = b
        self.c = c
        self.d = d
    def __repr__( self ):
        return "({},{}i,{}j,{}k)".format( self.a, self.b,
            self.c, self.d )
    def __eq__( self, n ):
        if isinstance( n, int ) or isinstance( n, float ):
            if self.a == n and self.b == 0 and \
                self.c == 0 and self.d == 0:
                return True
            else:
                return False
        elif isinstance( n, Quaternion ):
            if self.a == n.a and self.b == n.b and \
                self.c == n.c and self.d == n.d:
                return True
            else:
                return False
        return NotImplemented

c1 = Quaternion( 1, 2, 3, 4 )
c2 = Quaternion( 1, 2, 3, 4 )
c3 = Quaternion( 3, 0, 0, 0 )
```

```
if c1 == c2:
    print( c1, "==", c2 )
else:
    print( c1, "!=", c2 )
if c1 == c3:
    print( c1, "==", c3 )
else:
    print( c1, "!=", c3 )
if c3 == 1:
    print( c3, "==", 1 )
else:
    print( c3, "!=", 1 )
if c3 == 3:
    print( c3, "==", 3 )
else:
    print( c3, "!=", 3 )
if c3 == 3.0:
    print( c3, "==", 3.0 )
else:
    print( c3, "!=", 3.0 )
if c3 == "3":
    print( c3, "== \"3\"" )
else:
    print( c3, "!= \"3\"" )
if 3 == c3:
    print( 3, "==", c3 )
else:
    print( 3, "!=", c3 )
```

The implementation of the `__eq__()` method in the code above checks if the value the comparison is made with is a `Quaternion`, an integer, or a float. If so, it makes the comparison and returns **`True`** or **`False`**. If not, it returns `NotImplemented`. `NotImplemented` is a special value that indicates that the comparison has no sensible outcome. While the `__ne__()` method automatically inverts the result of the `__eq__()` method, it will not (and cannot) invert `NotImplemented`.

The last comparison in the code above is noteworthy. It executes comparison `3 == c3`. Normally, when the comparison operator is defined, it will be executed for the left operand, i.e., the comparison operator of the integer 3 is executed, with `c3` as argument. However, integers have not defined the `__eq__()` method for `Quaternion`, and thus this returns `NotImplemented`. If that happens, Python inverts the operands, so in this case executes the comparison `c3 == 3`. This comparison leads to a result as for `Quaternion` the comparison with an integer is defined. The same happens with the `!=` operator. Something similar is done for the other comparison operators, but when the operands are inverted, < is swapped with >, and <= is swapped with >=, just as you would expect.

Exercise In Chapter 20, a `Rectangle` class was defined. Add to this class operators to test for equality of rectangles (two rectangles are equal if they have exactly the same shape), and greater/smaller operators (a rectangle is smaller than another rectangle if it has a smaller surface area). Test the new operators. Note: I am a bit on the fence on whether these

are acceptable definitions for equality and the other comparisons, but for practice they are okay.

There is one special comparison I want to bring up, and that is testing whether an object is **True** or **False**. Many objects are considered to be **False** in particular circumstances; for instance, and empty list evaluates to **False**. This was briefly discussed in Chapter 6.

```
buffer = []
if buffer:
    print( buffer )
else:
    print( "buffer is empty" )
```

You can define your own evaluation of an object that is called when the object is used as condition. This is the `__bool__()` method.

`__bool__()` is called when an object is treated as a condition. It must return **True** or **False**. If `__bool__()` is not implemented, `__len__()` is called (see below), which will evaluate to **False** if `__len__()` returns zero. If neither `__bool__()` nor `__len__()` is implemented, the object is always **True** when used as condition.

21.3 Calculations

There are methods available to define what should happen when you combine an instance of a class with a value using a regular calculation operator. The most important of these are:

- `__add__()` for addition (+)
- `__sub__()` for subtraction (-)
- `__mul__()` for multiplication (*)
- `__truediv__()` for division (/)
- `__floordiv__()` for integer division (//)
- `__mod__()` for modulo (%)
- `__pow__()` for power (**)
- `__lshift__()` for left shift (<<)
- `__rshift__()` for right shift (>>)
- `__and__()` for bitwise **and** (&)
- `__or__()` for bitwise **or** (|)
- `__xor__()` for bitwise `xor` (^)

For example, for quaternions the addition is defined as: $(A + Bi + Cj + Dk) + (E + Fi + Gj + Hk) = (A + E) + (B + F)i + (C + G)j + (D + H)k$. Naturally, you can also add integers and floats to quaternions. This can be implemented as follows:

listing2104.py

```
class Quaternion:
    def __init__( self, a, b, c, d ):
        self.a, self.b, self.c, self.d = a, b, c, d
    def __repr__( self ):
        return "({},{}i,{}j,{}k)".format( self.a, self.b,
            self.c, self.d )
    def __add__( self, n ):
        if isinstance( n, int ) or isinstance( n, float ):
            return Quaternion( n+self.a, self.b, self.c, self.d )
        elif isinstance( n, Quaternion ):
            return Quaternion( n.a + self.a, n.b + self.b, \
                n.c + self.c, n.d + self.d )
        return NotImplemented

c1 = Quaternion( 3, 4, 5, 6 )
c2 = Quaternion( 1, 2, 3, 4 )
print( c1 + c2 )
print( c1 + 10 )
```

If a calculation operator is used with your new class as the right operand, and the left operand does not support the operator with your new class (it returns `NotImplemented`), Python checks if your new class supports the operation as the right operand. For that, you need to implement extra methods, which have the same names as the methods above, but with an `r` in front of the name, e.g., `__radd__()` is the addition operator with corresponding object as the right operand (all the other methods can be created in this way too).

The code above will actually produce a runtime error if you try to calculate `10 + c1` (try it). You will have to implement `__radd__()` to solve that.

listing2105.py

```
class Quaternion:
    def __init__( self, a, b, c, d ):
        self.a, self.b, self.c, self.d = a, b, c, d
    def __repr__( self ):
        return "({},{}i,{}j,{}k)".format( self.a, self.b,
            self.c, self.d )
    def __add__( self, n ):
        if isinstance( n, int ) or isinstance( n, float ):
            return Quaternion( n+self.a, self.b, self.c, self.d )
        elif isinstance( n, Quaternion ):
            return Quaternion( n.a + self.a, n.b + self.b, \
                 n.c + self.c, n.d + self.d )
        return NotImplemented
    def __radd__( self, n ):
        return self.__add__( n )

c1 = Quaternion( 3, 4, 5, 6 )
print( 10 + c1 )
```

As you see, I resolved the problem by making `__radd__()` a direct call to `__add__()`. You might wonder why Python does not do that automatically. The reason is mathematical: while in many cases + is "commutative," i.e., you can exchange the operands without the result changing, this is definitely not always the case. But if your addition operator is commutative, a simple call from `__radd__()` to `__add__()` will do the trick.

For the shorthand operators +=, -=, *=, etcetera, you can also define separate methods. These have the same names as the methods above, but with an `i` in front of the name, e.g., `__iadd__()` implements the += operator (again, for all the other methods you can create this variant too). These methods should actually modify `self`, and also **`return`** the result (usually `self`). If they are not implemented, Python reverts to the regular interpretation, i.e., if a statement is `x += y`, then Python tries to execute `x.__iadd__(y)`, and if that returns `NotImplemented`, it will execute `x = x.__add__(y)`. Thus, in general you do not need to implement methods for the shorthand operators.

Exercise Extend the `Quaternion` class with subtraction. Subtraction works similar to addition, except all the pluses are replaced by minuses. Note that subtraction is not commutative, so you cannot implement `__rsub__()` by a simple call to `__sub__()`. However, it is not hard to implement `__rsub__()`, so make sure that you do it.

21.4 Unary operators

Unary operators are operators which work only on the object itself, so not in combination with another object. A typical example is using the minus (-) sign in front of a number to turn it into a negative number. You can overload some of the unary operators, and also some of the basic functions that work on an object.

- `__neg__()` implements the negation (-) of an object
- `__pos__()` implements placing a plus (+) in front of an object (usually without effect)
- `__invert__()` implements the bitwise **`not`** (~)
- `__abs__()` implements taking the object's absolute value when using the **`abs`**`()` function
- `__int__()` implements taking the (rounded down) integer value of an object when using the **`int`**`()` function; must return an integer
- `__float__()` implements taking the floating-point value of an object when using the **`float`**`()` function; must return a float
- `__round__()` implements rounding using the **`round`**`()` function. An optional second argument can be given to specify the number of decimals; must return an integer or a float
- `__bytes__()` implements representing the object as a byte string. It is in that respect similar to the `__str__()` method which was discussed in Chapter 20

listing2106.py

```
class Quaternion:
    def __init__( self, a, b, c, d ):
        self.a, self.b, self.c, self.d = a, b, c, d
```

```
    def __repr__( self ):
        return "({},{}i,{}j,{}k)".format( self.a, self.b,
            self.c, self.d )
    def __neg__( self ):
        return Quaternion( -self.a, -self.b, -self.c, -self.d)
    def __abs__( self ):
        return Quaternion( abs( self.a ), abs( self.b ),
            abs( self.c ), abs( self.d ) )
    def __bytes__( self ):
        return self.__str__().encode( "utf-8" )

c1 = Quaternion( 3, -4, 5, -6 )
print( c1 )
print( -c1 )
print( abs( c1 ) )
print( bytes( c1 ) )
```

Note: You might think it would be a good idea to also implement the `__int__()`, `__float__()`, and `__round__()` methods, that, respectively, use the **`int`**`()`, **`float`**`()`, and **`round`**`()` functions on `self.a`, `self.b`, `self.c`, and `self.d`. Unfortunately, that cannot be done, as these methods must return integers or floats, and not `Quaternions`. Other than what I propose, I see no sensible interpretation of **`int`**`()`, **`float`**`()`, and **`round`**`()` for `Quaternion`, so these methods should not be implemented.

21.5 Sequences

A special kind of class is the sequence class. You have seen several sequence classes, namely tuples, lists, dictionaries, and sets. Such classes contain a sequence of elements, that can be accessed using indices or keys. You can create such classes yourself, by overloading several methods that support changing or getting information on the elements of the class.

- `__len__()` implements the **`len`**`()` function, which should return an integer that indicates the number of elements in the object.
- `__getitem__()` implements returning the element with the key (or index) that is supplied as argument. This method is called when the object is referred to with a value between square brackets after it, e.g., `x[key]` with `x` the object and `key` the key or index of the element. If `key` is an index and the index is not appropriately referring to an object, then you are supposed to raise an `IndexError` (see Chapter 17). If `key` is something else (as with, for instance, a dictionary) and it is not appropriately referring to an object, then you are supposed to raise a `KeyError`. When `key` is an index, for a complete implementation it should also support slices (implemented as so-called slice objects).
- `__setitem__()` implements assigning a value to an element of the object which has the key or index that is given as argument, and the value as second argument. This method is called when a value is assigned to the object with a key or index value between square brackets after it, e.g., `x[key] = value`.

- `__delitem__()` implements removing from the object the element that has the key or index that is given as argument, when the **`del`** keyword is used, e.g., **`del`** `x[key]`.
- `__missing__()` is called by `__getitem__()`, with the key or index as argument, when the key or index is not referring to an element found in the object. This method is used in particular by subclasses of the Python dictionary.
- `__contains__()` should be given an item (and not a key or index) as argument, and returns **`True`** if the item is found in object, and **`False`** otherwise. It is called when the **`in`** keyword is used to test for the existence of an item.

To demonstrate how these methods work, I have implemented a sequence class that implements a Mesostic Puzzle. In Dutch, such puzzles are known as "Filippines," but outside The Benelux they are not well known. The puzzle consists of a list of questions, each of which is answered by one word. Of each answer, one letter is indicated as "special." The special letters, in order of the questions, provide the solution to the puzzle.

I have defined each of the words for the puzzle as an instance of the class `MesosticWord`, which consists of the answer, the index of the special letter in the answer, and the question. The class `Mesostic` is the complete puzzle, i.e., it is a sequence of `MesosticWords`. I implemented the `__len__()`, `__getitem__()`, `__setitem__()`, and `__delitem__()` methods (the last two are not actually used in the code).

I also implemented two more methods, which demonstrate how the overloaded methods manage to do their job. `display()` displays the puzzle, and uses thereby the **`len`**`()` function and indices on the puzzle object itself. `solution()` displays the solution, also using **`len`**`()` and indices.

listing2107.py

```
class MesosticWord:
    def __init__( self,  word, index, question ):
        self.word = word
        self.index = index
        self.question = question

class Mesostic:
    def __init__( self, name, words ):
        self.name, self.words = name, words
    def __len__( self ):
        return len( self.words )
    def __getitem__( self, n ):
        return self.words[n]
    def __setitem__( self, n, value ):
        self.words[n] = value
    def __delitem__( self, n ):
        del self.words[n]
    def display( self ):
        print( self.name )
        for i in range( len( self ) ):
            print( "{}. {}".format( i+1, self[i].question ),
                end = "  " )
            for j in range( len( self[i].word ) ):
```

```
                if j == self[i].index:
                    print( "* ", end="" )
                else:
                    print( "_ ", end="" )
            print()
    def solution( self ):
        s = ""
        for i in range( len( self ) ):
            s += self[i].word[self[i].index]
        return s

puzzle = Mesostic(
    "The Monty Python and the Holy Grail Mesostic Puzzle",
    [ MesosticWord( "ANTHRAX", 5,
          "Sir Galahad's tale took place in the Castle" ),
      MesosticWord( "PERIL", 2,
          "Sir Robin was thrown into the Gorge of Eternal" ),
      MesosticWord( "RABBIT", 5,
          "Sir Bors was killed by a" ),
      MesosticWord( "SHRUBBERY", 1,
          "The Knights of Ni!'s first demand was to get a" ),
      MesosticWord( "COCONUT", 5,
          "A horse can be replaced by a" ),
      MesosticWord( "MINSTRELS", 5,
          "They were forced to eat Robin's" ) ] )

puzzle.display()
```

Note: It would have been nicer if the stars, which indicate the special letters, were printed in a column. However, depending on the editor that you use, the display format does not always use a fixed letter width, so it is hard to organize that. You may implement a solution for this on your own, if you like (it is not relevant for the theme of the chapter).

Another important method that you can implement for sequence classes is `__iter__()`. This one will be discussed in Chapter 23.

When implementing a sequence class, you should also consider creating a suitable implementation of the `__add__()` method, and possibly a suitable implementation of the `__mul__()` method.

Exercise A `Sentence` is a list of words. A basic `Sentence` class is given below. Implement `__len__()`, `__getitem__()`, `__setitem__()`, and `__contains__()` methods for this class.

listing2108.py

```
class Sentence:
    def __init__( self, words ):
        self.words = words
    def __repr__( self ):
        return " ".join( self.words )
```

```
s = Sentence( [ "There", "is", "only", "one", "thing", "worse",
"than", "being", "talked", "about",
"and", "that", "is", "not", "being", "talked", "about" ] )
print( s )
print( len( s ) )
print( s[5] )
s[5] = "better"
print( "being" in s )
```

What you learned

In this chapter, you learned about:

- Operator overloading
- Overloading comparison operators using `__eq__()`, `__ne__()`, `__gt__()`, `__ge__()`, `__lt__()`, and `__le__()`
- `NotImplemented`
- `__bool__()`
- Overloading calculation using `__add__()`, `__sub__()`, `__mul__()`, `__truediv__()`, `__floordiv__()`, `__mod__()`, `__pow__()`, `__lshift__()`, `__rshift__()`, `__and__()`, `__or__()`, and `__xor__()`
- Righthand versions of overloading calculation operators
- Shorthand versions of overloading calculation operators
- Overloading unary operators `__neg__()`, `__pos__()`, `__invert__()`, `__abs__()`, `__int__()`, `__float__()`, `__round__()`, and `__bytes__()`
- Overloading operators for sequence classes `__len__()`, `__getitem__()`, `__setitem__()`, `__delitem__()`, `__missing__()`, and `__contains__()`

Exercises

Exercise 21.1 A playing card consists of a suit (`"Hearts"`, `"Spades"`, `"Clubs"`, `"Diamonds"`) and a value (`2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `"Jack"`, `"Queen"`, `"King"`, `"Ace"`). Implement a `Card` class. Implement that cards are equal when they have an equal rank, and that the other comparisons use the ranks in the order given above (2 lowest, Ace highest). Test the class.

Exercise 21.2 Use the `Card` class as given above. Now also create a `Drawpile` class. A `Drawpile` consists of a sequence of cards. The cards are supposed to form a pile with the top card having the lowest index, and the bottom card the highest index. Implement the `__len__()` and `__getitem__()` methods. Create an `add()` method to add a card to the draw pile at the bottom, and a `draw()` method to remove the top card from a draw pile and return it. Test the class.

Exercise 21.3 Using the definitions created in the previous exercises, create two draw-piles. The first has the 2 of Diamonds, King of Hearts, and 7 of Clubs (in this order). The second has the 4 of Hearts, 3 of Hearts, and 8 of Spades (in this order). Let the draw piles play "War!" This game is played as follows: Draw the top card from each deck. The highest of these cards goes on the bottom of its own deck, and the other card goes there too. The game continues until there is only one pile left.

Hint: With this setup, the game will take 13 rounds and the first deck wins (it has to, as it contains a card that can never be beaten by the second deck). Do you see what a boring game "War!" is? Why children insist on playing this – with full decks even – I'll never know.

Note: Normally when "War!" is played there are special rules for when two cards have the same rank, but in this case the draw piles contain only cards of a unique rank. You do not have to take into account playing the game where that can happen, though if you want to do that, be my guest.

Exercise 21.4 Implement a `FruitBasket` class. The `FruitBasket` contains fruit items, and it may contain a certain number of each item type. Keep it simple: store the fruit items as a dictionary, with the name of the fruit as key, and the quantity as value. For this exercise there is no need to limit what keys can be, anything can be the name of a fruit. Implement the `__add__()` method to add a piece of fruit to the basket (and it might be a good idea to also implement `__iadd()__`), and implement the `__sub__()` method to remove a piece of fruit from the basket (and `__isub__()` is a good candidate too). Implement the `__contains__()` method to check if a certain kind of fruit is in the basket. Also implement `__getitem__()` to check how much of a piece of fruit there is, `__setitem__()` to add a whole bunch of a piece of fruit at once, and `__len__()` to check how many different pieces of fruit there are in the basket. Note that when nothing more of a piece of fruit remains in the basket, you have to remove the key.

Chapter 22

Inheritance

Inheritance allows you to create new classes based on existing ones, just by indicating the difference. It is an extremely powerful concept that allows for the creation of highly flexible, easily maintainable programs.

22.1 Class inheritance

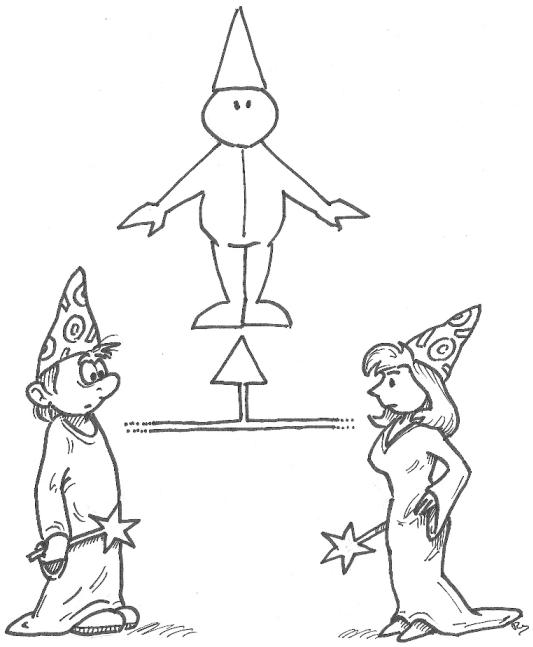
In Chapter 20 I gave the example of `Apple` and `Orange` both being subclasses of a class `Fruit`, and `Student` and `Teacher` both being subclasses of a class `Person`. You can implement such a hierarchy of classes and subclasses using "inheritance."

Basically, inheritance is really simple. When you define a new class, between parentheses you can specify another class. The new class inherits all the attributes and methods of the other class, i.e, they are automatically part of the new class.

listing2201.py

```
class Person:
    def __init__( self, firstname, lastname, age ):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age
    def __repr__( self ):
        return "{} {}".format( self.firstname, self.lastname )
    def underage( self ):
        return self.age < 18

class Student( Person ):
    pass

albert = Student( "Albert", "Applebaum", 19 )
print( albert )
print( albert.underage() )
```

As you can see, the `Student` class inherits all properties and methods of the class `Person`.



22.1.1 Extending and overriding

To extend a subclass with new methods, you can just define the new methods for the subclass. If you define methods that already exist in the parent class (or "superclass"), they "override" the parent class methods, i.e., they use the new method as specified by the subclass.

Often, when you override a method, you still want to use the method of the parent class. For instance, if the `Student` class needs a list of courses in which the student is enrolled, the course list must be initialized as an empty list in the `__init__()` method. Yet if I override the `__init__()` method, the student's name and age are no longer initialized, unless I make sure that they are. You can make a copy of the `__init__()` method for `Person` into `Student` and adapt that copy, but it is better to actually call the `__init__()` method of `Person` inside the `__init__()` method of `Student`. That way, should the `__init__()` method of `Person` change, there is no need to update the `__init__()` method of `Student`.

There are two ways of calling a method of another class: by using a "class call," or by using the **`super`**`()` method.

A class call entails that a method is called using the syntax `<classname>.<method>()`. So, to call the `__init__()` method of `Person`, I can write `Person.__init__()`. I am not limited to calling methods of the superclass this way; I can call methods of any class. Since such a call is not a regular method call, you have to supply `self` as an argument. So, for the code above, to call the `__init__()` method of `Person` from the `__init__()` method of `Student`, you write `Person.__init__( self, firstname, lastname, age )` (I am allowed to use `self` here because every instance of `Student` is also an instance of `Person`, as `Student` is a subclass of `Person`).

Using **`super`**`()` means that you can directly refer to the superclass of a class by using the standard function **`super`**`()`, without knowing the name of the superclass. So to call the `__init__()` method of the superclass of `Student`, I can write **`super`**`().__init__()`. You

do not supply `self` as the first argument if you use **`super`**`()` like this. So, for the code above, to call the `__init__()` method of `Person` from the `__init__()` method of `Student`, you write **`super`**`().__init__( firstname, lastname, age )`.

Of these two approaches, I prefer the use of **`super`**`()`, but only in this specific way: to call the immediate superclass in single-class inheritance. **`super`**`()` can be called in different ways and has a few intricacies, which I will get to below.

In the code below, the class `Student` gets two new attributes: a program and a course list. The method `__init__()` gets overridden to create these new attributes, but also calls the `__init__()` method of `Person`. `Student` gets a new method, `enroll()`, to add courses to the course list. Finally, as a demonstration I overrode the method `underage()` to make students underage when they are not 21 yet (sorry about that).

listing2202.py

```
class Person:
    def __init__( self, firstname, lastname, age ):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age
    def __repr__( self ):
        return "{} {}".format( self.firstname, self.lastname )
    def underage( self ):
        return self.age < 18

class Student( Person ):
    def __init__( self, firstname, lastname, age, program ):
        super().__init__( firstname, lastname, age )
        self.courselist = []
        self.program = program
    def underage( self ):
        return self.age < 21
    def enroll( self, course ):
        self.courselist.append( course )

albert = Student( "Albert", "Applebaum", 19, "CSAI" )
print( albert )
print( albert.underage() )
print( albert.program )
albert.enroll( "Methods of Rationality" )
albert.enroll( "Defense Against the Dark Arts" )
print( albert.courselist )
```

22.1.2 Multiple inheritance

You can create a class that inherits from multiple classes. This is called "multiple inheritance." You specify all the superclasses, with commas in between, between the parentheses of the class definition. The new class now forms a combination of all the superclasses.

When a method is called, to decide which method implementation to use, Python first checks whether it exists in the class for which the method is called itself. If it is not there, it checks all the superclasses, from left to right. As soon as it finds an implementation of the method, it will execute it.

If you want to call a method from a superclass, you have to tell Python which superclass you wish to call. You best do that directly with a class call. However, you can use `super()` for this too, but it is pretty tricky. You provide the order in which the classes should be checked as arguments to `super()`. However, the first argument is not checked by `super()` (I assume that it is supposed to be `self`).

It is something like this: You have three classes, A, B, and C. You create a new class D which inherits from all other three classes, by defining it as `class D( A, B, C )`. When in the `__init__()` method of D you want to call the `__init__()` methods of the three parent classes, you can call them using class calls as `A.__init__()`, `B.__init__()`, and `C.__init__()`. However, if you want to call the `__init__()` method of one of them, but you do not know exactly which, but you do know the order in which you want to check them (for instance, B, C, A), then you can call `super()` with `self` as the first argument and the other three classes following it in the order in which you want to check them (for instance, `super( self, B, C, A ).__init__()`).

As I said, it is pretty tricky. Multiple inheritance is tricky anyway. My general recommendation is that you do not use it, unless there is really no way around it. Many object oriented languages do not even support multiple inheritance, and those that do tend to warn against using it.

So I am not even going to give an example of using multiple inheritance, and neither am I going to supply exercises for multiple inheritance. You should simply avoid using it, until you have a lot of experience with Python and object oriented programming. And by that time, you probably see ways of constructing your programs that do not need multiple inheritance at all.

22.2 Interfaces

An interface is a class that specifies attributes and methods without an actual implementation of the methods. The idea is that subclasses implement the methods, while functions can be defined as working on the interface class, under the assumption that the methods will be filled in. Such functions can then be called with instances of the subclasses.

For good understanding, it is probably better to give an example.

Suppose that I want to design an application that works with vehicles. Maybe it is a travel-planning application that calculates how to get from point A to point B. The application will have a map containing all possible points and connections between the points. It will also have a list of vehicles, with certain vehicles being restricted to specific points, and connecting only specific points (e.g., planes will only be available at airports, and only connect to specific other airports, while boats are only found in harbors and connect to specific other harbors). The application gets a start and end point as input, and provides a list of the sort: take the car to drive from start point to point X, take the plane to fly from point X to point Y, take the bus to drive from point Y to point Z, and then walk from point Z to the end point.

This application will need a definition of vehicles. To be able to come to an optimal travel plan, it must know for each vehicle at what points it is available, to what points it can travel, and the average speed of travel (so that you do not get a travel plan that says "walk from Amsterdam to Moscow"). It might also be a good idea to include a verb that is used when the plan refers to travel with a vehicle (e.g., "walk," "drive," or "fly"). You might need to think a lot about how to implement such vehicles. A possible approach is to supply each vehicle with a method that gets a point as argument and that returns whether or not the vehicle is available at that point, a method that gets a point as argument and that returns whether or not the vehicle travels to that point, a method that gets two points and returns the average speed of travel of the vehicle between those two points, and a method that returns the verb (I am not saying that this implementation is a good idea, just that it could potentially be used).

So you can implement a `Vehicle` class as follows:

listing2203.py

```
class Vehicle:
    def __init__( self ):
        self.startpoint = []
        self.endpoints = []
        self.verb = ""
        self.name = ""
    def __str__( self ):
        return self.name
    def isStartpoint( self, p ):
        return NotImplemented
    def isEndpoint( self, p ):
        return NotImplemented
    def travel_speed( self, p1, p2 ):
        return NotImplemented
    def travelVerb( self ):
        return NotImplemented
```

A class like this is called an interface or "abstract class" (there are subtle differences between interfaces and abstract classes in computational theory, but for Python these do not matter). It is not to be used as a class of which you create instances, which is why all methods return `NotImplemented`. Instead, it is to be used as a template to inherit subclasses from, that will all create implementations for the predefined methods. This means that regardless which vehicle subclass you define later, you will always have to make sure the methods of the `Vehicle` class are implemented. So functions that make use of instances of subclasses of `Vehicle` may count on these methods being available.

What you learned

In this chapter, you learned about:

- Inheritance
- Overriding

- Class calls
- `super()`
- Multiple inheritance
- Interfaces

Exercises

Exercise 22.1 Below I give a `Rectangle` class that is created with the `x` and `y` coordinate of the top-left corner, a width `w`, and a height `h`. Now create a `Square` class that inherits as much as possible from the `Rectangle` class.

exercise2201.py

```
class Rectangle:
    def __init__( self, x, y, w, h ):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
    def __repr__( self ):
        return "[({},{}),w={},h={}]".format( self.x, self.y,
            self.w, self.h )
    def area( self ):
        return self.w * self.h
    def circumference( self ):
        return 2*(self.w + self.h)
```

Exercise 22.2 A `Rectangle` and a `Square` can be considered shapes. There are, of course, different kinds of shapes which are defined differently, but share with rectangles and squares that they have an area and circumference. Define an interface class `Shape`, of which `Rectangle` and `Square` are sub(sub)classes. Also define a class `Circle` that you derive from `Shape`.

Exercise 22.3 In the Iterated Prisoner's Dilemma, two strategies play against each other over multiple rounds. Every round, the strategies can decide to either Coorperate (C) or Defect (D). If both cooperate, they both get 3 points. If both defect, they both get 1 point. If one cooperates and one defects, the one that defects gets 6 points, and the one that cooperates gets nothing. The goal for each strategy is to score as many points as possible.

Below a simple version of the Iterated Prisoner's Dilemma is coded. A strategy to play the game is defined by the class `Strategy`. The main loop lets two strategies play each other for 100 rounds (it is not hard to create a main loop that lets more than two strategies play each other in pairs, but that increases the size of the code quite a bit and is not important for the exercise). `Strategy` has not implemented the `choice()` method. To create a strategy, you inherit a new class from `Strategy`, and code the `choice()` method. Optionally you can also implement the `lastmove()` method, and extend the `__init__()` method.

Implement the following strategies:

- `Random` just plays COOPERATE or DEFECT at random.
- `AlwaysDefect` always plays DEFECT.
- `TitForTat` starts with COOPERATE, then plays what the opponent played on the previous move (the `lastmove()` method gets to see what the opponent played after a choice has been made).
- `TitForTwoTats` starts with two COOPERATEs, then plays DEFECT if the opponent played DEFECT on both the previous two moves, otherwise COOPERATEs.
- `Majority` starts with COOPERATE, then plays what the opponent played on the majority of the previous moves.

If you want to implement more strategies, be my guest. Test out some of the strategies against each other by filling in the assignments for `strategy1` and `strategy2` (do not forget to give them a name between the parentheses).

Note that the shorthand way that I use in this code to write a simple condition (with a statement like `3 if c1 == COOPERATE else 1`), which looks like a list comprehension (see Chapter 12), is just to save some space and make the code a bit more readable. It would be just as well to write the 4 lines of code that would be needed to do this with a regular **if** statement.

exercise2203.py

```
# Iterated Prisoner's Dilemma
COOPERATE = 'C'
DEFECT = 'D'
ROUNDS = 100

class Strategy:
    def __init__( self, name="" ):
        self.name = name
        self.score = 0
    def choice( self ):
        # Should return COOPERATE or DEFECT
        return NotImplemented
    def lastmove( self, mymove, opponentmove ):
        # Gets passed the last move made, after a call of choice()
        pass
    def incscore( self, n ):
        self.score += n

strategy1 = Strategy()
strategy2 = Strategy()

for i in range( ROUNDS ):
    c1 = strategy1.choice()
    c2 = strategy2.choice()
    if c1 == c2:
        strategy1.incscore( 3 if c1 == COOPERATE else 1 )
```

```
        strategy2.incscore( 3 if c2 == COOPERATE else 1 )
    else:
        strategy1.incscore( 0 if c1 == COOPERATE else 6 )
        strategy2.incscore( 0 if c2 == COOPERATE else 6 )
    strategy1.lastmove( c1, c2 )
    strategy2.lastmove( c2, c1 )

print( "End score of", strategy1.name, "is", strategy1.score )
print( "End score of", strategy2.name, "is", strategy2.score )
```

Chapter 23

Iterators and Generators

Iterators allow your classes to be used in **for** ... **in** ... statements. Generators are an easy way to create iterators.

23.1 Iterators

You have used the **for** ... **in** ... command on many occasions. You may have noticed that it can be used for many different applications.

listing2301.py

```
for i in [1,2,3,4]:
    print( i, end=" " )
print()
for i in ( "pi", 3.14, 22/7 ):
    print( i, end=" ")
print()
for i in range( 3, 11, 2 ):
    print( i, end=" ")
print()
for c in "Hello":
    print( c, end=" " )
print()
for key in { "apple":1, "banana":3 }:
    print( key, end=" " )
```

List, strings, and dictionaries are all "iterables," which means they can be used in such **for** ... **in** ... expressions. Many other objects can also be used as iterables. You can actually ensure that your own classes can be used as iterables as well.

An "iterator" is an object that returns a new item every time you call the **next()** function with the object as argument. When there are no items left, it raises a `StopIteration` exception. If you want to avoid the exception, you can give an optional second argument to **next()**, which is returned when the iterator is exhausted. You can turn an iterable into an iterator object using the built-in function **iter()**.

```
iterator = iter( ["apple", "banana", "cherry"] )
print( next( iterator, "END" ) )
print( next( iterator, "END" ) )
print( next( iterator, "END" ) )
print( next( iterator, "END" ) )
```

You can use iterators in **for ... in ...** statements.

```
iterator = iter( ["apple", "banana", "cherry"] )
for fruit in iterator:
    print( fruit )
```

23.1.1 Iterable objects

An object that should function as an iterable has two elements:

- a method `__iter__()` that returns the object itself
- a method `__next__()` that provides access to all the items that the object contains, one by one, and when no more objects are left, raises `StopIteration` (in a **for ... in ...** loop, this will cause the loop to end)

You can loop over all the items of the iterable using **for ... in ...**. There are three main ways that you can create such an iterable object. The first two ways start with the iterable as a container of a sequence of items.

The first way, when `__next__()` is called, removes one of the items and returns it, after which the iterable holds one less item. Once all items are "consumed," it can only raise `StopIteration`. Here is an example of such an iterator that contains the first 10 numbers of the Fibonacci sequence.

listing2302.py

```
class Fibo:
    def __init__( self ):
        self.seq = [1,1,2,3,5,8,13,21,34,55]
    def __iter__( self ):
        return self
    def __next__( self ):
        if len( self.seq ) > 0:
            return self.seq.pop(0)
        raise StopIteration()

fseq = Fibo()
for n in fseq:
    print( n, end=" " )
```

The second way keeps track of an index in the sequence of items, increasing it every time `__next__()` is called, and raises `StopIteration` when the index goes beyond the range of

items. In the second way, you can implement a method that resets the index so that the iterable can be used again.

listing2303.py

```
class Fibo:
    def __init__( self ):
        self.seq = [1,1,2,3,5,8,13,21,34,55]
        self.index = -1
    def __iter__( self ):
        return self
    def __next__( self ):
        if self.index < len( self.seq )-1:
            self.index += 1
            return self.seq[self.index]
        raise StopIteration()
    def reset( self ):
        self.index = -1

fseq = Fibo()
for n in fseq:
    print( n, end=" " )
print()
fseq.reset()
for n in fseq:
    print( n, end=" " )
```

The third way has the iterable not as a container of items, but as a calculator that every time that `__next__()` is called, determines the next item. Such an iterable can either be finite, or have the ability to supply an infinite number of items. It can also be reset if the programmer supplied a method to do that.

listing2304.py

```
class Fibo:
    def reset( self ):
        self.nr1 = 0
        self.nr2 = 1
    def __init__( self, maxnum=1000 ):
        self.maxnum = maxnum
        self.reset()
    def __iter__( self ):
        return self
    def __next__( self ):
        if self.nr2 > self.maxnum:
            raise StopIteration()
        nr3 = self.nr1 + self.nr2
        self.nr1 = self.nr2
        self.nr2 = nr3
        return self.nr1
```

```
fseq = Fibo()
for n in fseq:
    print( n, end=" " )
print()
fseq.reset()
for n in fseq:
    print( n, end=" " )
```

Warning: You have to be very careful when making an iterable that in principle may return an infinite number of items. Programmers count on **for ... in ...** never leading to an endless loop, but in the example above, without limiting the number of items to a maximum of 1000 when creating the `fseq` object, an endless loop would result. It is best to force the programmer to set a maximum to the number of items.

Exercise Create an iterator that generates all the squares of integers between 1 and 10. You may choose whichever approach you prefer.

23.1.2 Delegated iteration

In the examples above, the iterable was created by calling the `__iter__()` method for the object, which returned itself. That is not needed. An iterable may delegate[12] the iteration to another object, that it creates and returns when `__iter__()` is called.

listing2305.py

```
class FiboIterable:
    def __init__( self, seq ):
        self.seq = seq
    def __next__( self ):
        if len( self.seq ) > 0:
            return self.seq.pop(0)
        raise StopIteration()

class Fibo:
    def __init__( self, maxnum=1000 ):
        self.maxnum = maxnum
    def __iter__( self ):
        nr1 = 0
        nr2 = 1
        seq = []
        while nr2 <= self.maxnum:
            nr3 = nr1 + nr2
            nr1 = nr2
            nr2 = nr3
            seq.append( nr1 )
        return FiboIterable( seq )
```

[12]The name "delegated iteration" I came up with myself. If there is an "official" name for the approach, I gladly update the book.

```
fseq = Fibo()
for n in fseq:
    print( n, end=" " )
print()
for n in fseq:
    print( n, end=" " )
```

This approach has several advantages:

- You can run several instances of the iterable in parallel without the need to explicitly create more than one (as they are created automatically when needed, i.e., when you use **for** ... **in** ...)
- You do not need to call a `reset()` method to start from the beginning
- The delegated iterable is automatically erased from memory after it is used up (Python automatically frees up memory of objects that the program no longer can refer to)

23.1.3 `zip()`

You can create tuples that contain the items of multiple iterables using the standard function **zip**(). To give a simple example:

```
z = zip( [1,2,3], [4,5,6], [7,8,9] )
for x in z:
    print( x )
```

A zip-object is an iterator, i.e., you cannot print the zip-object itself, but you have to loop over its elements instead. The *i*th element of the zip-object consists of the *i*th elements of each of the iterables that are its arguments. If these iterables are of unequal length, the number of elements in the zip-object will be the same as the number of elements of the shortest of the iterables.

In the example above, I just used lists as arguments. But you can use any kind of iterable as argument. For example, the following code block I zip together a range, an iterator, and a list comprehension:

listing2306.py

```
class Doubles:
    def __init__( self ):
        self.seq = [2*x for x in range( 1, 11 )]
    def __iter__( self ):
        return self
    def __next__( self ):
        return self.seq.pop(0)

seq = zip( range( 1, 11 ), Doubles(), [3*x for x in range(1,11)])
for x in seq:
    print( x )
```

Exercise Create a zip-object that produces tuples of two items: the first item is an integer, which runs from 1 to 10. The second item is the square of that integer.

23.1.4 `reversed()`

The built-in function **`reversed`**`()` creates an iterator from an iterable that processes the items of the iterator in reversed order. It gets the iterable as argument.

Not all iterables can be reversed, but the ones that are part of the standard Python specification (such as lists) can. For details on how to make sure that iterables that you create yourself can be **`reversed`**`()`, study the Python documentation.

```
fruitlist = ["apple", "orange", "cherry", "banana"]
for fruit in reversed( fruitlist ):
    print( fruit )
```

23.1.5 `sorted()`

The built-in function **`sorted`**`()` creates an iterator from an iterable that processes the items of the iterator in sorted order. It gets the iterable as argument.

Moreover, it can get two optional arguments. The first is `key=<key>`, where `<key>` is the name of a function that is used to determine the key for the sorting process. This works exactly as the `key=<key>` parameter for the list `sort()` method – see Chapter 12 for more information. If no key is given, the sorting is alphabetical order for strings, and numerical order for numbers. For other data types, or mixed data types, it depends on the specification of the key argument.

The second optional argument is `reverse=<boolean>`, that indicates with **`True`** or **`False`** whether or not the sorting should give a reversed result.

```
fruitlist = ["apple", "orange", "cherry", "banana"]
for fruit in sorted( fruitlist ):
    print( fruit )
```

23.2 Generators

A generator is a function that emulates the behavior of an iterable object. In general, implementing a generator is shorter and easier than creating an iterable. Several standard functions are implemented as generators, for example **`range`**`()`.

Generators are based on the **`yield`** keyword. When calling `__next__()` on a generator, the function is executed until **`yield`** is reached, then the value that is associated with **`yield`** is returned. At that point, the function "waits" until `__next__()` is called again, after which it continues until **`yield`** is reached again. `StopIteration` is raised automatically when the function ends.

There is no need to explicitly define __next__() and/or __iter__(). A function is a generator simply because it contains the **yield** keyword, and the associated iterable object is automatically created by Python, including appropriate implementations for __next__() and __iter__().

listing2307.py

```
def fibo( maxnum ):
    nr1 = 0
    nr2 = 1
    while nr2 <= maxnum:
        nr3 = nr1 + nr2
        nr1 = nr2
        nr2 = nr3
        yield nr1

fseq = fibo( 1000 )
for n in fseq:
    print( n, end=" " )
print()
for n in fseq:
    print( n, end=" " )
```

23.2.1 Generator expressions

In Chapter 12, I introduced the concept of list comprehension. Since any list can be turned into an iterator, and thus into a generator, Python introduced a similar concept for generators, and calls it "generator expressions." The syntax for a generator expression is the same as for a list comprehension, except that the square brackets are replaced by round brackets.

For example, the following generator expression returns all squares up to 100:

```
seq = (x*x for x in range( 11 ))
for x in seq:
    print( x, end=" " )
```

If you just replace the outer two parentheses by square brackets in the generator expression, the code runs with seq being the result of list comprehension. To be absolutely clear about it: with list comprehension the whole list is generated at once, while with a generator expression the items are generated when needed. Thus, in principle a generator expression is preferable, as it saves memory.

23.3 itertools module

The itertools module contains a collection of functions that allow advanced manipulation of iterators. Taken to the extreme, they allow for a sort of "iterator algebra" that can be used to implement specialized tools in Python. Here I just highlight a few of the basic functions from itertools that you might find handy at times.

23.3.1 chain()

`chain()` takes two or more iterables as arguments and functions as an iterable that works through them in sequence.

```
from itertools import chain

seq = chain( [1,2,3], [11,12,13,14], [x*x for x in range(1,6)] )
for item in seq:
    print( item, end=" ")
```

23.3.2 zip_longest()

`zip_longest()` works like **`zip()`**, but will create an iterable that generates as many elements as there are elements in the longest argument. You specify a `fillvalue=` argument to indicate what value should be used for empty spots.

```
from itertools import zip_longest

seq = zip_longest( "apple", "coconut", "banana", fillvalue=" ")
for item in seq:
    print( item )
```

23.3.3 product()

`product()` creates an iterable that produces all elements of the Cartesian product of the iterables that are given as its arguments. To put that in less mathematical terms: if two iterables are given as arguments, and the first has elements *x*, *y*, and *z*, while the second has elements *a* and *b*, `product()` produces *xa*, *xb*, *ya*, *yb*, *za*, and *zb*.

```
from itertools import product

seq = product( [1,2,3], "ABC", ["apple","banana"] )
for item in seq:
    print( item )
```

23.3.4 permutations()

`permutations()` gets an iterable as argument, and an optional second argument that indicates a length. It creates an iterable that produces all permutations of the elements of the first argument of the given length. If no length is given, it generates all permutations that contain all the elements. Note that if the iterable has certain elements multiple times, you will get copies of permutations.

```
from itertools import permutations
```

```
seq = permutations( [1,2,3], 2 )
for item in seq:
    print( item )
```

23.3.5 `combinations()`

`combinations()` gets an iterable as argument, and a second argument that indicates a length. It creates an iterable that produces all combinations of the elements of the first argument of the given length. The length is *not* optional (which is logical, if you think about it for one moment – for maximum length there is only one combination). The elements of the combinations will be in the order that they appeared in the original iterable. Note that if the iterable has certain elements multiple times, you will get copies of combinations.

```
from itertools import combinations

seq = combinations( [1,2,3], 2 )
for item in seq:
    print( item )
```

23.3.6 `combinations_with_replacement()`

`combinations_with_replacement()` works like `combinations()`, except that each element of the iterable can be used multiple times.

```
from itertools import combinations_with_replacement

seq = combinations_with_replacement( [1,2,3], 2 )
for item in seq:
    print( item )
```

What you learned

In this chapter, you learned about:

- Iterators
- Iterables
- `__iter__()`, `__next__()`, and `StopIteration`
- Different approaches to implementing iterable objects
- **`zip`**`()`
- Generators
- **`yield`**
- Generator expressions
- `itertools` functions `chain()`, `zip_longest()`, `product()`, `permutations()`, `combinations()`, and `combinations_with_replacement()`

Exercises

Exercise 23.1 Create a program that asks the user to enter positive integers. The user can enter as many as desired, and indicates that the last integer was entered by supplying zero. The program the prints all numbers between 1 and 100 that are not divisible by any of the integers entered. Print those numbers in a **`for ... in ...`** loop, using an iterator to produce the numbers.

Exercise 23.2 Create a generator that produces factorials. The first value returned is 1!, the second 2!, the third 3!, etcetera, up to 10!. Do not calculate the factorial every time from scratch, but retain the value that you used in the previous cycle and use that.

Exercise 23.3 Ask the user to enter a word. Produce all anagrams of that word. If the word contains multiple copies of a letter, it is acceptable if you produce certain anagrams multiple times. For example, if the word is `"ape"`, you produce `"aep"`, `"ape"`, `"eap"`, `"epa"`, `"pae"`, and `"pea"` (in any order).

Exercise 23.4 Do the previous exercise, but now make sure that all anagrams are unique, even if the word contains repetitions of letters. For example, if the word is `"bee"`, you produce `"bee"`, `"ebe"`, and `"eeb"`.

Exercise 23.5 The "subset sum" problem asks the question whether a list of integers contains a subset of integers that, when summed, gives zero as answer. For instance, for the list [1, 4, -3, -5, 7] the answer is "yes," as 1 + 4 - 5 = 0. However, for the list [1, 4, -3, 7] the answer is "no," as there is no subset of integers that adds up to zero. Write a program that solves the "subset sum" problem for a list of integers. If there is a solution, print it; if not, report that there is no solution.

This is a repetition of one of the exercises of Chapter 12 (Lists). In that chapter I said that you have to solve the exercise recursively. However, using the `itertools` module, you can now solve it without recursion (though I suspect that recursion still is used within the `itertools` module – you, however, do not have to).

Exercise 23.6 Write a program that produces all possible sub-dictionaries from a dictionary, and stores them in a list. For instance, if the dictionary is `{"a":1,"b":2}`, the program produces `[{},{"a":1},{"b":2},{"a":1,"b":2}]` (the ordering of the list does not matter). Again, use the `itertools` module.

Exercise 23.7 Write a program that determines how you can place eight queens on a chess board in such a way that none of them attacks any of the other ones. This is a classic problem that sounds like it has little to do with this chapter, but when you consider that you may solve it using the `permutations()` function in a smart way, you will find that this program can be surprisingly short.

Chapter 24

Command Line Processing

In Chapter 15 I mentioned that when handling large volumes of data spread over multiple files, you occasionally might want to write a Python program that supports command-line processing. This chapter tells you how to do that.

24.1 The command line

Chapter 15 explained how you can get access to your computer's "command prompt". If you do not remember, please go refresh your memory. The chapter also explained that you can start Python programs directly from the command prompt with the command:

```
python <programname>.py
```

This works as long as your system knows how to find Python and the program resides in the current directory. If the program is not in the current directory, it will still work as long as you specify the complete path to the program in the command.

24.1.1 Batch processing

Suppose you have written a program that asks a user for a filename and maybe for a few extra parameters, then processes the file with the indicated name using the parameters. I then ask you to run that program on all the files in a specific directory. The directory contains over 10,000 files. What will you do?

You might adapt the program so that instead of asking for the name of file and processing that file, it processes all files in a directory, using parameters that are not asked of the user. If the parameters that you have to use differ per file, there has to be a way to know what they are (perhaps you can derive them from the file name), so it should be possible to calculate them. This solves your problem. But then I ask you to run your program on a bunch of different directories. What will you do?

You can adapt your program so that it contains a list of all the directories that you need to process, then work through them one by one. And every time that I ask you to add an extra directory, just change the program. Regardless what I ask you, you can always adapt

your program to encompass my requests. Though you might get a bit annoyed that I am asking you to change the program again and again and again.

There is a different way of handling such requests, and that is through batch processing. All command lines support the running of so-called "batch files," which contain a list of commands that you give to the operating system. Under Windows such files have the extension `.bat`, while under Mac and Linux they can have any name. However, you can install different command shells which use different conventions – potentially you could even use a Python shell and use Python itself to write batch files.

The batch file can contain commands that use a close derivative of your first program, that processes just one file, and call it for every file that you want to process. The problem is, of course, that the program requires user input, and you are not prepared to type inputs every time the batch file calls the program. The solution is to change the program in such a way that it processes command line arguments.

24.1.2 Command line arguments

On the command line, you can start a Python program with a list of arguments:

```
python <programname>.py <argument_1> <argument_2> ... <argument_n>
```

The arguments are separated from each other by spaces and can be anything, though if you have an argument that contains spaces, you should enclose it in double quotes. This, of course, immediately raises the question what you should do if an argument contains a double quote, and unfortunately the answer is "that depends on the command shell that you are using." Most commonly, you either precede the double quote with a backslash, or you write a double double quote.

Of course, just writing the arguments will not let your program process them automatically. You have to extend your program with code that processes command line arguments, which means you have to be able to access those command line parameters directly.

24.1.3 `sys.argv`

You get access to the command line arguments that your program was supplied with via a pre-defined list that is available when you import the `sys` module. The list is called `sys.argv`. It is a list of strings, each string being one of the command line arguments that was supplied to the program.

`sys.argv` always contains at least one element, namely the complete name of the Python file that you are running, including the directories and (under Windows) the drive letter of the place where the program file resides. To know how many arguments were supplied (including the filename), you can, of course, use the **`len`**`()` function.

When you work from an editor, in general you cannot supply command-line arguments to your Python programs. So if you want to experiment with this, you actually have to test your programs on the command line. That is a bit of a hassle, especially during development time. However, I can tell you how to set up your programs in such a way that handling command line arguments is optional.

24.2 Flexible command line processing

When I code Python programs, I prefer working from an editor. There are a few editors that support supplying command line arguments to a program when testing it, but most do not. So I want to develop my programs in such a way that I can build them as if they process command line arguments, but that I can test them directly from an editor, and only need to test whether or not they actually process command line parameters correctly once. I do that as follows:

For every parameter that can be controlled via the command line, I create a global variable. I fill these global variables with default values. In the rest of the program, I use these variables as if they are constants. Only at the very start of the main program I check for the existence of command line arguments, and if I find those, I overwrite the variables with values that are supplied on the command line.

The advantage of this approach is that I can develop my program without using command line arguments. If I want to use different values for the command line arguments for testing, I simply use different values for the variables in which I will store the command line arguments. I can even set up the program in such a way that it will either fill a variable via a command line argument, or will ask the user for the value if it was not supplied on the command line.

Typically, such code looks as follows:

listing2401.py

```
import sys

# 3 variables for holding the command line parameters
inputfile = "input.txt"
outputfile = "output.txt"
shift = 3

# Processing the command line parameters
# (works with 0, 1, 2, or 3 parameters)
if len( sys.argv ) > 1:
    inputfile = sys.argv[1]
if len( sys.argv ) > 2:
    outputfile = sys.argv[2]
if len( sys.argv ) > 3:
    try:
        shift = int( sys.argv[3] )
    except TypeError:
        print( sys.argv[3], "is not an integer." )
        sys.exit(1)
```

In this code, three command line arguments are supported: the first two are strings, and the third is an integer. The third one is immediately converted from a string (which a command line argument always is) to an integer, and the program is aborted if this conversion fails. I could have built in more checks for demonstration, but I assume that at this point in your programming career that does not pose any problems to you.

All three arguments have a default value: the first string has default value `"input.txt"`, the second string has default value `"output.txt"`, and the integer has as default value 3. You are not expected to supply all three arguments on the command line: if you supply zero, all three default values will be used; if you supply one, the first string will be overwritten with the command line argument, while the other two variables will retain their default value; etcetera.

24.2.1 `sys.exit()`

In the example code above the program is aborted using `sys.exit()` if an argument is not meeting the requirements set to it. `sys.exit()` was introduced in Chapter 6. However, I did not explain at that time that `sys.exit()` can get a numerical argument, which you see above. The use of this argument is that it will be returned to the batch file that is running the program, and the batch file can respond to it if it was designed to. The numerical argument is supposed to be an error code. Typically, zero is given as argument if everything was processed correctly (a program that ends normally will also return zero), and otherwise another number is used. As some systems are limited to the values zero to 255 as program return values, it is convention to limit the `sys.exit()` argument to that range of values too.

24.2.2 `argparse`

If you want module support for command line processing, Python supplies a standard module `argparse` for that. Frankly, I do not see much use for such a module, as command line processing is too simple to spend much time on. However, some Python programs, in particular those that are meant to enhance the system, support a large and complex variety of command line arguments, and may benefit from such a module. It is up to you to study it if you think it may help you.

What you learned

In this chapter, you learned about:

- Command line arguments
- `sys.argv`
- Flexible command line processing

Exercises

Exercise 24.1 Create a program that can be started with zero or more numerical arguments. If it gets presented with a non-numerical argument, the program gives an error message. If it gets presented with only numerical arguments, it adds them up and prints the sum. Test the program on the command line.

Chapter 25

Regular Expressions

When you are facing a problem with text mining, data processing, finding patterns in large collections of data, or scraping data from web pages, and you explore the Internet to find a solution to your problem, you will find that often the very first answer given to questions in this respect is "Why don't you use regular expressions?" or even "Just use regex," without further explanations. Rather smug answers, as many people have never heard of regular expressions, and if they have, might find them scary and incomprehensible. In fact, at first glance they come over as so arcane and confusing that most people rather shy away than delve into them. Which is a pity, as regular expressions are a powerful tool that should not be missing in the toolbox of anyone who deals with unstructured data on a regular basis.

In this chapter I will explain how to write and use basic regular expressions with Python. You will find them indeed a powerful way to quickly express and discover complex and diverse patterns in data, providing access to functionalities that would be very hard to implement in vanilla Python. While this chapter does not contain a complete overview of regular expressions, after studying it you will be able to understand and use regular expressions for most, if not all, pattern-matching problems that you encounter in practice, and be confident in telling the uninitiated: "You should use regular expressions to solve your problems." Now you can feel smug too!

25.1 Regular expressions with Python

Regular expressions are text strings that describe a "pattern" that can be found in textual data. For example, the regular expression `a+` describes a pattern that consists of a sequence of one or more times the letter "a." In the string "aardvark" this pattern can be found twice, namely as the "aa" at the start of the string, and the single "a" in the second half of the string.

A regular expression always consists of a string, which may contain any character. Some characters are "meta-characters" which have a special meaning in regular expressions. You should be careful when using them (how you should use them will be discussed later). The meta-characters are:

```
. ^ $ * + ? { } [ ] \ | ( )
```

I will discuss how to write regular expressions later in this chapter. First, I need to discuss how to use regular expressions in Python code.

25.1.1 The `re` module

To use regular expressions in Python, you must import the `re` module.

A regular expression can be considered a piece of code. That code can be "compiled" by the `re` module to produce a "pattern object." That pattern object can then be used to search for the pattern in data. For instance, in the following code, the regular expression `a+` is compiled to produce a pattern stored as `pAplus`, which is then used to search for the pattern in the string "aardvark." It stores the occurrences of the pattern as a list, and prints that list.

listing2501.py

```
import re

pAplus = re.compile( r"a+" )
lAplus = pAplus.findall( "aardvark" )
print( lAplus )
```

Exercise You can change the word "aardvark" into something else, and see how that affects the output.

You might be wondering what that letter "r" is doing in front of the regular expression string. Why did I write `r"a+"` instead of just `"a+"`? This letter "r" tells Python that it should consider the string as "raw data," i.e., it should not try to convert parts of the string according to standard Python string interpretations. This is mainly necessary when the regular expression contains `"\b"`, which for regular expressions means "word boundary" (I will get to that later in this chapter), but for Python is an escape sequence that means "backspace." So it is good practice to always put that "r" in front of a regular expression, to avoid problems.

While it is seldom done in practice, you may add an optional second parameter (a so-called "flag") to the `compile()` call, which indicates a special way to use the created pattern. The parameter `re.I` indicates that the pattern should be used case-insensitively, while `re.S` indicates that the pattern should also process newlines, and `re.M` indicates that the pattern should match the meta-characters `^` and `$` to every line of the text, and not just the text as a whole. You may combine them by putting pipe-lines (`|`) between them.

25.1.2 Shorthand

You are allowed to skip the compile-step, and call the pattern search using a class call to the `re` module. Instead of calling methods of the pattern that the compilation produced, I can directly call the method for `re`, and use the regular expression as the first parameter. The code above then becomes:

```
import re

lAplus = re.findall( r"a+", "aardvark" )
print( lAplus )
```

If you run this code, you will notice that the output is exactly the same as for the first bit of code. The second approach still compiles the regular expression, but does not store the pattern. If a pattern is only needed a few times in a program, the second approach is fine. However, if it is used many times, the first approach is preferred, as in the first approach the compilation of the regular expression (which takes by far the most time of the whole process) is only done once, as opposed to every time.

25.1.3 Match objects

The `findall()` method used above returns the occurrences of the pattern in the target string. Often you need more information than just the actual patterns; for instance, you might want to know where the pattern occurs in the target string. The `re` module has methods that result in so-called "match objects," which are objects that contain, besides the textual result, more information, such as the index where the result is found in the target string. For example, the `search()` method returns a match object for the first occurrence of a pattern in a string.

```
import re

m = re.search( r"a+", "Look out for the aardvark!" )
print( "{} is found at index {}".format( m.group(), m.start() ) )
```

As you can see, the match object has several useful methods. These are:

- `group()` to return the found pattern
- `start()` to return the index at which the pattern starts
- `end()` to return the index where the pattern has ended

The `group()` method has some handy applications which you can control with parameters, which I will get to later.

The `match()` method is similar to the `search()` method, but checks if the pattern exists at the very start of a string. Both methods will return **`None`** if the pattern is not found, which as a condition is processed by Python as **`False`**.

```
import re

m = re.match( r"a+", "Look out for the aardvark!" )
if m:
    print( "{} starts the string".format( m.group() ) )
else:
    print( "The pattern is not found at the start of the string")
```

25.1.4 Lists of matches

I already showed that the `findall()` method creates a list of occurrences of a pattern in a string. The `finditer()` method is its complement, which creates a list (or rather, an iterator) of match objects for where the pattern occurs in a string. The best way to process such a list is by using the **`for`** `m` **`in`** `...` approach. For example:

listing2502.py

```
import re

mlist = re.finditer( r"a+",
    "Look out! A dangerous aardvark is on the loose!" )
for m in mlist:
    print( "{} starts at index {} and ends at index {}.".format(
        m.group(), m.start(), m.end() ) )
```

25.2 Writing regular expressions

Now the basics of using regular expression in Python via the `re` module have been explained, I can get into the actual writing of regular expressions.

25.2.1 Regular expressions with square brackets

The simplest regular expression is a string of characters, which describes a pattern consisting of exactly that string of characters. You may also describe a range of characters using square brackets `[` and `]`. For instance, the regular expression `[aeiou]` describes any of the characters `"a"`, `"e"`, `"i"`, `"o"`, or `"u"`. This means that if `[aeiou]` is part of a regular expression, at that location in the pattern one of these letters must reside (note: exactly one of them, so not multiple). For instance, to search for the words `"ball"`, `"bell"`, `"bill"`, `"boll"` and `"bull"`, the regular expression `b[aeiou]ll` can be used.

listing2503.py

```
import re

slist = re.findall( r"b[aeiou]ll", "Bill Gates and Uwe Boll \
drank Red Bull at a football match in Campbell." )
print( slist )
```

Exercise Change the regular expression above so that it not only finds the words "ball" and "bell", but also "Bill", "Boll", and "Bull".

You can use a dash within the square brackets between two characters to indicate that they represent not only these two characters, but also all the characters in between. For instance, the regular expression `[a-dqx-z]` is equivalent to `[abcdqxyz]`. To describe any of the letters of the alpabet, either as capital or lower case, you can use `[A-Za-z]`.

Moreover, if you place a caret (`^`) right next to the opening square bracket, that means that you want the opposite of what is within the square brackets. For instance, `[^0-9]` indicates any character except for a digit.

25.2.2 Special sequences

In a regular expression, just like in strings, the backslash character (`\`) indicates that the character that follows it has a special meaning, i.e., it is an escape sequence. The escape

sequences that hold for strings also hold for regular expressions, but regular expressions have many more. There are also a few meta-characters that are interpreted in a particular way. The following special sequences are defined (there are more, but these are the most common ones):

```
\b    Word boundary (zero-width)
\B    Not a word boundary (zero-width)
\d    Digit [0-9]
\D    Not a digit [^0-9]
\n    Newline
\r    Carriage return
\s    Whitespace (including tabulation)
\S    Not a whitespace
\t    Tabulation
\w    Alphanumeric character [A-Za-z0-9_]
\W    Not an alphanumeric character [^A-Za-z0-9_]
\/    Forward slash
\\    Backslash
\"    Double quote
\'    Single quote
^     Start of a string (zero-width)
$     End of a string (zero-width)
.     Any character
```

Note that "zero-width" means that the sequence does not represent a character, but a position in the string between two characters (or the start or end of the string). For instance, the regular expression `^A` represents a string that starts with the letter `"A"`.

Moverover, you can place characters or substrings between parentheses, in which case the characters are "grouped." Within a group, you can indicate a choice between multiple (sequences of) characters by placing pipe-lines (`|`) between them. For instance, the regular expression `(apple|banana|orange)` is the string `"apple"` or the string `"banana"` or the string `"orange"`.

You should be aware that some of these special sequences (in particular those without a backslash, the parentheses, and the pipe-line) do not work like indicated here when placed within square brackets. For instance, a period within square brackets does not mean "any character," but an actual period.

25.2.3 Repetition

Where regular patterns get really interesting is when repetitions are used. Several of the meta-characters are used to indicate that (part of) a regular expression is repeated multiple times. In particular, the following repetition operators are often used:

```
*      Zero or more times
+      One or more times
?      Zero or one time
{p,q}  At least p and at most q times
{p,}   At least p times
{p}    Exactly p times
```

You place such an operator after the (part of the) expression it repeats. For instance, `ab*c` means the letter `"a"`, followed by zero or more times the letter `"b"`, followed by the letter `"c"`. Thus, it matches the strings `"ac"`, `"abc"`, `"abbc"`, `"abbbc"`, `"abbbbc"`, etcetera.

A repetition operator after a group (between parentheses) indicates the repetition of the whole group. For instance, `(ab)*c` matches the strings `"c"`, `"abc"`, `"ababc"`, `"abababc"`, `"abababababc"`, etcetera.

Regular expression matching for repetitions is greedy. It will always try to match the earliest occurring pattern first, extended to its longest possible extension. For example:

listing2504.py

```
import re

mlist = re.finditer(r"ba+","A sheep says 'baaaaah' to Ali Baba.")
for m in mlist:
    print( "{} is found at {}.".format(m.group(), m.start()))
```

Exercise Change the regular expression in the code above so that it finds any `"b"` followed by one or more `"a"`s, where the `"b"` might be captitalized. The output should be `"baaaaa"`, `"Ba"` and `"ba"`.

Exercise Once you have solved the previous exercise, change the regular expression so that it finds the pattern consisting of a `"b"` or `"B"` followed by a sequence of one or more `"a"`s, repeated one or more times. The output should be `"baaaaa"` and `"Baba"`. You will need to use parentheses for this. When you think that your regular expression is correct, also test it on several other strings.

Here is another one, which searches for occurrences of one or more `"a"`s:

listing2505.py

```
import re

mlist = re.finditer(r"a+","A sheep says 'baaaaah' to Ali Baba.")
for m in mlist:
    print( "{} is found at {}.".format(m.group(), m.start()))
```

When you run this code, you see that it finds four occurrences of the pattern: three times a single `"a"`, and one time a sequence of five `"a"`s. You might wonder why the pattern matching process does not also find the four `"a"`s starting at position 16, the three `"a"`s starting at position 17, the two `"a"`s starting at position 18, and the single `"a"` starting at position 19. The reason is that the `finditer()` and `findall()` methods, when they find a match, continue searching immediately after the end of the last found match. Normally, this is the behavior that you want.

Exercise Now change the `r"a+"` in the code above to `r"a*"`, which changes it to searching for zero or more `"a"`s. Before running the code, think about what you expect the outcome to be. Then run the code and see if your prediction was correct. If it wasn't, do you now realize why the outcome is what it is?

You may have noticed that regular expressions may become overly complex fast. It is a good idea to comment them so that you can understand them on later examination.

25.2.4 Practice

With all you learned until now, you should be able to do the following exercise. It is wise to solve this one before continuing with the remainder of this chapter. The exercise consists of a piece of code that you have to complete.

Exercise When you run the code below, it tries to search for all the regular expressions in `relist`, in all the strings in `slist`. It prints for each string the numbers of all the regular expressions for which matches are found. Your goal is to fill in the regular expressions in `relist` according to the specification in the comments to the right of each expression. Note that the first seven regular expressions need to cover the string as a whole, so you should have them start with a caret and end with a dollar sign, which indicates that the expression should match the string from the start to the end.

listing2506.py

```
import re

# List of strings used for testing.
slist = [ "aaabbb", "aaaaaa", "abbaba", "aaa", "gErbil ottEr",
    "tango samba rumba", " hello world ", " Hello World " ]

# List of regular expressions to be completed by the student.
relist = [
    r"",  # 1. Only a's followed by only b's, including ""
    r"",  # 2. Only a's, including ""
    r"",  # 3. Only a's and b's, in any order, including ""
    r"",  # 4. Exactly three a's
    r"",  # 5. Neither a's nor b's, but "" allowed
    r"",  # 6. An even number of a's (and nothing else)
    r"",  # 7. Exactly two words, regardless of white spaces
    r"",  # 8. Contains a word that ends in "ba"
    r""   # 9. Contains a word that starts with a capital
]

for s in slist:
    print( s, ':', sep='', end=' ' )
    for i in range( len( relist ) ):
        m = re.search( relist[i], s )
        if m:
            print( i+1, end=' ' )
    print()
```

The correct output is:

```
aaabbb: 1 3
aaaaaa: 1 2 3 6
```

```
abbaba: 3 8
aaa: 1 2 3 4
gErbil ottEr: 7
tango samba rumba: 8
 hello world : 5 7
 Hello World : 5 7 9
```

Make sure that you can do all of these correctly before you continue!

25.3 Grouping

As shown above, when parentheses are used in regular expressions, they create so-called "groups." Take for instance the regular expression `(\d{1,2})-(\d{1,2})-(\d{4})`, which describes a sequence that could represent a date: one or two digits, followed by a dash, followed by one or two digits, followed by a dash, followed by four digits (if you do not understand this regular expression, check back in previous sections of this chapter until you do understand it). This expression contains three groups: the first containing one or two digits, the second containing one or two digits, and the third one containing the four digits at the end. The code below searches for this pattern in a string.

listing2507.py

```
import re

pDate = re.compile( r"(\d{1,2})-(\d{1,2})-(\d{4})" )
m = pDate.search( "In response to your letter of 25-3-2015, \
I decided to hire a hitman to get you." )
if m:
    print( "Date {}; day {}; month {}; year {}".format(
            m.group(0), m.group(1), m.group(2), m.group(3) ) )
```

When you run the code, you see that it not only gets out the result as a whole (using the method `group()` or `group(0)`), but that you can also access each of the groups that is found in the result, using methods `group(1)` for the day, `group(2)` for the month, and `group(3)` for the year. You can also use the method `groups()` to get a tuple with all the groups.

25.3.1 `findall()` and groups

The `findall()` methods returns a list of pattern objects. In the examples where it was used until now, it returned a list of strings. Indeed, pattern objects are strings if there is at most one group in the regular expression. If there are multiple groups, pattern objects are actually tuples that contain all the groups.

listing2508.py

```
import re

pDate = re.compile( r"(\d{1,2})-(\d{1,2})-(\d{4})" )
datelist = pDate.findall( "In response to your letter of \
25-3-2015, on 27-3-2015 I decided to hire a hitman to get you." )
```

```
for date in datelist:
    print( date )
```

25.3.2 Named groups

It is possible to give each group a name, by placing the construct `?P<name>` (where you replace "name" with the name you want the group to have – you leave the < and > in the expression in this case) immediately after the opening parenthesis. You can then refer to the groups by these names, instead of their index.

listing2509.py

```
import re

pDate = re.compile(
    r"(?P<day>\d{1,2})-(?P<month>\d{1,2})-(?P<year>\d{4})")
m = pDate.search( "In response to your letter of 25-3-2015, \
I curse you." )
if m:
    print( "day is {}".format( m.group('day') ) )
    print( "month is {}".format( m.group('month') ) )
    print( "year is {}".format( m.group('year') ) )
```

25.3.3 Referring within a regular expression

Suppose that you have to create a regular expression that represents a string that contains an arbitrary non-space character twice. For instance, the string "regular" would not have a match, but the string "expression" would (as it contains two `"e"`s and two `"s"`s). This cannot be done with the regular expression features that we discussed until now. It can be solved, however, with groups, and special references within a regular expression, namely as follows: using the special sequence `\x`, whereby `x` is a number, you refer to the group with index `x` in the match. Thus, a regular expression that represents a string with an arbitrary non-space character twice is `(\S).*\1`.

Since at this point this regular expression might still be a bit hard to understand, let's look at it in depth. The `\S` is a special sequence that represents a non-space character. Putting it in parentheses turns it into a group, and since this is the first (and only) group in the expression, its index is 1. The `.*` represents a sequence of zero or more characters, which can be anything (the period is a meta-character that represents any character). Finally, the `\1` refers to the first group, and says that here you want to have exactly the same thing as the first group represents. If you are wondering why you do not need to represent anything that can be placed before the `\S`, or anything that can come after the `\1`, then the answer is that you are not specifying that this regular expression represents a string as a whole, so as long as it occurs anywhere in the string, it matches.

Test this pattern with the code below, by replacing the string `"Monty Python's Flying Circus"` with different strings, and running the code to examine the results.

listing2510.py

```
import re

m = re.search( r"(\S).*\1", "Monty Python's Flying Circus" )
if m:
    print( "The character {} occurs twice".format( m.group(1) ) )
else:
    print( "No match was found." )
```

Exercise Can you change the regular expression in the code above so that it checks if the string contains a character at least three times?

Exercise Can you change the regular expression so that it checks whether it contains at least two characters twice? This is quite hard and therefore optional, but if you try to do it, make sure that you test it with at least the strings `"aaaa"`, `"aabb"`, `"abab"` and `"abba"`. These all should match, unless you also want the two repeated characters different, then `"aaaa"` should not match (but note that that makes the regular expression even harder to design).

25.4 Replacing

While regular expressions are mainly used for searching, you can also use regular expressions to replace substrings in a string with different substrings. You can use the `sub()` method for this. `sub()` gets as arguments the to-be-replaced pattern, the replacement, and the string. The `sub()` method returns the new string (remember that strings are immutable, so `sub()` will not actually change your original string, even if it is stored in a variable; you will have to store its return value if you want access to the new string).

The replacement is usually just a string, but it may contain references to groups in the original pattern. You will have to use a format that is different from the `\x` format shown before. If you want to refer to group `x` in the pattern (`x` being a number), you write `\g<x>`. The reason for the difference is disambiguation; it allows you to distinguish a reference to, for instance, group 2 followed by a character zero, from a reference to group 20.

```
import re

s = re.sub( r"([iy])se", "\g<1>ze", "Whether you categorise, \
emphasise, or analyse, you should use American spelling!" )
print( s )
```

What you learned

In this chapter you learned about:

- What regular expressions are
- Which meta-characters can be used in regular expressions

- How to use regular expressions in Python, using the `re` module
- Compiling regular expressions with `re.compile()`
- What match objects are
- Searching for patterns using `match()`, `search()`, `findall()`, and `finditer()`
- Replacing patterns using the `sub()` method
- Using square brackets in regular expressions to represent different possibilities for characters
- Using special sequences in regular expressions, many of which use the backslash character
- Repeating sub-patterns in regular expressions using repetition operators
- Grouping of sub-patterns using parentheses
- Using groups to unravel results
- Referencing within patterns
- Smugly referring people with pattern mining problems to regular expressions

Exercises

Exercise 25.1 Assume that a word consist of only letters from the alphabet (upper case or lower case). Write some code that uses a regular expression to make a list of all the words in a text.

Exercise 25.2 Using a regular expression and the `findall()` method, create a list of all the occurrences of the word "the" a sentence. Print the number of items in the list. Your code should handle the problem in a case-insensitive way. Make sure that you do not count the letter combination "the" when it occurs as part of another word (e.g., "thesaurus" or "ether").

Exercise 25.3 A person's full name consists of two words, next to each other, consisting of only letters from the alphabet, all lower case except for the first one, which is upper case. Between the two words there should only be white spaces. The words start and end at a word boundary. E.g., according to this specification Cardinal Richelieu is a name, but Charles d'Artagnan is not, and neither is Gilbert duPrez, Joe DiMaggio, or Unit X1138. Under this assumption, use a regular expression to list all the two-word combinations in a sentence which are probably names of persons.

Exercise 25.4 As a follow-up to the previous exercise, now assume that a person's name consists of two or more words that meet the criteria spelled out above. Use a regular expression to extract all names.

Exercise 25.5 When a person speaks in a piece of text, this is often represented by enclosing the spoken part within double quotes ("). Write a regular expression that extract all double-quote enclosed parts from a text. Hint: Use groups, and remember that regular expressions are greedy.

Exercise 25.6 When scraping data from HTML pages, you can often find the items you are interested in by looking for mark-ups. Suppose that we have a page with data of persons, who have an ID and a name. The ID is a nine-digit number, and has a marker `<id>` in front of it, and a marker `</id>` after it. The name belonging to the ID will follow immediately after the ID, and has a marker `<name>` in front of it, and a marker `</name>` after it. Use a regular expression to extract all the IDs and corresponding names from the text below, and print them. There should be five of them.

exercise2506.py

```
import re

text = "<html><head><title>List of persons with ids</title>\
</head><body>\
<p><id>123123123</id><name>Groucho Marx</name>\
<p><id>123123124</id><name>Harpo Marx</name>\
<p><id>123123125</id><name>Chico Marx</name>\
<randomcrap>Etaoin<id>Shrdlu</id>qwerty</name></randomcrap>\
<nocrap><p><id>123123126</id><name>Zeppo Marx</name></nocrap>\
<address>Chicago</address>\
<morerandomcrap><id>999999999</id>nonametobeseen!\
</morerandomcrap>\
<p><id>123123127</id><name>Gummo Marx</name>\
<note>Look him up on <a href=\"http://www.google.com\">\
Google.</a></note>\
</body></html>"
```

Chapter 26

File Formats

Data is usually stored in files, which are constructed according to a specific file format. Standardized file formats often are supported by Python using a particular module. In this chapter, I discuss some of more common file formats and the modules that support them.

26.1 Comma-Separated Values (CSV)

Comma-Separated Values (CSV) is the most common text file format that is used for importing and exporting data to and from spreadsheets and databases. The general format says that each line contains one record (a record is a complete entity), listing each of the fields of the record in a specific order, separating the fields by commas. The first line of the file may or may not consist of names for the fields in the CSV file.

The code below loads and displays the contents of a typical CSV file. Appendix E explains how to get this CSV file.

```
fp = open( "pc_inventory.csv" )
print( fp.read().strip() )
fp.close()
```

Unfortunately, the CSV format is not standardized, and different software packages tend to use slightly different implementations of CSV files. However, over the years the different conventions used by the major software packages have converged to something of a standard, that is implemented in the Python `csv` module. The module supports "dialects" of CSV formats to handle files from different sources.

Instead of using the `csv` module, if you have to deal with an excentric CSV format that the module does not support, you can try to design your own interpretation of lines of the file using regular expressions. You can also try to design your own dialect. Neither option is very appealing.

26.1.1 CSV `reader()`

The `csv` module contains a `reader()` function that provides access to a CSV file. The `reader()` function gets a file handle as argument, and returns an iterator that allows you

to get the lines from the file, as a list with each of the fields as an element of the list. You should leave the file open while accessing it with `reader()`.

listing2601.py

```
from csv import reader

fp = open( "pc_inventory.csv", newline='' )
csvreader = reader( fp )
for line in csvreader:
    print( line )
fp.close()
```

The Python documentation recommends that if you use `reader()` on a file (and that is what you usually do), you specify a `newline=''` argument as extra argument when opening the file (I did this in the code above). This is necessary in case some of the text fields in the CSV file contain newline characters.

`reader()` takes extra arguments too. Common ones are `delimiter=<character>`, which indicates the `<character>` which is placed between different fields (default is the comma), and `quotechar=<character>`, which indicates which `<character>` is used to enclose strings with (default is the double quote).

26.1.2 CSV `writer()`

Writing a CSV file is just a little bit harder than reading one. You create a file handle to a file that you open for writing (`"w"` mode), and use it as an argument when you call the `writer()` function from the `csv` module. The object that is returned from the `writer()` call has a method `writerow()` that you can call with a list of fields, that it then writes to the output file in CSV format.

The call to `writer()` can get the same arguments as the call to `reader()` can get, including specifying a `delimiter` and a `quotechar`. You can also supply a `quoting=<quotemethod>` argument, that supports the following methods of quoting:

- `csv.QUOTE_ALL`, which encloses every field in quotation characters
- `csv.QUOTE_MINIMAL`, which only encloses fields in quotation characters if it is absolutely necessary (this is the default)
- `csv.QUOTE_NONNUMERIC`, which encloses fields in quotation characters if they are not integers or floats
- `csv.QUOTE_NONE`, which encloses no fields in quotation characters

Enclosing a string within quotation characters is generally only needed if the string contains exceptional characters, such as newlines or the character that is used as delimiter.

listing2602.py

```
from csv import writer

fp = open( "pc_writetest.csv", "w", newline='' )
```

```
csvwriter = writer( fp )
csvwriter.writerow( ["MOVIE", "RATING"] )
csvwriter.writerow( ["Monty Python and the Holy Grail", 8] )
csvwriter.writerow( ["Monty Python's Life of Brian", 8.5] )
csvwriter.writerow( ["Monty Python's Meaning of Life", 7] )
fp.close()
```

Exercise After using the code above to create the file "pc_writetest.csv," open it and use `reader()` to list its contents.

26.2 Pickling

Suppose that you want to store a certain data structure in a file, for instance, a list of tuples. One way of doing that is to turn the tuples into strings and write those into the file, one line for every tuple. When you then later want to rebuild the data structure in a program, you read the file, unravel the lines, and reconstruct the list of tuples. As you can imagine, this encompasses a considerable amount of quite difficult code.

Fortunately, you do not have to write such code. Python offers a solution for storing data structures in files, including both structure and content, which is called "pickling." You can write the whole data structure to the file in one go, if you just open a binary file for writing, and call the function `dump()` from the `pickle` module with the data structure as first argument, and the file handle as second argument.

listing2603.py

```
from pickle import dump

cheeseshop = [ ("Roquefort", 12, 15.23),
    ("White Stilton", 25, 19.02), ("Cheddar", 5, 0.67) ]

fp = open( "pc_cheese.pck", "wb" )
dump( cheeseshop, fp )
fp.close()

print( "Cheeseshop was pickled" )
```

To read the contents of a pickle file, you use the function `load()` from the `pickle` module. `load()` gets a handle to the file as argument. Do not forget to open the file in binary mode.

listing2604.py

```
from pickle import load

fp = open( "pc_cheese.pck", "rb" )
buffer = load( fp )
fp.close()

print( type( buffer ) )
print( buffer )
```

As you can see, `load()` restores the data structure completely.

Pickling works even for your own classes:

listing2605.py

```
from pickle import dump, load

class Point:
    def __init__( self, x, y ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({},{})".format( self.x, self.y )

p = Point( 2, 5 )
fp = open( "pc_point.pck", "wb" )
dump( p, fp )
fp.close()

fp = open( "pc_point.pck", "rb" )
q = load( fp )
fp.close()

print( type( q ) )
print( q )
```

26.3 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is a file format that is often used in modern applications, in particular those that communicate via web services. It is supported by many languages (JavaScript amongst them, of course). It is similar to pickling in the sense that it stores in-memory objects to files, retaining their structure. A difference with pickling is that JSON files are in human-readable format.

The `json` module works equivalent to the `pickle` module, with a `dump()` function that writes data structures to a file, and a `load()` function to load data structures from a file. The file must be a text file, and not a binary file.

listing2606.py

```
from json import dump, load

cheeseshop = [ ("Roquefort", 12, 15.23),
    ("White Stilton", 25, 19.02), ("Cheddar", 5, 0.67) ]

fp = open( "pc_cheese.json", "w" )
dump( cheeseshop, fp )
fp.close()
```

```
fp = open( "pc_cheese.json", "r" )
buffer = load( fp )
fp.close()

print( type( buffer ) )
print( buffer )
```

Alternatives for `dump()` and `load()` are the functions `dumps()` and `loads()`, which do not get a file argument. Instead, `dumps()` gets no file argument at all, and just produces a string that contains the data structure in JSON format, while `loads()` gets a string as argument instead of a file, and loads the data structure from that string.

These functions can get many optional arguments that determine how exactly the data will be stored; for instance, you can set the `indent=` argument for `dump()` and `dumps()` to determine which indentation value will be used, and you can use arguments to sort the data in the dump. If you want to know more about this, consult the references.

A weakness of the `json` module is that it only supports the standard Python data structures. If you want to use it to store instances of classes of your own making, you have to find a way to convert your own classes to standard Python structures. The `json` module offers special `JSONencoder` and `JSONdecoder` classes to help you with that. It goes too far to discuss these here.

26.4 HTML and XML

HTML and XML are standard formats that are used to display information on webpages. They consist of readable text files, with many instructions on formatting. It is a common task for data miners to "scrape" data from webpages. You can use regular expressions for that, but if the webpages are reasonably well-formatted, the "Beautiful Soup" module may help you out.

The Beautiful Soup module is named `bs4` in Python (naturally, `bs3` came before it, and it may get more updates later). It contains the `BeautifulSoup` class that you can use to load and interpret HTML and XML files. `bs4` is not part of the standard Python package; you have to install it separately, which is quite a hassle, unless you use a tool called `pip` which comes standard with Python 3.

There are alternative modules that can ease the pain of web scraping for you, notably `lxml`, but Beautiful Soup seems to be the most popular.

Since all such modules require separate installations, I will not discuss them here. I only wish to indicate that if you need to do web scraping (and it is likely you have to do that at some point), you should check out some of the standard tools available for that before you delve into eccentric regular expression-design.

What you learned

In this chapter, you learned about:

- Reading and writing CSV files using the `csv` module
- Pickling using the `pickle` module
- Reading and writing JSON files using the `json` module
- The availability of tools for web scraping

Exercises

Exercise 26.1 Open the file "pc_inventory.csv" and read its contents using `reader()`. Write the contents to a different CSV file, using a space as delimiter and a single quote as quotation character. Open the file you created as text file and display its contents to check them.

Exercise 26.2 Load the contents from the file "pc_inventory.csv," and put them in a list of lists (each line in the file being one list in the list of lists). Store the list in JSON format. Open the file you created as a text file and display its contents to examine them.

Chapter 27

Various Useful Modules

At this point in the book, everything that you really need to know to be an accomplished Python programmer – or maybe even an accomplished programmer in general – has been covered. I want to quickly highlight a few useful modules that do not need a chapter of their own. I will not give many details; once you know what the purpose of a module is, you can look up more information on it in the Python reference.

27.1 `datetime`

The `datetime` module contains functions that allow the manipulation of date and time. The module contains various classes for date and time manipulation, of which the most important ones are `datetime`, `timedelta`, `date`, and `time`. `datetime` contains attributes `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, and `tzinfo` (the last attribute provides time zone information). `date` and `time` contain subsets of these attributes. Objects of these types are immutable.

I restrict myself to discussing the `datetime` and `timedelta` classes, though related functions and methods exist for the other classes.

`datetime` objects hold a date and a time. Amongst the methods for `datetime` objects are:

- `now()` creates a `datetime` object that contains the current day and time. You would typically use a class call to get a value for `now()`.
- `datetime()` creates a `datetime` object using given arguments. The first three arguments are not optional, and are year, month, and day. The others, hour, minute, second, microsecond, and tzinfo are optional. Arguments can either be given in this order, or by specifying `<argument>=<value>`, with `<argument>` an argument name as specified above.

```
from datetime import datetime

print( datetime.now() )
```

When printing `datetime` objects you get a specific format as output. If you want a different format (including printing such things as the day of the week) then the `datetime` module has functions that allow you to specify different kinds of formatting. For more information, see the Python reference.

To calculate with `datetime` objects, you need `timedelta`. A `timedelta` object specifies a difference between two `datetime` objects. A `timedelta` object stores `days`, `seconds`, and `microseconds`. You can create `timedelta` objects with other period-representing arguments, but it only stores the three mentioned here; other arguments are recalculated into these three.

You can perform all kinds of calculations with `timedelta` objects, but the most useful ones are concerning the difference between `datetime` objects. So you can add a `timedelta` object to a `datetime` object to get a new `datetime` object, or subtract two `datetime` objects from each other to get their difference as a `timedelta` object.

listing2701.py

```
from datetime import datetime, timedelta

thisyear = datetime.now().year
xmasthisyear = datetime( thisyear, 12, 25, 23, 59, 59 )
thisday = datetime.now()
days = xmasthisyear - thisday

if days.days < 0:
    print( "Christmas will come again next year." )
elif days.days == 0:
    print( "It's Christmas!" )
else:
    print( "Only", days.days, "days to Christmas!" )
```

27.2 `collections`

The `collections` module contains handy classes that allow you to manipulate iterables such as strings, tuples, lists, dictionaries, and sets. `collections` offers many interesting functionalities, most of which are a bit eccentric, making it unlikely that you will need to use them soon. I discuss two of them, namely the `Counter` class and the `deque` class.

A `Counter` object is a similar to a dictionary, which contains items as keys, and for each of the items a "count" as value. You create a `Counter` object by providing the sequence of which you want to count the items as argument. It has some useful methods, such as:

- `most_common()` gets an integer as argument and returns a list containing the items that have the highest count, as many as the integer argument indicates. The items on the list are 2-tuples, the first element of a tuple being the counted item, and the second element being the count. They are ordered from most common to least common. If no integer argument is specified, the list contains all the items.
- `update()` gets an iterable as argument and "adds in" the items of the iterable.

listing2702.py

```
from collections import Counter

data = [ "apple", "banana", "apple", "banana", "apple", "cherry" ]
c = Counter( data )
print( c )
print( c.most_common( 1 ) )

data2 = [ "orange", "cherry", "cherry", "cherry", "cherry" ]
c.update( data2 )
print( c )
print( c.most_common() )
```

A deque object is a list that is supposed to be used as a "queue," i.e., a list for which items are added and removed from either end of the list. It supports methods that are similar to list methods, such as append(), extend(), and pop(), which work at the right side end of the list, but also has similar methods that work at the left side end of the list, such as appendleft(), extendleft(), and popleft(). For the rest, it has the same methods that you expect a list to have. You create a deque object with the iterable which you want to turn into a deque as argument.

```
from collections import deque

dq = deque( [ 1, 2, 3 ] )
dq.appendleft( 4 )
dq.extendleft( [ 5, 6 ] )
print( dq )
```

27.3 urllib

The urllib module allows you to access web pages in the same way that you access files. There are two modules of main interest: urllib.request contains functions to access Internet content, and urllib.error contains definition for exceptions that can be raised. You can also use urllib to communicate with webpages; if you want to do so, you need to study the urllib.parse module. For now, I only give a simple example in which you want to open a webpage and read its contents.

listing2703.py

```
from urllib.request import urlopen
from urllib.error import HTTPError, URLError
from sys import exit

try:
    u = urlopen( "http://www.python.org" )
except HTTPError as e:
    print( "HTTP Error", e )
    sys.exit()
```

```
except URLError as e:
    print( "URL error", e )
    sys.exit()

for i in range( 5 ):
    text = u.readline()
    print( text )

u.close()
```

Note that from `urllib` only `urlopen` needs to be imported. Once you have opened a web address, it is returned as a file handle, so you can use the regular file methods on it.

27.4 glob

The `glob` module provides a function `glob()` to produce lists of files based on a wildcard search pattern that is provided as argument. The wildcard search uses Unix conventions, most of which also hold on other systems. They are as follows:

- A question mark (`?`) in a file name indicates any character
- A Kleene star (`*`) in a file name indicates any sequence of characters
- A sequence of characters between square brackets (`[]`) indicates any of these characters; a dash may be used to indicate a sequence that runs from the character to the left of the dash to the character to the right of the dash

For instance, the wildcard search `"A[0-9]?B.*"` looks for all files that start with the letter A, followed by a digit, followed by any character, followed by a B, with any extension. It depends on the operating system whether this is a case-sensitive or case-insensitive search.

Do not confuse a wildcard search pattern with a regular expression. While they have some superficial resemblance (such as an asterisk indicating "a series of any characters" in both of them), they are nothing alike. Wildcard searches only support the patterns listed above (some of which have a different meaning for regular expressions), and are only used for `glob` and when directly communicating with the system via the command prompt.

```
from glob import glob

glist = glob( "*.pdf" )
for name in glist:
    print( name )
```

The `glob` module also contains a function `iglob()`, which has the same functionality as `glob()`, but produces an iterator instead of a list.

Exercise Use `glob()` to list all Python files in the current directory.

27.5 statistics

The statistics module gives you access to various common statistical functions. All of these functions get as argument a sequence or iterator of numbers (integers or floats).

- `mean()` calculates the mean (or average) of a sequence of numbers
- `median()` calculates the median of a sequence of numbers, i.e., the "middle" number
- `mode()` calculates the mode of a sequence of numbers, i.e., the number that occurs most often
- `stdev()` calculates the standard deviation of a sequence of numbers
- `variance()` calculates the variance of a sequence of numbers

There are a few more functions in the `statistics` module, but these are the most-used ones. For more advanced statistical calculations, there are other modules available, which I do not discuss in this book.

These functions may raise a `StatisticsError`. This is particularly relevant in the case of the `mode()` function, as it is generated when no unique mode can be found.

listing2704.py

```
from statistics import mean, median, mode, stdev, variance, \
    StatisticsError

data = [ 4, 5, 1, 1, 2, 2, 2, 3, 3, 3 ]

print( "mean:", mean( data ) )
print( "median:", median( data ) )
try:
    print( "mode:", mode( data ) )
except StatisticsError as e:
    print( e )
print( "st.dev.: {:.3f}".format( stdev( data ) ) )
print( "variance: {:.3f}".format( variance( data ) ) )
```

Note that for a sequence with an even number of numbers, the median is the average of the two "middle" numbers. There are different ways of calculating the median in case of an even number of numbers; if you want to use a different one, examine other functions in the `statistics` module.

As for the mode, in the literature you find multiple definitions of what the mode is supposed to be. The general definition is "the most common number," but what if there are multiple of those? What if each number is unique? The version of the mode that the `statistics` module supports does not seem to be the most common one.

What you learned

In this chapter, you learned about:

- `datetime` module
- `collections` module
- `urllib` module
- `glob` module
- `statistics` module

Exercises

Exercise 27.1 Use the `Counter` class to list the five most common letters in a text, with their counts.

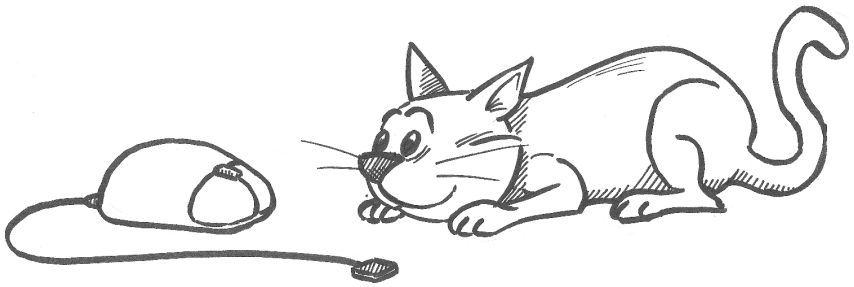
Exercise 27.2 Create a program that asks the user for numbers, until the user enters zero. It then prints the mean, median, and mode of these numbers. The `statistics` module can be used for the mean and median; however, for the mode, print all those numbers that have the highest count, even if that entails that you print more than one number. By definition, for a number to be the mode it must occur at least twice; so if every number only occurs once, there is no mode. Hint: Consider using the `Counter` class to construct the mode.

Epilogue

If you made it through the book and managed to do most of the exercises all by yourself: Congratulations! You have become a programmer. You have internalized the basics of dealing with any programming language, and should be able to solve most of the programming problems that you encounter in your career as a student or as a professional worker. If you need to learn another programming language, by now you have a solid basis to quickly get to grips with any other language. Even better, you have learned to approach problems like programmers do, which is a valuable skill that you will find many uses for.

Future editions of this book should have even more information and more interesting problems to work on. This will all be optional material, but if you enjoy programming as much as I do, you might want to check it out.

If you have any remarks on the contents of the present version of the book, I'll be glad to receive your message at `pythonbook@spronck.net`.



Appendix A

Troubleshooting

When I run code I get an ImportError

Check the name of the file that your code tries to import. If it is supposed to be one of the standard Python libraries, you probably made a spelling mistake in the name. Either that, or you added `.py` to the name – you should not do that. If the error concerns `pcinput` or `pcmaze`, then probably you either did not build these files (check Appendix C or D to solve the problem), or you placed them in a location that Python has no access to. In the last case, make sure you copy them to the same place as where you keep your Python programs.

I get a FileNotFoundError: [Errno 2]

You try to open a file that Python cannot find. You might have forgotten to include the complete pathname in the file name, or you think the file is located in the current directory while actually it is not. Or maybe your code tries to use one of the standard test files that I use for the book, and you do not have those yet. If that is the case, check Appendix E to learn how to make them.

I get a SyntaxError but I have no idea what I did wrong

When one or more syntax errors are reported, you should try to solve the one reported first. Further errors are often caused by the first one. The line of code where Python discovered the error is reported with the error. Check that line. Also check the line above it: it might very well happen that you made an error in a line of code, but Python only notices it when it processes the next line. Syntax highlighting may also give an indication of where you made your mistake. Common syntax errors that beginning programmers make are forgetting to include a colon (`:`) at the end of an **`if`**, **`while`**, or **`for`** statement, misspelling the name of variables that they created, or making errors in indentation.

I get a SyntaxError that reports a "Non-UTF-8 code"

You have used a character in your program that Python cannot process. For instance, you might have placed your own name in the code (maybe even only in a comment), and your

name is spelled with a special character that is not directly found on the keyboard. Stick to characters that are found on the US keyboard. It is not that you cannot include special characters in your code, but the rules for doing that are explained in later chapters in this book.

I get a curious error even when running the example code

Make sure that you use Python version 3.4 or later. The code was written with Python 3.4, and I have noticed that some code elements do not work correctly with earlier versions of Python.

When I run my program it doesn't do anything

Maybe you built an endless loop in your program, so the program is actually working but never getting to the point where it produces output. Check your loops. Sometimes it is helpful to include a `print()` statement at the start of your program, just to see that the program actually has started. `print()` statements in the code might also help to discover where it gets stuck.

I defined a function (or class) in my program, but it seems Python cannot access it

Make sure that you spelled your call to the function (or class) correctly. Remember that Python is case sensitive! If the spelling seems to be correct, make sure that you have not created a variable with the same name as the function (or class). If you did, this will definitely destroy the possibilities of Python to access your function (or class).

I have been staring at my program for hours and can't get it to work

Sometimes it is good to take a break. Put the program away, go home, play some games, exercise, take a shower, anything. Get back to it tomorrow. Ask any programmer: sometimes you get stuck and cannot resolve a problem, for which the solution is immediately clear to you when you come in the next day. What might also help is get someone else in and explain your problem to them. Often, during such an explanation, you suddenly see what you were doing wrong all this time. What you definitely should *not* do, however, is continue writing your program without resolving the problem. You will only create a bigger mess. A much better idea would be to make a copy of your program and then actually remove/simplify code until the program starts working. That at least gives you an indication where you have to look for the error.

Appendix B

Differences with Python 2

This appendix contains an overview of the differences between Python 2 and Python 3, as far as they relate to the contents of this book and as far as I am aware of them. You can ignore this appendix if you are only going to use Python 3.

Operators

The division operator (`/`) in Python 2 works differently from the one in Python 3. In Python 3, it is automatically assumed that if you use division, you need floating-point numbers, so the division will assume that all numbers involved are floats, and will always result in a float. In Python 2, it is assumed that the division will be of the type that is "most detailed" for the numbers involved, i.e., if you divide two numbers and at least one is a float, it will be a floating-point division and the result is a float, but if you divide two integers, it will be an integer division and the result will be an integer (if the resulting number would mathematically have decimals, then the decimals are simply removed). The way Python 2 works is similar to what most programming languages do, but the way Python 3 works is more intuitive and leads to fewer errors.

Reserved words

In Python 3, **`print`** and **`exec`** are no longer reserved words (or keywords), as they are now functions. However, **`True`**, **`False`**, and **`None`** are now keywords, as is the word **`nonlocal`**.

Basic functions

A small difference between Python 2 and Python 3 is that when using the **`type`**`()` function, where Python 3 displays the word `class`, Python 2 displays the word `type`. The reason is that in Python 3, all types are actually implemented as classes.

The **`format`**`()` function was implemented in later versions of Python 2, but did not exist in the earlier versions. Instead, it supported a style of formatting, using "percentage-codes," similar to what is used in languages such as C++. This was directly implemented in the

print() function, and remained part of the **print**() function even after the introduction of **format**() (for compatibility reasons). Consequently, most Python 2 programs, even the ones written using the most recent version of Python 2, use this older-style formatting. These older approaches are no longer supported in Python 3. By the way, the parentheses after the **print**() function are not necessary in Python 2, but obligatory in Python 3.

In Python 3, the **range**() function is an iterator (see Chapter 23). This entails that it needs very little memory space: it only retains the last number generated, the step size, and the limit that it should reach. In Python 2 **range**() is implemented differently: it produces all the numbers of the range in memory at once. A statement like **range**(1000000000) in Python 2 requires so much memory that it may very well crash the program. In Python 3, such issues do not exist. In Python 2 it is therefore recommended not to use **range**() for more than 10,000 numbers or so, while in Python 3 no restrictions exist.

In Python 2, the string manipulation methods were part of the `string` module, and were generally called by importing the module and then writing `string.<method>()`. Such calls are no longer supported by Python 3.

Data structures

In Python 2, sorting a mixed list does not give a runtime error. The `sort()` function also supports an argument `cmp=<function>`, that allows you to specify a function that does the comparison between two elements. This function is no longer supported in Python 3, but you can use the key parameter for the same purpose. In the Python module `functools` a function `cmp_to_key()` is found that turns the old-style `cmp` specification into the new-style `key` specification.

In Python 2, the dictionary methods `keys()`, `values()`, and `items()` return lists instead of iterators. Python 2 also supports a method `has_key()` that you can use to check if a certain key is in a dictionary, but this method has been removed from Python 3.

Python 2 does not support sets natively. You have to import the `sets` module to use them. Moreover, to create a set you use the `Set()` method, not the **set**() function. To create a set with elements in them, in Python 2 the only way is to give the elements as a list argument to the `Set()` method.

The structure of exceptions has changed a bit between Python 2 and Python 3. In particular, the exception `IOError` has been made an alias for the `OSError` exception, as it was quite hard to distinguish which each of them tried to capture.

Unicode

Python 2 was not yet supporting Unicode natively, while Python 3 does. Python 3 strings are Unicode strings. You will not notice the difference as long as strings are ASCII, but Python 2 strings might get into troubles when Unicode characters are inserted into the strings. Python 3 also supports Unicode characters in identifiers such as variable names, while Python 2 does not. It is not recommended that you do use non-ASCII characters in identifiers, though.

Python 2 does not have byte strings. There is no need for them in Python 2, as it does not support Unicode. In Python 2, if you want to write a byte with ordinal value zero, you

just write **chr**(0). The read() and write() methods for binary files use regular strings in Python 2. This is not allowed in Python 3.

In Python 2, the pickle module worked on text files. This is no longer possible in Python 3, as Python 3 supports Unicode. Pickle files written with Python 2 cannot be loaded in Python 3 and vice versa.

Iterators

Python 3 is much more based on iterators and generators than Python 2, which has all kinds of advantages, mainly as far as speed and memory usage is concerned. Consequently, there are quite a few differences between Python 2 and 3 in this respect. I have not inventoried all of them, but here are a few that struck me:

Iterators in Python 2 have a **next**() method. They no longer have it in Python 3, where it is called __next__().

In Python 2 **zip**() produces a list rather than an iterable.

The itertools module has some differences too. For instance, in Python 2 it has a function izip() that produces an iterable, but since in Python 3 **zip**() does that already, the function izip() has been removed from itertools.

Modules

Besides pickle and itertools, urllib has been changed considerably between Python 2 and Python 3.

The statistics module does not exist for Python 2.

Appendix C

pcinput.py

In many of the exercises in this book, it is useful to have a function available that accepts inputs of a certain type. I created a module called `pcinput` which contains a number of such functions. During many of the exercises in this book, I assume that you have that module available. To get it, download it from `http://www.spronck.net/pythonbook`, or copy the code below to a file called "pcinput.py," and make sure that it is located in same folder where you keep the files with your own code.

Note that these functions are rather ugly as they print error messages if something is wrong. However, nicer functions would be more difficult to use (you would have to know about exceptions, which are not covered until Chapter 17). For the purpose of learning to code Python, they work fine.

Each of the functions asks the user to supply a value of a certain type (a float, an integer, a string, or a capital letter), and returns that value. You can call each of the functions with a string as argument, that will be used as prompt.

pcinput.py

```
def getFloat( prompt ):
    while True:
        try:
            num = float( input( prompt ) )
        except ValueError:
            print( "That is not a number -- please try again" )
            continue
        return num

def getInteger( prompt ):
    while True:
        try:
            num = int( input( prompt ) )
        except ValueError:
            print( "That is not an integer -- please try again" )
            continue
        return num
```

```
def getString( prompt ):
    line = input( prompt )
    return line.strip()

def getLetter( prompt ):
    while True:
        line = input( prompt )
        line = line.strip()
        line = line.upper()
        if len( line ) != 1:
            print( "Please enter exactly one character" )
            continue
        if line < 'A' or line > 'Z':
            print( "Please enter a letter from the alphabet" )
            continue
        return line
```

Appendix D

pcmaze.py

In Chapter 9 an example of searching a maze is presented. In that example a module `pcmaze` is used, that I wrote for this book. The module contains a specific maze, and functions to access features of the maze. To create the module, download it from `http://www.spronck.net/pythonbook`, or copy the code below to a file called "pcmaze.py," and make sure that it is located in same folder where you keep the files with your own code.

pcmaze.py

```
def connected( x, y ):
    if x > y:
        return connected( y, x )
    if (x,y) in ((1,5),(2,3),(3,7),(4,8),(5,6),(5,9),(6,7),
        (8,12),(9,10),(9,13),(10,11),(10,14),(11,12),(11,15),
        (15,16)):
        return True
    return False

def entrance():
    return 1

def exit():
    return 16
```

Appendix E

Test Text Files

In Chapter 16 several small text files are used to demonstrate functionalities. Moreover, in Chapter 26 I use a CSV file as demonstration. You can download these files either from `http://www.spronck.net/pythonbook`, or create them yourself. The contents of these files are given below: just create them with any text editor (you can even do that with IDLE), saving them under the name that is given as title. The first one is a small excerpt from Shakespeare's *Romeo and Juliet*, the second is a traditional English tongue twister, and the third one is taken from Lewis Carroll's *Through the Looking Glass*. The last one is an example CSV file that I built.

pc_rose.txt

```
'Tis but thy name that is my enemy.
Thou art thyself, though not a Montague.
What's Montague? it is nor hand, nor foot,
Nor arm, nor face, nor any other part
Belonging to a man. O, be some other name!
What's in a name? That which we call a rose
By any other name would smell as sweet.
So Romeo would, were he not Romeo call'd,
Retain that dear perfection which he owes
Without that title. Romeo, doff thy name;
And for that name, which is no part of thee,
Take all myself.
```

pc_woodchuck.txt

```
How much wood would a woodchuck chuck
If a woodchuck could chuck wood?
He would chuck, he would, as much as he could,
And chuck as much as a woodchuck would
If a woodchuck could chuck wood.
```

pc_jabberwocky.txt

```
JABBERWOCKY

'Twas brillig, and the slithy toves
 Did gyre and gimble in the wabe;
All mimsy were the borogoves,
 And the mome raths outgrabe.

'Beware the Jabberwock, my son!
 The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
 The frumious Bandersnatch!'

He took his vorpal sword in hand:
 Long time the manxome foe he sought--
So rested he by the Tumtum tree,
 And stood awhile in thought.

And as in uffish thought he stood,
 The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
 And burbled as it came!

One, two! One, two! And through and through
 The vorpal blade went snicker-snack!
He left it dead, and with its head
 He went galumphing back.

'And hast thou slain the Jabberwock?
 Come to my arms, my beamish boy!
O frabjous day! Callooh! Callay!'
 He chortled in his joy.

'Twas brillig, and the slithy toves
 Did gyre and gimble in the wabe;
All mimsy were the borogoves,
 And the mome raths outgrabe.
```

pc_inventory.csv

```
ID,CATEGORY,NAME,STOCK,UNITPRICE
1,Fruit,apple,1000,0.87
2,Fruit,banana,2500,0.34
3,Fruit,cherry,11225,0.07
4,Fruit,durian,0,5.52
5,Cheese,Roquefort,46,12.23
6,Cheese,Blue Stilton,1,19.88
7,Cheese,Gouda,7,11.99
```

```
8,Fruit,orange,355,0.77
9,Fruit,mango,24,1.56
10,Cheese,Cheddar,333,13.15
```

Appendix F

Answers to Exercises

This appendix provides answers to most of the exercises. They are also available in file format from `http://www.spronck.net/pythonbook`.

Note that it is useless to look up answers to exercises if you have not spent a significant amount of time trying to solve the exercises. You can only learn programming by doing. Only use the answers to compare them with your own, completed solutions, or as a final resort should you have no idea how to solve a problem. However, if you cannot solve a problem, it is usually better to return to an earlier part of the book that contains information that you evidently did not understand or forgot about.

Note that very often, the answers given here are only one of many possible ways to solve a problem. If you have found a different way to solve a problem, that might be just fine, though make sure you test your answer extensively to ensure that it is correct.

Moreover, while the answers given here are usually efficient, efficiency is not the first goal when writing code. You should first make sure that your code solves the problem at hand, before you look into making the code more efficient. Readability and maintainability are far more important than efficiency.

Chapter 1

Answer 1.1 A straightforward sorting procedure that first identifies the highest card, then the next highest card, then the lowest-but-one, and then the lowest card, needs six comparisons. You could perform it, for instance, as follows:

Number the cards from 1 to 4, left to right. The number is tied to the spot where the card is, i.e., when you switch two cards, they exchange numbers. Compare cards 1 and 2, and switch them if 1 is higher than 2. Compare cards 2 and 3, and switch them if 2 is higher than 3. Compare cards 3 and 4, and switch them if 3 is higher than 4. Now the highest card will be the number 4. Then, compare cards 1 and 2 again, and switch them if 1 is higher than 2. Compare cards 2 and 3 again, and switch them if 2 is higher than 3. Now the next-highest card is number 3. Finally, compare cards 1 and 2 and switch them if 1 is higher than 2. You have now sorted the four cards, with six comparisons.

To do it with less comparisons, you need a procedure that is a bit more complex. Start again by numbering the cards from 1 to 4. Compare cards 1 and 2, and switch them if 1 is higher than 2. Compare cards 3 and 4, and switch them if 3 is higher than 4. Compare cards 1 and 3. If 1 is higher than 3, you switch card 1 and 3, and also card 2 and 4. The lowest card is now in spot 1, and you also know that card 3 is lower than card 4. Now compare cards 2 and 3. If 2 is lower than 3, you are done, needing only four comparisons. If 2 is higher than 3, you switch 2 and 3. You now still have to compare cards 3 and 4, and switch them if 3 is higher than 4, but then you are done, needing only five comparisons. So you can do the sorting with a guaranteed maximum of five comparisons.

You might think that you can be smart and, after comparing 1 and 2 and maybe switching them, and comparing 3 and 4 and maybe switching them, to compare 2 and 3, because if at that point 2 is smaller than 3, you only need three comparisons. However, should at that point 2 be higher than 3, if you are unlucky you will need six comparisons in total.

For those new to programming, the more efficient procedure which only needs five comparisons is quite hard to design, so do not get discouraged if you cannot come up with it. It is more important that you solve a task, than that you solve it in the most efficient manner.

Answer 1.2 You should draw four boxes on a piece of paper, number them, and put each of the cards in one of the boxes. Tell the person who is following the instructions that "compare the card in box *x* with the card in box *y*" means that they should ask the processor to take up those cards, look at them, put them back in the same spots where they were taken from, and then indicate which of the two is the higher card. Then you can give them instructions for the simple, six-comparisons procedure outlined above:

1. Compare the card in box 1 with the card in box 2. If the card in box 1 is higher than the card in box 2, switch these two cards.
2. Compare the card in box 2 with the card in box 3. If the card in box 2 is higher than the card in box 3, switch these two cards.
3. Compare the card in box 3 with the card in box 4. If the card in box 3 is higher than the card in box 4, switch these two cards.
4. Compare the card in box 1 with the card in box 2. If the card in box 1 is higher than the card in box 2, switch these two cards.
5. Compare the card in box 2 with the card in box 3. If the card in box 2 is higher than the card in box 3, switch these two cards.
6. Compare the card in box 1 with the card in box 2. If the card in box 1 is higher than the card in box 2, switch these two cards.

Using the idea of drawing boxes that you have numbered to refer to the cards put in those boxes is similar to using variables in a computer program. Variables will be explained in one of the early chapters. Trying to write instructions for the more efficient procedure outlined above is harder, because you need nested conditions and an early escape.

Chapter 2

Answer 2.1 You now have Python running on your computer. Congratulations!

Answer 2.2 You will see nothing in the shell (apart from a display of the word RESTART that you always see when running a program). 7/4 is a legal Python statement, so the program will not give an error. The program just calculates 7/4, but does not display the result using a print statement, so the program does not show 1.75. The shell, however, displays the result of running the program. But since a program has no result by itself, there is nothing for the shell to display either. So you see nothing.

Chapter 3

Answer 3.1

answer0301.py

```
print( 60 * (0.6 * 24.95 + 0.75) + (3 - 0.75) )
```

Answer 3.2 Each of the lines should be either **print**("A message") or **print**('A message'). The error in the first line is that it ends in a period. That period should be removed. The error in the second line is that it contains something that is supposed to be a string, but starts with a double quote while it ends with a single quote. Either the double quote should become a single quote, or the single quote should become a double quote. The third line is actually syntactically correct, but probably it was meant to be **print**('A message'), so the f" should be removed.

Answer 3.3

answer0303.py

```
print( 1/0 )
```

Answer 3.4 The problem is that there is one closing parenthesis missing in the first line of code. I actually deleted the closing parenthesis that should be right of the 6, but you cannot know that; you can only count the parentheses in the first statement and see that there is one less closing parenthesis than there are opening parentheses.

The confusing part of this error message is that it says that the error is in the second line of code. The second line of the code, however, is fine. The reason is that since Python has not seen the last required closing parenthesis on the first line, it starts looking for it on the second line. And while doing that, it notices that something is going wrong, and it reports the error. Basically, while trying to process the second line, Python finds that it cannot do that, so it indicates that there is an error with the second line.

You will occasionally encounter this in your own code: an error is reported for a certain line of code, but the error is actually made in one of the previous lines. Such errors often encompass the absence of a required parenthesis or single or double quote. Keep this in mind.

Answer 3.5

answer0305.py

```
print( str( (14 + 535) % 24 ) + ".00" )
```

Chapter 4

Answer 4.1

answer0401.py

```
# This program calculates the average of three variables,
# var1, var2, and var3
var1 = 12.83
var2 = 99.99
var3 = 0.12
average = (var1 + var2 + var3) / 3 # Calculate the average
print( average ) # May look a bit ugly, but we might make this
# look a bit better when we have learned about formatting
```

Answer 4.2

answer0402.py

```
pi = 3.14159
radius = 12
print( "The surface area of a circle with radius",
    radius, "is", pi * radius * radius )
```

Answer 4.3

answer0403.py

```
CENTS_IN_DOLLAR = 100
CENTS_IN_QUARTER = 25
CENTS_IN_DIME = 10
CENTS_IN_NICKEL = 5

amount = 1156
cents = amount

dollars = int( cents / CENTS_IN_DOLLAR )
cents -= dollars * CENTS_IN_DOLLAR
quarters = int( cents / CENTS_IN_QUARTER )
cents -= quarters * CENTS_IN_QUARTER
dimes = int( cents / CENTS_IN_DIME )
cents -= dimes * CENTS_IN_DIME
nickels = int( cents / CENTS_IN_NICKEL )
cents -= nickels * CENTS_IN_NICKEL
cents = int( cents )

print( amount / CENTS_IN_DOLLAR, "consists of:" )
print( "Dollars:", dollars )
print( "Quarters:", quarters )
print( "Dimes:", dimes )
print( "Nickels:", nickels )
```

```
print( "Pennies:", cents )
```

Answer 4.4

answer0404.py

```
a = 17
b = 23
print( "a =", a, "and b =", b )
a += b
b = a - b
a -= b
print( "a =", a, "and b =", b )
```

Chapter 5

Answer 5.1

answer0501.py

```
s = input( "Enter a string: " )
print( "You entered", len( s ), "characters" )
```

Answer 5.2

answer0502.py

```
from pcinput import getFloat
from math import sqrt

side1 = getFloat( "Please enter the length of the first side: " )
side2 = getFloat( "Please enter the length of the second side: ")
side3 = sqrt( side1 * side1 + side2 * side2 )
print( "The length of the diagonal is {:.3f}.".format( side3 ) )
```

Answer 5.3

answer0503.py

```
from pcinput import getFloat

num1 = getFloat( "Please enter number 1: " )
num2 = getFloat( "Please enter number 2: " )
num3 = getFloat( "Please enter number 3: " )

print( "The largest is", max( num1, num2, num3 ) )
print( "The smallest is", min( num1, num2, num3 ) )
print( "The average is", round( (num1 + num2 + num3)/3, 2 ) )
```

Answer 5.4

answer0504.py

```
from math import exp

s = "e to the power of {:2d} is {:>9.5f}"
print( s.format( -1, exp( -1 ) ) )
print( s.format( 0, exp( 0 ) ) )
print( s.format( 1, exp( 1 ) ) )
print( s.format( 2, exp( 2 ) ) )
print( s.format( 3, exp( 3 ) ) )
```

Answer 5.5

answer0505.py

```
from random import random

print( "A random integer between 1 and 10 is",
    1 + int( random() * 10 ) )
```

Chapter 6

Answer 6.1

answer0601.py

```
from pcinput import getFloat

grade = getFloat( "Please enter a grade: " )
check = int( grade * 10 )
if grade < 0 or grade > 10:
    print( "Grades have to be in the range 0 to 10." )
elif check%5 != 0 or check != grade*10:
    print( "Grades should be rounded to the nearest half point.")
elif grade >= 8.5:
    print( "Grade A" )
elif grade >= 7.5:
    print( "Grade B" )
elif grade >= 6.5:
    print( "Grade C" )
elif grade >= 5.5:
    print( "Grade D" )
else:
    print( "Grade F" )
```

Answer 6.2 The grade will always be "D" or "F," as the tests are placed in the incorrect order. E.g., if score is 85, then it is not only greater than 80.0, but also greater than 60.0, so that the grade becomes "D."

Answer 6.3

answer0603.py

```
from pcinput import getString

s = getString( "Please enter a string: " )
count = 0
if ("a" in s) or ("A" in s):
    count += 1
if ("e" in s) or ("E" in s):
    count += 1
if ("i" in s) or ("I" in s):
    count += 1
if ("o" in s) or ("O" in s):
    count += 1
if ("u" in s) or ("U" in s):
    count += 1

if count == 0:
    print( "There are no vowels in the string." )
elif count == 1:
    print( "There is only one different vowel in the string." )
else:
    print( "There are", count, "different vowels in the string.")
```

Answer 6.4

answer0604.py

```
from pcinput import getFloat
from math import sqrt

a = getFloat( "A: " )
b = getFloat( "B: " )
c = getFloat( "C: " )

if a == 0:
    if b == 0:
        print( "There is not even an unknown in this equation!" )
    else:
        print( "There is one solution, namely", -c/b )
else:
    discriminant = b*b - 4*a*c
    if discriminant < 0:
        print( "There are no solutions" )
    elif discriminant == 0:
```

```
        print( "There is one solution, namely", -b/(2*a) )
    else:
        print( "There are two solutions, namely",
                (-b+sqrt(discriminant))/(2*a), "and",
                (-b-sqrt(discriminant))/(2*a) )
```

Chapter 7

Answer 7.1

answer0701.py

```
from pcinput import getInteger

num = getInteger( "Give a number: " )
i = 1
while i <= 10:
    print( i, "*", num, "=", i*num )
    i += 1
```

Answer 7.2

answer0702.py

```
from pcinput import getInteger

num = getInteger( "Give a number: " )
for i in range( 1, 11 ):
    print( i, "*", num, "=", i*num )
```

Answer 7.3

answer0703.py

```
from pcinput import getInteger

TOTAL = 10
largest = 0
smallest = 0
div3 = 0

for i in range( TOTAL ):
    num = getInteger( "Please enter number "+str( i+1 )+": " )
    if num%3 == 0:
        div3 += 1
    if i == 0:
        smallest = num
        largest = num
        continue
```

```
    if num < smallest:
        smallest = num
    if num > largest:
        largest = num

print( "Smallest is", smallest )
print( "Largest is", largest )
print( "Dividable by 3 is", div3 )
```

Answer 7.4

answer0704.py

```
bottles = 10
s = "s"

while bottles != "no":
    print( "{0} bottle{1} of beer on the wall, "\
        "{0} bottle{1} of beer.".format( bottles, s ) )
    bottles -= 1
    if bottles == 1:
        s = ""
    elif bottles == 0:
        s = "s"
        bottles = "no"
    print( "Take one down, pass it around, {} bottle{} "\
        "of beer on the wall.".format( bottles, s ) )
```

The backslash (\) at the end of two of the strings in this code concatenates the string on the next line to the string after which the backslash is found. I use that feature here to make the code fit in the box. Use of the backslash to allow code to span multiple lines is discussed in Chapter 10. You do not need it at this time: you can just put the whole **print()** statement on one long line.

Answer 7.5

answer0705.py

```
num1 = 0
num2 = 1
print( 1, end=" " )

while True:
    num3 = num1 + num2
    if num3 > 1000:
        break
    print( num3, end=" " )
    num1 = num2
    num2 = num3
```

Answer 7.6

answer0706.py

```
from pcinput import getString

word1 = getString( "Give word 1: " )
word2 = getString( "Give word 2: " )
common = ""
for letter in word1:
    if (letter in word2) and (letter not in common):
        common += letter
if common == "":
    print( "The words share no characters." )
else:
    print( "The words have the following in common:", common )
```

Answer 7.7

answer0707.py

```
from random import random

DARTS = 100000
hits = 0
for i in range( DARTS ):
    x = random()
    y = random()
    if x*x + y*y < 1:
        hits += 1

print( "A reasonable approximation of pi is", 4 * hits / DARTS )
```

Answer 7.8

answer0708.py

```
from random import randint
from pcinput import getInteger

answer = randint( 1, 1000 )
count = 0

while True:
    guess = getInteger( "What is your guess? " )
    if guess < 1 or guess > 1000:
        print( "Your guess should be between 1 and 1000" )
        continue
    count += 1
    if guess < answer:
        print( "Higher" )
```

```
    elif guess > answer:
        print( "Lower" )
    else:
        print( "You guessed it!" )
        break

if count == 1:
    print( "You needed only one guess. Lucky bastard." )
else:
    print( "You needed", count, "guesses." )
```

Answer 7.9

answer0709.py

```
from pcinput import getLetter
from sys import exit

count = 0
lowest = 0
highest = 1001
print( "Take a number in mind between 1 and 1000." )

while True:
    guess = int( (lowest + highest) / 2 )
    count += 1
    prompt = "I guess "+str( guess )+". Is your number"+\
        " (L)ower or (H)igher, or is this (C)orrect? "
    response = getLetter( prompt )
    if response == "C":
        break
    elif response == "L":
        highest = guess
    elif response =="H":
        lowest = guess
    else:
        print( "Please enter H, L, or C." )
        continue
    if lowest >= highest-1:
        print( "You must have made a mistake,",
            "because you said that the answer is higher than",
            lowest, "but also lower than", highest )
        print( "I quit this game" )
        exit()

if count == 1:
    print( "I needed only one guess! I must be a mind reader." )
else:
    print( "I needed", count, "guesses." )
```

Answer 7.10

answer0710.py

```
from pcinput import getInteger

num = getInteger( "Please enter a number: " )
if num < 2:
    print( num, "is not prime" )
else:
    i = 2
    while i*i <= num:
        if num%i == 0:
            print( num, "is not prime" )
            break
        i += 1
    else:
        print( num, "is prime" )
```

There is a small trick in this code to speed up the calculation enormously. The code only tests dividers up to the square root of `num` (i.e., it tests up to the point that `i*i > num`). The reason that it does this, is that if there is a number bigger than the square root of `num` that divides it, there must necessarily also be one smaller than that square root. For instance, if the number is 21, the square root is between 4 and 5. So the algorithm only tests numbers up to 4. There is a divider for 21 higher than 4, namely 7. But that means there must also be one lower than (or equal to) 4, and indeed, there is one, namely 3. This is a gigantic speedup compared to testing all numbers up to `num`; for instance, if `num` is somewhere in the neighborhood of 1 million, and is prime, you would need to test about 1 million numbers if you test all of them up to `num`, while this particular algorithm would only test about one thousand.

If you found this trick, great! If not, don't worry: as long as your code works, you are doing just fine. The hard part is getting the code to work in the first place. As long as you accomplish that, you are well under way to becoming a great programmer.

One final note on this code: you should test it extensively! It is very easy to go wrong on particular numbers. In this case, make sure you test a negative number, zero, 1 (which all three should be reported as not prime), 2 (which is the lowest prime number and the only even one), 3 (which is the lowest odd prime), several not-prime numbers, several prime numbers, and numbers which are the square of a prime. Even better, if you can write a for loop around your code that tests all numbers between, for instance, -10 and 100, then you are really doing an extensive test of your code.

Answer 7.11

answer0711.py

```
num = 9
print( ". |", end="" )
for i in range( 1, num+1 ):
    print( "{:>3}".format( i ), end="" )
print()
```

```
for i in range( 3*(num+1) ):
    print( "-", end="" )
print()
for i in range( 1, num+1 ):
    print( i, "|", end="" )
    for j in range( 1, num+1 ):
        print( "{:>3}".format( i*j ), end="" )
    print()
```

Answer 7.12

answer0712.py

```
for i in range( 1, 101 ):
    for j in range( 1, i ):
        for k in range( j, i ):
            if j*j + k*k == i:
                print( "{} = {}**2 + {}**2".format( i, j, k ) )
```

A more efficient version of this algorithm is:

answer0712a.py

```
from math import sqrt

for i in range( 1, 101 ):
    for j in range( 1, int( sqrt( i ) )+1 ):
        for k in range( j, int( sqrt( i ) )+1 ):
            if j*j + k*k == i:
                print( "{} = {}**2 + {}**2".format( i, j, k ) )
```

Answer 7.13

answer0713.py

```
from random import randint

TRIALS = 10000
DICE = 5
success = 0

for i in range( TRIALS ):
    lastdie = 0
    for j in range( DICE ):
        roll = randint( 1, 6 )
        if roll < lastdie:
            break
        lastdie = roll
    else:
        success += 1
```

```
print( "The probability of an increasing sequence",
    "of five die rolls is {:.3f}".format( success/TRIALS ) )
```

Answer 7.14

answer0714.py

```
for A in range( 1, 10 ):
    for B in range( 10 ):
        if B == A:
            continue
        for C in range( 10 ):
            if C == A or C == B:
                continue
            for D in range( 1, 10 ):
                if D == A or D == B or D == C:
                    continue
                num1 = 1000*A + 100*B + 10*C + D
                num2 = 1000*D + 100*C + 10*B + A
                if num1 * 4 == num2:
                    print( "A={}, B={}, C={}, D={}".format(
                        A, B, C, D ) )
```

The program will generate all combinations of A, B, C, and D that solve the puzzle. In this case there is only one, but the program continues seeking for more solutions until it has tested all possibilities. Fortunately, that process is very fast. If you want to stop the program once it has found a solution, the best approach is to put most of it in a function and return from that function as soon as a solution is found. Creating your own functions is discussed in the next chapter.

Answer 7.15

answer0715.py

```
PIRATES = 5
coconuts = 0
while True:
    coconuts += 1
    pile = coconuts
    for i in range( PIRATES ):
        if pile % PIRATES != 1:
            break
        pile = (PIRATES-1) * int( (pile - 1) / PIRATES )
    if pile % PIRATES == 1:
        break
print( coconuts )
```

This solution start with zero coconuts and keeps adding coconuts to the pile until the pile is of a size whereby it can let each pirate take his share leaving one coconut, removing

the share from the pile, and removing the remaining coconut. Testing whether a single coconut remains after division is done using the modulo operator. After all pirates took their share, the problem is solved if there again would be one coconut remaining when the pile is equally divided amongst the pirates.

Isn't it a bit surprising that you can solve a problem for which the description is so long in so few lines of code?

Answer 7.16 First I give the solution that simulates each Triangle Crawler separately. The advantage of coding it like this is that it is not hard to accomplish. The disadvantage is that this code is rather slow. It should be pretty easy to understand how this solution works:

answer0716.py

```
from random import randint

NUMCRAWLERS = 100000
totalage = NUMCRAWLERS # They all live at least one day

for i in range( NUMCRAWLERS ):
    if randint( 0, 2 ): # Don't die on first day
        totalage += 1
        while randint( 0, 1 ): # Don't die on following day
            totalage += 1

print( "{:.2f}".format( totalage / NUMCRAWLERS ) )
```

The second solution considers the population as one big whole, and simply divides it up into chunks. The first day, it takes out a chunk the size of a third of the population, adds to the total age the fact that this chunk only lived for one day, and lets the rest remain. On the second day, it takes out half the population, and adds to the total age that those lived for two days. On the third day, it again takes out half of the remainder, and adds up that those lived three days. This continues until no Crawlers remain.

It should be noted that often there will be a Crawler remaining which could not be divided (as a Crawler either survives or dies, but it cannot be Schrödinger's Cat). Such a Crawler is assigned randomly to the group that dies or the group that survives. You can see that this code can deal with huge numbers of Triangle Crawlers without problems.

answer0716a.py

```
from random import randint

NUMCRAWLERS = 1000000000
num = NUMCRAWLERS
die = int( num / 3 )
if NUMCRAWLERS % 3:
    if randint( 0, NUMCRAWLERS % 3 ): # Remaining on day 1
        die += 1
totalage = die # Day-1 deaths added to totalage
num -= die
```

```
age = 2
while num > 0:
    die = int( num / 2 )
    num -= die # Kill off half the population
    if die != num: # There is a single remaining
        if randint( 0, 1 ): # Decide on kill of single
            die, num = num, die # swap values
    totalage += die * age
    age += 1

print( "{:.2f}".format( totalage / NUMCRAWLERS ) )
```

Finally, I present a solution that I did not suggest, but that works great. It is very brief, incredibly fast, and gives an almost exact answer. It simply calculates $\frac{1}{3} + \frac{2}{3}(\frac{1}{2}(2 + \frac{1}{2}(3 + \frac{1}{2}(4 + \frac{1}{2}(5 + ...)))))$ for a limited number of terms, in this case 100, which is plenty to get a highly accurate approximation. Of course, this a calculation rather than a simulation, but it is the mathematical expression that you were actually asked to solve.

answer0716b.py

```
estimate = 1/3
remainder = 2/3
for days in range( 2, 101 ):
    remainder /= 2
    estimate += remainder * days
print( "{:.2f}".format( estimate ) )
```

By the way, the exact answer is $2\frac{1}{3}$ days; an approximation should give 2.33 or 2.34.

Chapter 8

Answer 8.1

answer0801.py

```
from pcinput import getInteger

# multiplicationtable gets an integer as parameter.
# It prints the multiplication table for that integer.
def multiplicationtable( n ):
    i = 1
    while i <= 10:
        print( i, "*", n, "=", i*n )
        i += 1

num = getInteger( "Give a number: " )
multiplicationtable( num )
```

Answer 8.2

answer0802.py

```
from pcinput import getString

# commoncharacters gets two strings as parameters.
# It returns the number of characters that they have in common.
def commoncharacters( w1, w2 ):
    common = ""
    for letter in w1:
        if (letter in w2) and (letter not in common):
            common += letter
    return len( common )

word1 = getString( "Give word 1: " )
word2 = getString( "Give word 2: " )

num = commoncharacters( word1, word2 )
if num <= 0:
    print( "The words share no characters." )
elif num == 1:
    print( "The words have one character in common" )
else:
    print( "The words have", num, "characters in common")
```

Answer 8.3

answer0803.py

```
# The function gregoryLeibnitz approximates pi using the Gregory-
# Leibnitz series. It gets one parameter, which is an integer
# that indicates how many terms are calculated. It returns the
# approximation as a float.
def gregoryLeibnitz( num ):
    appr = 0
    for i in range( num ):
        if i%2 == 0:
            appr += 1/(1 + i*2)
        else:
            appr -= 1/(1 + i*2)
    return 4*appr

print( gregoryLeibnitz( 50 ) )
```

Answer 8.4

answer0804.py

```
from pcinput import getFloat
from math import sqrt
```

```
# This function solves a quadratic equation.
# Its parameters are the numeric values for A, B, and C in the
# equation Ax**2 + Bx + C = 0. It returns three values: an int
# 0, 1, or 2, indicating the number of solutions, followed by two
# numbers which are the solutions. With no solutions, both
# solutions are set to zero. With one solution, it is returned as
# the first of the two, while the other is set to zero.
def quadraticFormula( a, b, c ):
    if a == 0:
        if b == 0:
            return 0, 0, 0
        return 1, -c/b, 0
    discriminant = b*b - 4*a*c
    if discriminant < 0:
        return 0, 0, 0
    elif discriminant == 0:
        return 1, -b/(2*a), 0
    else:
        return 2, (-b+sqrt(discriminant))/(2*a), \
            (-b-sqrt(discriminant))/(2*a)

num, sol1, sol2 = quadraticFormula( getFloat( "A: " ),
    getFloat( "B: " ), getFloat( "C: " ) )
if num == 0:
    print( "There are no solutions" )
elif num == 1:
    print( "There is one solution, namely:", sol1 )
else:
    print( "There are two solutions, namely:", sol1, "and", sol2)
```

Answer 8.5

answer0805.py

```
from pcinput import getInteger

def getNumber( prompt ):
    while True:
        num = getInteger( prompt )
        if num < 0 or num > 1000:
            print( "Please enter a number between 0 and 1000" )
            continue
        return num

def main():
    while True:
        x = getNumber( "Enter number 1: " )
        if x == 0:
            break
```

```
        y = getNumber( "Enter number 2: " )
        if y == 0:
            break
        if x%y == 0 or y%x == 0:
            print( "Error: the numbers cannot be dividers" )
            return
        print( "Multiplication of", x, "and", y, "gives", x * y )
    print( "Goodbye!" )

if __name__ == '__main__':
    main()
```

Answer 8.6

answer0806.py

```
# Calculates the factorial of parameter n, which must be an
# integer. Returns the value of the factorial as an integer.
def factorial( n ):
    value = 1
    for i in range( 2, n+1 ):
        value *= i
    return value

# Calculates n over k; parameters n and k are integers. Returns
# the value n over k as an integer (because it always must be
# an integer).
def binomialCoefficient( n, k ):
    if k > n:
        return 0
    return int( factorial( n ) /
        (factorial( k )*factorial( n - k )) )

def main():
    print( factorial( 5 ) )
    print( binomialCoefficient( 8, 3 ) )

if __name__ == '__main__':
    main()
```

Answer 8.7 The code tries to print the return value of the function `area_of_triangle()`, but since this function has no return value, it prints the word **`None`**. To display the output of a function, you can either print the output in the function and then call it without using the return value, or you let the function produce the output, return it, and print the return value of the function in your main program. Not both. In general it is preferable if functions do not do the printing themselves, because it makes them more generally usable (for instance, the function `area_of_triangle()`, if it just returns the area instead of printing it, you can not only use the function to print the area, but also use the area in calculations). If you do not understand the explanation given here, revisit Chapter 5.

Chapter 9

Answer 9.1

answer0901.py

```
def fib( n ):
    if n <= 2:
        return 1
    return fib( n-1 ) + fib( n-2 )

print( fib( 20 ) )
```

Answer 9.2

answer0902.py

```
def fib( n, depth ):
    indent = 6 * depth * " "
    print( "{}fib({})".format( indent, n ) )
    if n <= 2:
        print( "{}return {}".format( indent, 1 ) )
        return 1
    value = fib( n-1, depth+1 ) + fib( n-2, depth+1 )
    print( "{}return {}".format( indent, value ) )
    return value

print( fib( 5, 0 ) )
```

Answer 9.3 Since the Fibonacci sequence can just as easily be implemented as an iterative function (you did this in the previous chapter), doing it as a recursive function is not a good idea. The reasons are the same as for the factorial, explained in the chapter. There is the additional reason that the recursive definition basically calculates all terms of the sequence multiple times, as you can see when you look at the output for the second exercise, while calculating them just once suffices.

Answer 9.4

answer0904.py

```
def gcd( m, n ):
    if m % n == 0:
        return n
    return gcd( n, m%n )

print( gcd( 7*5*13, 2*3*7*11 ) )
```

Interestingly, while the definition of the algorithm distinguishes the smallest and the largest number, you actually do not have to make that distinction in your code. If you call the function with the two exchanged, it just leads to one extra recursive call.

Answer 9.5 The problem in this code is that when I enter a string with two illegal characters, then it will recursively call itself twice, namely once for each of the letters. So, for instance, if the user enters `"route67"`, it will first call itself recursively for the `"6"`. If the user then enters a correct string, the function should end. Instead, it continues checking the original input `"route67"`, finds the `"7"`, and calls itself recursively again to ask for yet another string. I could explain how you can solve this problem by fixing the code, but the real fix here is that you should not write recursive functions that interact with the user.

Answer 9.6

answer0906.py

```
SIZE = 4

def solve_hanoi( pole_from, pole_tmp, pole_to, size ):
    if size == 1:
        print( "Disc 1 from", pole_from, "to", pole_to )
        return 1
    moves = solve_hanoi( pole_from, pole_to, pole_tmp, size-1 )
    print( "Disc", size, "from", pole_from, "to", pole_to )
    moves += 1+solve_hanoi( pole_tmp, pole_from, pole_to, size-1)
    return moves

moves = solve_hanoi( 'A', 'B', 'C', SIZE )
print( moves, "moves needed" )
```

An iterative approach to solve this puzzle is the following. Repeat these two steps until the problem is solved: (1) Move disc 1 to the next pole; (2) Move a disc that is not disc 1 to another pole (there is always exactly one disc for which you can do this). The only decision that remains is whether you should move disc 1 "clockwise" or "counter-clockwise." If the size of the biggest disc is even, go clockwise (A to B, B to C, or C to A). Otherwise, go counter-clockwise (A to C, C to B, or B to A). This solution is fast and avoids recursion, which means that it works for bigger disc sizes than the recursive solution tends to allow. However, it is more complex to implement as you have to represent each pole as a list of discs (lists will be introduced in Chapter 12), and you have to find a way to check whether the game is solved (you can just take the number of necessary moves, which is $2^N - 1$).

Chapter 10

Answer 10.1

answer1001.py

```
text = """And Saint Attila raised the hand grenade up on high,
saying, "O Lord, bless this thy hand grenade, that with it
thou mayst blow thine enemies to tiny bits, in thy mercy."
And the Lord did grin. And the people did feast upon the lambs,
and sloths, and carp, and anchovies, and orangutans, and
breakfast cereals, and fruit bats, and large chu..."""
```

```
counta, counte, counti, counto, countu = 0, 0, 0, 0, 0
for c in text:
    if c.upper() == "A":
        counta += 1
    elif c.upper() == "E":
        counte += 1
    elif c.upper() == "I":
        counti += 1
    elif c.upper() == "O":
        counto += 1
    elif c.upper() == "U":
        countu += 1

print( "Counts: a={}, e={}, i={}, o={}, u={}".format(
    counta, counte, counti, counto, countu ) )
```

Answer 10.2

answer1002.py

```
text = """And sending tinted postcards of places they don't
realise they haven't even visited to 'All at nu[m]ber 22, weather
w[on]derful, our room is marked with an 'X'. Wish you were here.
Food very greasy but we've found a charming li[t]tle local place
hidden awa[y ]in the back streets where they serve Watney's Red
Barrel and cheese and onion cris[p]s and the accordionist pla[y]s
"Maybe i[t]'s because I'm a Londoner"' and spending four days on
the tarmac at Luton airport on a five-day package tour wit[h]
n[o]thing to eat but dried Watney's sa[n]dwiches..."""

start = -1
while True:
    start = text.find( "[", start+1 )
    if start < 0:
        break
    finish = text.find( "]", start )
    if finish < 0:
        break
    print( text[start+1:finish], end="" )
    start = finish
print()
```

Answer 10.3

answer1003.py

```
ch = "A"
while ch <= "Z":
    print( ch, end=" " )
    ch = chr( ord( ch )+1 )
```

```
print()

for i in range( 26 ):
    rotr13 = (i + 13)%26
    ch = chr( ord( "A" ) + rotr13 )
    print( ch, end=" " )
```

Answer 10.4

answer1004.py

```
text = """How much wood would a woodchuck chuck
If a woodchuck could chuck wood?
He would chuck, he would, as much as he could,
And chuck as much as a woodchuck would
If a woodchuck could chuck wood."""

def clean( s ):
    news = ""
    s = s.lower()
    for c in s:
        if c >= "a" and c <= "z":
            news += c
        else:
            news += " "
    return news

count = 0
for word in clean( text ).split():
    if word == "wood":
        count += 1

print( "Number of times \"wood\" occurs in the text:", count )
```

Note that in the example text, the word "wood" never occurs with a capital. For a solid test, you should insert "wood" written with one or more capitals in the text somewhere.

Answer 10.5

answer1005.py

```
text = "Hello, world!"
newtext = ""

while len( text ) > 0:
    i = 0
    ch = text[i]
    j = 1
    while j < len( text ):
        if text[j] < ch:
```

```
            ch = text[j]
            i = j
        j += 1
    text = text[:i] + text[i+1:]
    newtext += ch

print( newtext )
```

Answer 10.6

answer1006.py

```
from sys import exit

sentence = "as it turned out our chance meeting with REverend \
aRTHUR BElling was was to change our whole way of life, and \
every sunday we'd hurry along to St lOONY up the Cream BUn \
and Jam..."

# Just check if there really is a sentence.
if len( sentence  ) <= 0:
    exit()

# Capitalize first letter
newsentence = sentence[0].upper() + sentence[1:]

wordlist = newsentence.split()
lastword = ""
newsentence = ""

for word in wordlist:

    # Correct double capitals
    if len( word ) > 2 and word[0] >= "A" and word[0] <= "Z" and\
        word[1] >= "A" and word[1] <= "Z" and word[2] >= "a" and\
        word[2] <= "z":
        word = word[0] + word[1].lower() + word[2:]

    # Capitalize days
    day = word.lower()
    if day == "sunday" or day == "monday" or day == "tuesday" or\
        day == "wednesday" or day == "thursday" or \
        day == "friday" or day == "saturday":
        word = day[0].upper() + day[1:]

    # Correct CAPS LOCK
    if word[0] >= "a" and word[0] <= "z":
        allcaps = True
        for c in word[1:]:
            if not (c >= "A" and c <= "Z"):
```

```
                allcaps = False
                break
        if allcaps:
            word = word[0].upper() + word[1:].lower()

    # Remove duplicates
    if word == lastword:
        continue

    newsentence += word + " "
    lastword = word

newsentence = newsentence.strip()
print( newsentence )
```

Chapter 11

Answer 11.1 I created `display_complex()` to nicely format complex numbers (it will not show a real part which is zero, or a 1 in front of the i, or clump a plus and minus together). Creating such a function was not a requirement.

answer1101.py

```
def add_complex( c1, c2 ):
    return (c1[0] + c2[0], c1[1] + c2[1])

def display_complex( c ):
    s = "("
    if c[1] == 0:
        return str( c[0] )
    elif c[0] != 0:
        s += str( c[0] )
        if c[1] > 0:
            s += "+"
    if c[1] != 1:
        if c[1] == -1:
            s += "-"
        else:
            s += str( c[1] )
    s += "i)"
    return s

num1 = (2,1)
num2 = (0,2)
print( display_complex( num1 ), "+", display_complex( num2 ),
    "=", display_complex( add_complex( num1, num2 ) ) )
```

Answer 11.2 I used a less nice version of `display_complex()` for this solution.

answer1102.py

```
def multiply_complex( c1, c2 ):
    return (c1[0]*c2[0] - c2[1]*c1[1], c1[0]*c2[1] + c1[1]*c2[0])

def display_complex( c ):
    return "({},{}i)".format( c[0], c[1] )

num1 = (2,1)
num2 = (0,2)
print( display_complex( num1 ), "*", display_complex( num2 ),
    "=", display_complex( multiply_complex( num1, num2 ) ) )
```

Answer 11.3

answer1103.py

```
inttuple = ( 1, 2, ( 3, 4 ), 5, ( ( 6, 7, 8, ( 9, 10 ), 11 ), 12,
    13 ), ( ( 14, 15, 16 ), ( 17, 18, 19, 20 ) ) )

def display_inttuple( it ):
    for element in it:
        if isinstance( element, int ):
            print( element, end=" ")
        else:
            display_inttuple( element )

display_inttuple( inttuple )
```

Chapter 12

Answer 12.1

answer1201.py

```
from random import choice

answers = [ "It is certain", "It is decidedly so", "Without a \
doubt", "Yes, definitely", "You may rely on it", "As I see it, \
yes", "Most likely", "Outlook good", "Yes", "Signs point to yes",
"Reply hazy try again", "Ask again later", "Better not tell you \
now", "Cannot predict now", "Concentrate and ask again", "Don't \
count on it", "My reply is no", "My sources say no", "Outlook \
not so good", "Very doubtful" ]

input( "Ask the magic 8-ball your question: " )
print( "The magic 8-ball says:", choice( answers ) )
```

The `choice()` function from `random` selects a random item from a list. You could also have used `randint()`, selecting an index from the range 0 to **`len`**`( answers )-1`.

Answer 12.2

answer1202.py

```
from random import randint

deck = []
for value in ("Ace", "2", "3", "4", "5", "6", "7", "8",
    "10", "Jack", "Queen", "King"):
    for suit in ("Hearts", "Spaces", "Clubs", "Diamonds"):
        deck.append( value + " of " + suit )

for i in range( len( deck ) ):
    j = randint( i, len( deck )-1 )
    deck[i], deck[j] = deck[j], deck[i]

for card in deck:
    print( card )
```

Answer 12.3

answer1203.py

```
fifo = []
while True:
    k = input( "> " )
    if k == "":
        break
    if k != "?":
        fifo.append( k )
    elif len( fifo ) > 0:
        print( fifo.pop(0) )
    else:
        print( "List is empty" )
```

Answer 12.4

answer1204.py

```
text = """Now, it's quite simple to defend yourself against a
man armed with a banana. First of all you force him to drop
the banana; then, second, you eat the banana, thus disarming
him. You have now rendered him helpless."""

def count_letter( x ):
    return x[0], -ord(x[1])

countlist = []
```

```
for i in range( 26 ):
    countlist.append( [0, chr(ord("a")+i)] )

for letter in text.lower():
    if letter >= "a" and letter <= "z":
        countlist[ord(letter)-ord("a")][0] += 1

countlist.sort( reverse=True, key=count_letter )

for count in countlist:
    print( "{:3}: {}".format( count[0],count[1] ) )
```

Answer 12.5 There are basically two general approaches to this program: either you make a list of numbers, or you make a list of booleans and use the index as numbers. In the solution below, I used the booleans method. I start the list at zero, as that saves many subtractions while only adding one useless boolean. The method is really fast because I can work with indices.

answer1205.py

```
numbers = 101 * [True]
numbers[1] = False
for i in range( 1, len( numbers ) ):
    if not numbers[i]:
        continue
    print( i, end=" " )
    j = 2
    while j*i < len( numbers ):
        numbers[j*i] = False
        j += 1
```

In the code below, I use the alternative method. I use the **range()** function to create the list. I actually remove items from the list when I know that they are not prime, which reduces the number of list manipulations needed when increasing numbers are removed.

answer1205a.py

```
MAXNUM = 100
numbers = list( range(2,MAXNUM+1) )
i = 0
while i < len( numbers ):
    j = i+1
    while j < len( numbers ):
        if numbers[j]%numbers[i]:
            j += 1
        else:
            numbers.pop(j)
    i += 1
for i in numbers:
    print( i, end=" " )
```

Answer 12.6

answer1206.py

```
from pcinput import getInteger

EMPTY = "-"
PLAYERX = "X"
PLAYERO = "O"
MAXMOVE = 9

def display_board( b ):
    print( "  1 2 3" )
    for row in range( 3 ):
        print( row+1, end=" ")
        for col in range( 3 ):
            print( b[row][col], end=" " )
        print()

def opponent( p ):
    if p == PLAYERX:
        return PLAYERO
    return PLAYERX

def getRowCol( player, what ):
    while True:
        num = getInteger( "Player "+player+", which "+what+
            " do you play? " )
        if num < 1 or num > 3:
            print( "Please enter 1, 2, or 3" )
            continue
        return num

def winner( b ):
    for row in range( 3 ):
        if b[row][0] != EMPTY and b[row][0] == b[row][1] \
            and b[row][0] == b[row][2]:
            return True
    for col in range( 3 ):
        if b[0][col] != EMPTY and b[0][col] == b[1][col] \
            and b[0][col] == b[2][col]:
            return True
    if b[1][1] != EMPTY:
        if b[1][1] == b[0][0] and b[1][1] == b[2][2]:
            return True
        if b[1][1] == b[0][2] and b[1][1] == b[2][0]:
            return True
    return False

board = [[EMPTY,EMPTY,EMPTY],[EMPTY,EMPTY,EMPTY],
    [EMPTY,EMPTY,EMPTY]]
```

```
player = PLAYERX

display_board( board )
move = 0
while True:
    row = getRowCol( player, "row" )
    col = getRowCol( player, "column" )
    if board[row-1][col-1] != EMPTY:
        print( "There is already a piece at row", row,
            "and column", col )
        continue
    board[row-1][col-1] = player
    display_board( board )
    if winner( board ):
        print( "Player", player, "won!" )
        break
    move += 1
    if move == MAXMOVE:
        print( "It's a draw." )
        break
    player = opponent( player )
```

Answer 12.7

answer1207.py

```
from pcinput import getString
from random import randint

EMPTY = "."
BATTLESHIP = "X"
SHIPS = 3
WIDTH = 4
HEIGHT = 3

def displayBoard( b ):
    print( "  ", end="" )
    for col in range( WIDTH ):
        print( chr( ord("A")+col ), end=" " )
    print()
    for row in range( HEIGHT ):
        print( row+1, end=" ")
        for col in range( WIDTH ):
            print( b[row][col], end=" " )
        print()

def placeBattleships( b ):
    for i in range( SHIPS ):
        while True:
            x = randint( 0, WIDTH-1 )
```

```
            y = randint( 0, HEIGHT-1 )
            if b[y][x] == BATTLESHIP:
                continue
            if x > 0 and b[y][x-1] == BATTLESHIP:
                continue
            if x < WIDTH-1 and b[y][x+1] == BATTLESHIP:
                continue
            if y > 0 and b[y-1][x] == BATTLESHIP:
                continue
            if y < HEIGHT-1 and b[y+1][x] == BATTLESHIP:
                continue
            break
        b[y][x] = BATTLESHIP

def getTarget():
    while True:
        cell = getString( "Which cell do you target? " ).upper()
        if len( cell ) != 2:
            print( "Please enter a cell as XY,",
                "where X is a letter and Y a digit" )
            continue
        if cell[0] not in "ABCD":
            print( "The first character of the cell",
                "should be a letter in the range A-"+
                chr( ord("A")+WIDTH-1 ) )
            continue
        if cell[1] not in "123":
            print( "The second character of the cell should be",
                "a digit in the range 1-"+str( HEIGHT ) )
            continue
        return ord(cell[0])-ord("A"), ord(cell[1])-ord("1")

board = []
for i in range( HEIGHT ):
    row = WIDTH * [EMPTY]
    board.append( row )
placeBattleships( board )
displayBoard( board )

hits = 0
moves = 0
while hits < SHIPS:
    x, y = getTarget()
    if board[y][x] == BATTLESHIP:
        print( "You sunk my battleship!" )
        board[y][x] = EMPTY
        hits += 1
    else:
        print( "Miss!" )
    moves += 1
```

```
print( "You needed", moves, "moves to sink all battleships." )
```

Answer 12.8

answer1208.py

```
# Recursive function that determines if intlist, which is a list
# of integers, contains a subset that adds up to total. It
# returns the subset, or empty list if there is no such subset.
def subset_that_adds_up_to( intlist, total ):
    for num in intlist:
        if num == total:
            return [num]
        newlist = intlist[:]
        newlist.remove( num )
        retlist = subset_that_adds_up_to( newlist, total-num )
        if len( retlist ) > 0:
            retlist.insert( 0, num )
            return( retlist )
    return []

numlist = [ 3, 8, -1, 4, -5, 6 ]

solution = subset_that_adds_up_to( numlist, 0 )

if len( solution ) <= 0:
    print( "There is no subset which adds up to zero" )
else:
    for i in range( len( solution ) ):
        if solution[i] < 0 or i == 0:
            print( solution[i], end="" )
        else:
            print( "+{}".format( solution[i] ), end="" )
    print( "=0" )
```

The recursive function works as follows: It loops through all numbers on `intlist`. If the current number is equal to `total`, it has found a solution and returns a list which contains only that number. Otherwise, it recursively calls itself with a copy of `intlist` from which the current number is removed, for a total which is reduced by the current number (e.g., if the current number is 3 and the total is 5, it removes 3 from the list and checks whether a total of 2 can be achieved with the remaining list, because then 5 could be achieved by adding 3 to the solution for the remaining list). If the recursive call returns a list that is not empty, a solution is found, so the current number is appended to the returned list, and the result is returned. Once all numbers have been processed and no solution was found, an empty list is returned.

Note: The solution to this problem tests all possible subsets until one is found that solves the problem, or all of them have been tested. You might wonder whether there are smarter solutions, considering the fact that the number of subset rises exponentially with the size

of the list. The answer is that there might be solutions which improve upon this one (for instance, I could take into account that if a number occurs multiple times on the list that some subsets occur multiple times and need only be tested once), but that for general lists of numbers, no algorithm is known that avoids having to process each and every subset if there is no solution. For those who know some complexity theory: the subset sum problem is "NP-hard."

Chapter 13

Answer 13.1

answer1301.py

```
text = """How much wood would a woodchuck chuck
If a woodchuck could chuck wood?
He would chuck, he would, as much as he could,
And chuck as much as a woodchuck would
If a woodchuck could chuck wood."""

def clean( s ):
    news = ""
    s = s.lower()
    for c in s:
        if c >= "a" and c <= "z":
            news += c
        else:
            news += " "
    return news

worddict = {}
for word in clean( text ).split():
    worddict[word] = worddict.get( word, 0 ) + 1

keylist = list( worddict.keys() )
keylist.sort()
for key in keylist:
    print( "{}: {}".format( key, worddict[key] ) )
```

Answer 13.2

answer1302.py

```
movies = {  "Monty Python and the Holy Grail":
            [ 9, 10, 9.5, 8.5, 3, 7.5,8 ],
            "Monty Python's Life of Brian":
            [ 10, 10, 0, 9, 1, 8, 7.5, 8, 6, 9 ],
            "Monty Python's Meaning of Life":
            [ 7, 6, 5 ],
            "And Now For Something Completely Different":
            [ 6, 5, 6, 6 ] }
```

```
keylist = list( movies.keys() )
keylist.sort()
for key in keylist:
    print( "{}: {}".format( key, round(
        sum( movies[key] )/len( movies[key] ), 1 ) ) )
```

Answer 13.3 Many answers are possible. Probably the simplest one is to use a dictionary where the keys are tuples of writers' last and first names, and the values are lists of all the books of a writer. A book is a tuple consisting of title and location. To find all the books of a writer, use the writer's last and first name to find a list of his books. To find the location of a specific book of a writer, get the list of his books and find the book on that list, then get the corresponding location. You could choose to store the book list also as a dictionary with a book's title as key, but writers usually do not write so many books that you need to do that. Of course, using names as keys is not a great idea as it is easy to make spelling mistakes, and book titles as keys would be even less useful, but these were the only identifying elements I described. I can't complain if you want to introduce easier and less ambiguous keys in such a data structure (but in general, you should divert to a database system to deal with libraries).

Chapter 14

Answer 14.1

answer1401.py

```
allthings = {"Socrates", "Plato", "Eratosthenes", "Zeus", "Hera",
    "Athens", "Acropolis", "Cat", "Dog"}
men = {"Socrates", "Plato", "Eratosthenes"}
mortalthings = {"Socrates","Plato","Eratosthenes","Cat","Dog"}

print( men.issubset( mortalthings ) ) # (a)
print( "Socrates" in men ) # (b)
print( "Socrates" in mortalthings ) # (c)
print( len( mortalthings.difference( men ) ) > 0 ) # (d)
print( len( allthings.difference( mortalthings ) ) > 0 ) # (e)
```

Answer 14.2

answer1402.py

```
set3 = set( [3*x for x in range( 1, int( 1001/3 )+1 )] )
set7 = set( [7*x for x in range( 1, int( 1001/7 )+1 )] )
set11 = set( [11*x for x in range( 1, int( 1001/11 )+1 )] )

seta = set3 & set7 & set11
setb = (set3 & set7) - set11
setc = set( range( 1, 1001 ) ) - set3 - set7 - set11
```

Chapter 15

Answer 15.1

answer1501.py

```
from os import listdir, getcwd

flist = listdir( "." )
for name in flist:
    print( getcwd() + "/" + name )
```

Chapter 16

Answer 16.1 The code below is mostly a copy of some code you had to write in Chapter 13. The only difference is that the text here is not provided as a string, but read from a file.

answer1601.py

```
def clean( s ):
    news = ""
    s = s.lower()
    for c in s:
        if c >= "a" and c <= "z":
            news += c
        else:
            news += " "
    return news

fp = open( "pc_woodchuck.txt" )
text = fp.read()
fp.close()

wdict = {}
for word in clean( text ).split():
    wdict[word] = wdict.get( word, 0 ) + 1

keylist = list( wdict.keys() )
keylist.sort()
for key in keylist:
    print( "{}: {}".format( key, wdict[key] ) )
```

Answer 16.2

answer1602.py

```
def clean( s ):
    news = ""
    s = s.lower()
    for c in s:
```

```
        if c >= "a" and c <= "z":
            news += c
        else:
            news += " "
    return news

wdict = {}
fp = open( "pc_woodchuck.txt" )
while True:
    line = fp.readline()
    if line == "":
        break
    for word in clean( line ).split():
        wdict[word] = wdict.get( word, 0 ) + 1
fp.close()

keylist = list( wdict.keys() )
keylist.sort()
for key in keylist:
    print( "{}: {}".format( key, wdict[key] ) )
```

Answer 16.3

answer1603.py

```
from os.path import join
from os import getcwd

def removevowels( line ):
    newline = ""
    for c in line:
        if c not in "aeiouAEIOU":
            newline += c
    return newline

inputname = join( getcwd(), "pc_woodchuck.txt" )
outputname = join( getcwd(), "pc_woodchuck.tmp" )

fpi = open( inputname )
fpo = open( outputname, "w" )

countread = 0
countwritten = 0

while True:
    line = fpi.readline()
    if line == "":
        break
    countread += len( line )
    line = removevowels( line )
```

```
    fpo.write( line )
    countwritten += len( line )

fpo.close()
fpi.close()

print( "Characters read:", countread )
print( "Characters written:", countwritten )
```

Answer 16.4 The solution below makes use of sets: a set is created for each file, which contains all the words from that file that have 2 letters or more. You can change the length of the words searched for by changing `WORDLEN`. To make the program flexible, it is not limited to just three sets, but uses as many sets as there are names in the file list. At the end, the program creates an intersection of all the sets to determine the words that meet the requirements.

answer1604.py

```
from os.path import join
from os import getcwd

WORDLEN = 2

def clean( s ):
    news = ""
    s = s.lower()
    for c in s:
        if c >= "a" and c <= "z":
            news += c
        else:
            news += " "
    return news

files = ["pc_jabberwocky.txt","pc_rose.txt","pc_woodchuck.txt"]
setlist = []

for name in files:
    filename = join( getcwd(), name )
    wordset = set()
    setlist.append( wordset )
    fp = open( filename )
    while True:
        line = fp.readline()
        if line == "":
            break
        wordlist = clean( line ).split()
        for word in wordlist:
            if len( word ) >= WORDLEN:
                wordset.add( word )
    fp.close()
```

```
combination = setlist[0].copy()
i = 1
while i < len( setlist ):
    combination = combination & setlist[i]
    i += 1
for word in combination:
    print( word )
```

Answer 16.5 Again, in the solution below I chose to be flexible as far as the number of files is concerned. The program is easier to write if you assume there are three files, and no more. If you chose such a solution, that is acceptable as long as the program does what it should do, as allowing it to work with a variable number of files is more an exercise for the chapter on iterations. However, by now you should be quite familiar with iterations, so try to apply them whenever they provide a superior solution.

answer1605.py

```
from os.path import join
from os import getcwd

files = ["pc_jabberwocky.txt","pc_rose.txt","pc_woodchuck.txt"]
letterlist = [ len( files )*[0] for i in range( 26 ) ]
totallist = len( files ) * [0]

# Process all the input files, read their contents line by line,
# make them lower case, and keep track of the letter counts in
# letterlist, while keeping track of total counts in totallist.
filecount = 0
for name in files:
    filename = join( getcwd(), name )
    fp = open( filename )
    while True:
        line = fp.readline()
        if line == "":
            break
        line = line.lower()
        for c in line:
            if c >= 'a' and c <= 'z':
                totallist[filecount] += 1
                letterlist[ord(c)-ord("a")][filecount] += 1
    fp.close()
    filecount += 1

# Write the counts in CSV format.
outfilename = join( getcwd(), "pc_writetest.csv" )
fp = open( outfilename, "w" )
for i in range( len( letterlist ) ):
    s = "\"{}\"".format( chr( ord("a")+i ) )
    for j in range( len( files ) ):
```

```
        s += ",{:.5f}".format( letterlist[i][j]/totallist[j] )
    fp.write( s+"\n" )
fp.close()

# Print the contents of the created output file as a check.
fp = open( outfilename )
print( fp.read() )
fp.close()
```

Chapter 17

Answer 17.1 The code can generate a `ValueError` when you enter something that is not an integer, an `IndexError` when you give an index outside the range `{-5,4}`, a `ZeroDivisionError` when you enter index 2, and a `TypeError` when you enter index 3. The code below does the most straightforward handling, but you can also build a loop around the code so that the user gets asked for new inputs until it works.

answer1701.py

```
numlist = [ 100, 101, 0, "103", 104 ]
try:
    i1 = int( input( "Give an index: " ) )
    print( "100 /", numlist[i1], "=", 100 / numlist[i1] )
except ValueError:
    print( "You did not enter an integer" )
except IndexError:
    print( "You should specify an index between -5 and 4" )
except ZeroDivisionError:
    print( "It looks like the list contains a zero" )
except TypeError:
    print( "it looks like there is a non-numeric item" )
except:
    print( "Something unexpected happened" )
    raise
```

Chapter 18

Answer 18.1 For this program I created a copy of "pc_rose.txt" and called it "pc_rose_copy.txt." To demonstrate what happens, I display the contents of the file before and after the encryption process.

answer1801.py

```
FILENAME = "pc_rose_copy.txt"

def display_contents( filename ):
    fp = open( filename, "rb" )
```

```
    print( fp.read() )
    fp.close()

def encrypt( filename ):
    fp = open( filename, "r+b" )
    buffer = fp.read()
    fp.seek(0)
    for c in buffer:
        if c >= 128:
            fp.write( bytes( [c-128] ) )
        else:
            fp.write( bytes( [c+128] ) )
    fp.close()

display_contents( FILENAME )
encrypt( FILENAME )
display_contents( FILENAME )
```

Answer 18.2

answer1802.py

```
letters = "etaoinshrdlcum "
unencoded = "Hello, world!"

# Print the unencoded string, as a check.
print( unencoded, len( unencoded ) )

# Create a half-byte-list as a basis for the encoding.
halfbytelist = []
for c in unencoded:
    if c in letters:
        halfbytelist.append( letters.index( c )+1 )
    else:
        byte = ord( c )
        halfbytelist.extend( [0, int( byte/16 ), byte%16 ] )
if len( halfbytelist )%2 != 0:
    halfbytelist.append( 0 )

# Turn the half-byte-list into a byte-list.
bytelist = []
for i in range( 0, len( halfbytelist ), 2 ):
    bytelist.append( 16*halfbytelist[i] + halfbytelist[i+1] )

# Turn the byte-list into a byte string and print it, as a check.
encoded = bytes( bytelist )
print( encoded, len( encoded ) )
```

This solution needs 25 lines of code, of which 4 are comments, 4 are empty lines, and 3 are only for testing purposes. So, basically, 14 lines of code suffices. That wasn't too bad, right?

Answer 18.3

answer1803.py

```
letters = "etaoinshrdlcum "
encoded = b'\x04\x81\xbb@,\xf0wI\xba\x02\x10'

# Print the encoded byte string, as a check.
print( encoded, len( encoded ) )

# Create a half-byte-list on the basis of the byte string.
halfbytelist = []
for c in encoded:
    halfbytelist.extend( [ int( c/16 ), c%16 ] )
if halfbytelist[-1] == 0:
    del halfbytelist[-1]

# Turn the half-byte-list into a string.
decoded = ""
while len( halfbytelist ) > 0:
    num = halfbytelist.pop(0)
    if num > 0:
        decoded += letters[num-1]
        continue
    num = 16*halfbytelist.pop(0) + halfbytelist.pop(0)
    decoded += chr( num )

# Print the string, as a check.
print( decoded, len( decoded ) )
```

Answer 18.4 This program looks like it is quite long, but it is straightforward. `compress()` and `decompress()` were developed in the previous two exercises (I changed them so that input and output are byte strings). The rest of the program is mainly the handling of potential errors in dealing with the files. You often see this in programs: the core functionality needs a few lines, while handling of potential problems takes three-quarters of the code.

answer1804.py

```
from pcinput import getString, getLetter
from os.path import exists, getsize

LETTERS = b"etaoinshrdlcum "

# Compress byte string unencoded, return the compressed version.
def compress( unencoded ):
    halfbytelist = []
    for c in unencoded:
        if c in LETTERS:
            halfbytelist.append( LETTERS.index( c )+1 )
        else:
            halfbytelist.extend( [0, int( c/16 ), c%16 ] )
```

```
    if len( halfbytelist )%2 != 0:
        halfbytelist.append( 0 )
    bytelist = []
    for i in range( 0, len( halfbytelist ), 2 ):
        bytelist.append( 16*halfbytelist[i] + halfbytelist[i+1] )
    return bytes( bytelist )

# Decompress byte string encoded, return decompressed version.
def decompress( encoded ):
    halfbytelist = []
    for c in encoded:
        halfbytelist.extend( [ int( c/16 ), c%16 ] )
    if halfbytelist[-1] == 0:
        del halfbytelist[-1]
    bytelist = []
    while len( halfbytelist ) > 0:
        num = halfbytelist.pop(0)
        if num > 0:
            bytelist.append( LETTERS[num-1] )
            continue
        num = 16*halfbytelist.pop(0) + halfbytelist.pop(0)
        bytelist.append( num )
    return bytes( bytelist )

# Ask for the input file and read its contents.
while True:
    filein = getString( "Which is the input file? " )
    if not exists( filein ):
        print( filein, "does not exist" )
        continue
    try:
        fp = open( filein, "rb" )
        buffer = fp.read()
        fp.close()
    except IOError as ex:
        print( filein, "cannot be processed, choose another" )
        print( "Error [{}]: {}".format( ex.args[0], ex.args[1] ))
        continue
    break

# Ask for the output file and create it.
while True:
    fileout = getString( "Which is the output file? " )
    if exists( fileout ):
        print( fileout, "already exists" )
        continue
    try:
        fp = open( fileout, "wb" )
    except IOError as ex:
        print( fileout, "cannot be created,",
```

```
            "choose another file name" )
        print( "Error [{}]: {}".format( ex.args[0], ex.args[1] ))
        continue
    break

# Ask whether the user wants to compress or decompress.
while True:
    dc = getLetter( "Choose (C)ompress or (D)ecompress? " )
    if dc != 'C' and dc != 'D':
        print( "Please choose C or D" )
        continue
    break

# Compress or decompress the buffer.
if dc == 'C':
    buffer = compress( buffer )
else:
    buffer = decompress( buffer )

# Store the (de)compressed buffer in the output file.
try:
    fp.write( buffer )
    fp.close()
except IOError as ex:
    print( "The writing process failed" )
    print( "Error [{}]: {}".format( ex.args[0], ex.args[1] ))

# Report the sizes of input and output.
print( getsize( filein ), "bytes read" )
print( getsize( fileout ), "bytes written" )
```

Chapter 19

Answer 19.1

answer1901.py

```
s = "Hello, world!"
mask = (1<<5) | (1<<3) | (1<<1)    # 00101010

code = ""
for c in s:
    code += chr(ord(c)^mask)
print( code )

decode = ""
for c in code:
    decode += chr(ord(c)^mask)
print( decode )
```

Answer 19.2

answer1902.py

```
def setBit( store, index, value ):
    mask = 1<<index
    if value:
        store |= mask
    else:
        store &= ~mask
    return store

# getBit() returns 0 when the bit corresponding to index is set,
# and something else otherwise. As only 0 is interpreted as False
# this function can be used to test the value of the bit.
def getBit( store, index ):
    mask = 1<<index
    return store & mask

def displayBits( store ):
    for i in range( 8 ):
        index = 7 - i
        if getBit( store, index ):
            print( "1", end="" )
        else:
            print( "0", end="" )
    print()

store = 0
store = setBit( store, 0, True )
store = setBit( store, 1, True )
store = setBit( store, 2, False )
store = setBit( store, 3, True )
store = setBit( store, 4, False )
store = setBit( store, 5, True )
displayBits( store )

store = setBit( store, 1, False )
displayBits( store )
```

Chapter 20

Answer 20.1

answer2001.py

```
from copy import copy

class Point:
    def __init__( self, x=0.0, y=0.0 ):
```

```
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

class Rectangle:
    def __init__( self, point, width, height ):
        self.point = copy( point )
        self.width = abs( width )
        self.height = abs( height )
        if self.width == 0:
            self.width = 1
        if self.height == 0:
            self.height = 1
    def __repr__( self ):
        return "[{},w={},h={}]".format( self.point,
            self.width, self.height )
    def surface_area( self ):
        return self.width * self.height
    def circumference( self ):
        return 2*(self.width + self.height)
    def bottom_right( self ):
        return Point( self.point.x + self.width,
            self.point.y + self.height )
    def overlap( self,r ):
        r1, r2 = self, r
        if self.point.x > r.point.x or \
            (self.point.x == r.point.x and \
            self.point.y > r.point.y):
            r1, r2 = r, self
        if r1.bottom_right().x <= r2.point.x or \
            r1.bottom_right().y <= r2.point.y:
            return None
        return Rectangle( r2.point,
            min( r1.bottom_right().x - r2.point.x, r2.width ),
            min( r1.bottom_right().y - r2.point.y, r2.height ) )

r1 = Rectangle( Point( 1, 1 ), 8, 5 )
r2 = Rectangle( Point( 2, 3 ), 9, 2 )

print( r1.surface_area() )
print( r1.circumference() )
print( r1.bottom_right() )
r = r1.overlap( r2 )
if r:
    print( r )
else:
    print( "There is no overlap for the rectangles" )
```

Answer 20.2 Considering the list that must be displayed, I placed the `enroll()` method in `Student`. For the birth date I use the `datetime` module; as you have to calculate the student's age, you also need today's date, which is found in the `datetime` module anyway.

answer2002.py

```
from datetime import date
from random import random

class Course:
    def __init__( self, name, number ):
        self.name = name
        self.number = number
    def __repr__( self ):
        return "{}: {}".format( self.number, self.name )

class Student:
    def __init__( self, lastname, firstname, birthdate, anr ):
        self.lastname = lastname
        self.firstname = firstname
        self.birthdate = birthdate
        self.anr = anr
        self.courses = []
    def __str__( self ):
        return self.firstname+" "+self.lastname
    def age( self ):
        today = date.today()
        years = today.year - self.birthdate.year
        if today.month < self.birthdate.month or \
            (today.month == self.birthdate.month \
            and today.day < self.birthdate.day):
            years -= 1
        return years
    def enroll( self, course ):
        if course not in self.courses:
            self.courses.append( course )

students = [
    Student( "Arkansas", "Adrian", date( 1989, 10, 3 ), 453211 ),
    Student( "Bonzo", "Beatrice", date( 1991, 12, 29 ), 476239 ),
    Student( "Continuum", "Carola", date( 1992, 3, 7 ), 784322 ),
    Student( "Doofus", "Dunce", date( 1993, 7, 11 ), 995544 ) ]
courses =[
    Course( "Vinology", 787656 ),
    Course( "Advanced spoon-bending", 651121 ),
    Course( "Research Skills: Babbling", 433231 )]

for student in students:
    for course in courses:
        if random() > 0.3:
            student.enroll( course )
```

```
for student in students:
    print( "{}: {} {} ({})".format( student.anr,
        student.firstname, student.lastname, student.age() ) )
    if len( student.courses ) == 0:
        print( "\tNo courses" )
    for course in student.courses:
        print( "\t{}".format( course ) )
```

Chapter 21

Answer 21.1

answer2101.py

```
SUITS = ["Hearts","Spades","Clubs","Diamonds"]
RANKS = ["2","3","4","5","6","7","8","9","10",
    "Jack","Queen","King","Ace"]

class Card:
    def __init__( self, suit, rank ):
        self.suit = suit # used as index in the SUITS list
        self.rank = rank # used as index in the RANKS list
    def __repr__( self ):
        return "({},{})".format( self.suit, self.rank )
    def __str__( self ):
        return "{} of {}".format( RANKS[self.rank], \
            SUITS[self.suit] )
    def __eq__( self, c ):
        if isinstance( c, Card ):
            return self.rank == c.rank
        return NotImplemented
    def __gt__( self, c ):
        if isinstance( c, Card ):
            return self.rank > c.rank
        return NotImplemented
    def __ge__( self, c ):
        if isinstance( c, Card ):
            return self.rank >= c.rank
        return NotImplemented

c5 = Card( 2, 3 )
d5 = Card( 3, 3 )
sk = Card( 1, 11 )
print( "{}, {}, {}".format( c5, d5, sk ) )
print( c5 == d5 )
print( c5 == sk )
print( c5 > sk )
print( c5 >= sk )
```

```
print( c5 < sk )
print( c5 <= sk )
```

Note: There is no need to implement `__ne__()`, `__lt__()`, or `__le__()`, as they are automatically changed into calls to the methods that have been implemented.

Answer 21.2 For readability, I deleted the methods from `Card` that are not needed here.

answer2102.py

```
SUITS = ["Hearts","Spades","Clubs","Diamonds"]
RANKS = ["2","3","4","5","6","7","8","9","10",
    "Jack","Queen","King","Ace"]

class Card:
    def __init__( self, suit, rank ):
        self.suit = suit
        self.rank = rank
    def __str__( self ):
        return "{} of {}".format(
            RANKS[self.rank], SUITS[self.suit] )

class Drawpile:
    def __init__( self, pile=[] ):
        self.pile = pile
    def __len__( self ):
        return len( self.pile )
    def __getitem__( self, n ):
        return self.pile[n]
    def add( self, c ):
        self.pile.append( c )
    def draw( self ):
        if len( self ) <= 0:
            return None
        return self.pile.pop(0)
    def __repr__( self ):
        sep = ""
        s = ""
        for c in self.pile:
            s += sep + str( c )
            sep = ", "
        return s

dp1 = Drawpile( [Card(0,1), Card(0,5), Card(2,4), Card(1,12)] )
print( dp1 )
print( dp1[1] )
dp1.add( Card(3,12) )
print( dp1 )
print( dp1.draw() )
print( dp1 )
```

Answer 21.3

answer2103.py

```
SUITS = ["Hearts","Spades","Clubs","Diamonds"]
RANKS = ["2","3","4","5","6","7","8","9","10",
    "Jack","Queen","King","Ace"]

class Card:
    def __init__( self, suit, rank ):
        self.suit = suit
        self.rank = rank
    def __repr__( self ):
        return "({},{})".format( self.suit, self.rank )
    def __str__( self ):
        return "{} of {}".format(
            RANKS[self.rank], SUITS[self.suit] )
    def __eq__( self, c ):
        if isinstance( c, Card ):
            return self.rank == c.rank
        return NotImplemented
    def __gt__( self, c ):
        if isinstance( c, Card ):
            return self.rank > c.rank
        return NotImplemented
    def __ge__( self, c ):
        if isinstance( c, Card ):
            return self.rank >= c.rank
        return NotImplemented

class Drawpile:
    def __init__( self, pile=[] ):
        self.pile = pile
    def __len__( self ):
        return len( self.pile )
    def __getitem__( self, n ):
        return self.pile[n]
    def add( self, c ):
        self.pile.append( c )
    def draw( self ):
        if len( self ) <= 0:
            return None
        return self.pile.pop(0)
    def __repr__( self ):
        sep = ""
        s = ""
        for c in self.pile:
            s += sep + str( c )
            sep = ", "
        return s
```

```
dp1 = Drawpile( [Card(3,0), Card(0,11), Card(2,5)] )
dp2 = Drawpile( [Card(3,2), Card(3,1), Card(1,6)] )

i = 1
while len( dp1 ) > 0 and len( dp2 ) > 0:
    print( "Round", i )
    print( "Deck1:", dp1 )
    print( "Deck2:", dp2 )
    c1 = dp1.draw()
    c2 = dp2.draw()
    if c1 > c2:
        dp1.add( c1 )
        dp1.add( c2 )
    else:
        dp2.add( c2 )
        dp2.add( c1 )
    i += 1

print( "The game has ended" )
if len( dp1 ) > 0:
    print( "Deck1:", dp1 )
    print( "The first deck wins!" )
else:
    print( "Deck2:", dp2 )
    print( "The second deck wins!" )
```

Answer 21.4 Do not forget that in the `__add__()` and `__sub__()` methods you have to create a (deep) copy of the fruit basket, which you change and return. If you would change the `fruitbasket` itself instead of a deep copy, `newbasket = fruitbasket + "apple"` would make `newbasket` an alias for `fruitbasket`. It is unlikely that the programmer who uses the class wants that. It is up to the programmer of the main program to decide whether he wants to actually change the fruit basket, or just wants to see what a changed version looks like.

However, in `__iadd__()` and `__isub__()` you are actually supposed to change the fruit basket, and still return `self`. Because a statement like `fruitbasket += "apple"` clearly is intended to change `fruitbasket`.

The rest of the class is pretty straightforward, as long as you take into account that you have to delete fruits when their number has dropped to zero or less.

answer2104.py

```
from copy import deepcopy

class FruitBasket:

    def __init__( self, fruits={} ):
        self.fruits = fruits

    def __repr__( self ):
```

```
        s = ""
        sep = "["
        for fruit in self.fruits:
            s += sep + fruit + ":" + str( self.fruits[fruit] )
            sep = ", "
        s += "]"
        return s

    def __contains__( self, fruit ):
        return fruit in self.fruits

    def __add__( self, fruit ):
        fbcopy = deepcopy( self )
        fbcopy.fruits[fruit] = fbcopy.fruits.get( fruit, 0 ) + 1
        return fbcopy

    def __iadd__( self, fruit ):
        self.fruits[fruit] = self.fruits.get( fruit, 0 ) + 1
        return self

    def __sub__( self, fruit ):
        if fruit not in self.fruits:
            return self
        fbcopy = deepcopy( self )
        fbcopy.fruits[fruit] = fbcopy.fruits.get( fruit, 0 ) - 1
        if fbcopy.fruits[fruit] <= 0:
            del fbcopy.fruits[fruit]
        return fbcopy

    def __isub__( self, fruit ):
        self.fruits[fruit] = self.fruits.get( fruit, 0 ) - 1
        if self.fruits[fruit] <= 0:
            del self.fruits[fruit]
        return self

    def __len__( self ):
        return len( self.fruits )

    def __getitem__( self, fruit ):
        return self.fruits.get( fruit, 0 )

    def __setitem__( self, fruit, n ):
        if n <= 0:
            if fruit in self.fruits:
                del self.fruits[fruit]
        else:
            self.fruits[fruit] = n

fb = FruitBasket()
fb += "apple"
```

```
fb += "apple"
fb += "banana"
fb = fb + "cherry"
fb["orange"] = 20
print( len( fb ) )
print( fb )
print( "banana" in fb )
print( "durian" in fb )
fb -= "apple"
fb -= "banana"
fb = fb - "cherry"
fb -= "durian"
print( fb )
print( "banana" in fb )
fb["orange"] = 0
print( fb )
```

Chapter 22

Answer 22.1

answer2201.py

```
class Rectangle:
    def __init__( self, x, y, w, h ):
        self.x, self.y, self.w, self.h = x, y, w, h
    def __repr__( self ):
        return "[({},{}),w={},h={}]".format( self.x, self.y,
          self.w, self.h )
    def area( self ):
        return self.w * self.h
    def circumference( self ):
        return 2*(self.w + self.h)

class Square( Rectangle ):
    def __init__( self, x, y, w ):
        super().__init__( x, y, w, w )

s = Square( 1, 1, 4 )
print( s, s.area(), s.circumference() )
```

Answer 22.2

answer2202.py

```
from math import pi

class Shape:
    def area( self ):
```

```
        return NotImplemented
    def circumference( self ):
        return NotImplemented

class Circle( Shape ):
    def __init__( self, x, y, r ):
        self.x, self.y, self.r = x, y, r
    def __repr__( self ):
        return "[({},{}),r={}]".format( self.x, self.y, self.r )
    def area( self ):
        return pi * self.r * self.r
    def circumference( self ):
        return 2 * pi * self.r

class Rectangle( Shape ):
    def __init__( self, x, y, w, h ):
        self.x, self.y, self.w, self.h = x, y, w, h
    def __repr__( self ):
        return "[({},{}),w={},h={}]".format( self.x, self.y,
            self.w, self.h )
    def area( self ):
        return self.w * self.h
    def circumference( self ):
        return 2*(self.w + self.h)

class Square( Rectangle ):
    def __init__( self, x, y, w ):
        super().__init__( x, y, w, w )

s = Square( 1, 1, 4 )
print( s, s.area(), s.circumference() )
c = Circle( 1, 1, 4 )
print( c, c.area(), c.circumference() )
```

Answer 22.3 I implemented a `MemoryStrategy` class to derive `TitForTat`, `TitForTwoTats`, and `Majority` from. `MemoryStrategy` keeps track of all the moves in the game. That is overdoing it a bit, as `TitForTat` only needs to keep track of the last move, `TitForTwoTats` only needs to keep track of the last two moves, and `Majority` could make do with just a count of all the COOPERATEs and DEFECTs of the opponent. Still, if more elaborate strategies need to be implemented, `MemoryStrategy` is a good starting point.

One interesting thing to notice is that in any pairing `AlwaysDefect` has a higher score than its opponent. However, if you let each strategy play every other strategy and add up the scores, the only strategy that does worse than `AlwaysDefect` is `Random`. If you want to know why that is, take a course on Game Theory.

answer2203.py

```
from random import random

COOPERATE = 'C'
```

```
DEFECT = 'D'
ROUNDS = 100

class Strategy:
    def __init__( self, name="" ):
        self.name = name
        self.score = 0
    def choice( self ):
        # Should return COOPERATE or DEFECT
        return NotImplemented
    def lastmove( self, mymove, opponentmove ):
        # Gets passed the last move made, after call to choice()
        pass
    def incscore( self, n ):
        self.score += n

class AlwaysDefect( Strategy ):
    def choice( self ):
        return DEFECT

class Random( Strategy ):
    def choice( self ):
        if random() >= 0.5:
            return COOPERATE
        return DEFECT

class MemoryStrategy( Strategy ):
    def __init__( self, name="" ):
        super().__init__( name )
        self.history = []
    def lastmove( self, mymove, opponentmove ):
        self.history.append( (mymove, opponentmove) )

class TitForTat( MemoryStrategy ):
    def choice( self ):
        if len( self.history ) < 1:
            return COOPERATE
        return self.history[-1][1]

class TitForTwoTats( MemoryStrategy ):
    def choice( self ):
        if len( self.history ) < 2:
            return COOPERATE
        if self.history[-1][1] == DEFECT and \
            self.history[-2][1] == DEFECT:
            return DEFECT
        return COOPERATE

class Majority( MemoryStrategy ):
    def choice( self ):
```

```
        countD = 0
        for i in range( len( self.history ) ):
            if self.history[i][1] == DEFECT:
                countD += 1
        if countD > len( self.history ) / 2:
            return DEFECT
        return COOPERATE

strategy1 = AlwaysDefect( "Always Defect" )
strategy2 = Majority( "Majority" )

for i in range( ROUNDS ):
    c1 = strategy1.choice()
    c2 = strategy2.choice()
    if c1 == c2:
        strategy1.incscore( 3 if c1 == COOPERATE else 1 )
        strategy2.incscore( 3 if c2 == COOPERATE else 1 )
    else:
        strategy1.incscore( 0 if c1 == COOPERATE else 6 )
        strategy2.incscore( 0 if c2 == COOPERATE else 6 )
    strategy1.lastmove( c1, c2 )
    strategy2.lastmove( c2, c1 )

print( "Score of", strategy1.name, "is", strategy1.score )
print( "Score of", strategy2.name, "is", strategy2.score )
```

Chapter 23

Answer 23.1

answer2301.py

```
from pcinput import getInteger

class NotDividableBy:
    def __init__( self ):
        self.seq = list( range( 1, 101 ) )
    def __iter__( self ):
        return self
    def __next__( self ):
        if len( self.seq ) > 0:
            return self.seq.pop(0)
        raise StopIteration()
    def process( self, num ):
        i = len( self.seq )-1
        while i >= 0:
            if self.seq[i]%num == 0:
                del self.seq[i]
            i -= 1
```

```
    def __len__( self ):
        return len( self.seq )

ndb = NotDividableBy()
while True:
    num = getInteger( "Give an integer: " )
    if num < 0:
        print( "Negative integers are ignored" )
        continue
    if num == 0:
        break
    ndb.process( num )

if len( ndb ) <= 0:
    print( "No numbers are left" )
else:
    for num in ndb:
        print( num, end=" " )
```

Answer 23.2

answer2302.py

```
def factorial():
    total = 1
    for i in range( 1, 11 ):
        total *= i
        yield total

fseq = factorial()
for n in fseq:
    print( n, end=" " )
```

Answer 23.3

answer2303.py

```
from itertools import permutations

word = input( "Please enter a word: " )
seq = permutations( word )
for item in seq:
    print( "".join( item ) )
```

Answer 23.4 The only change I made with respect to the previous answer is that I cast the iterable to a set.

answer2304.py

```
from itertools import permutations
```

```
word = input( "Please enter a word: " )
seq = permutations( word )
for item in set( seq ):
    print( "".join( item ) )
```

Answer 23.5

answer2305.py

```
from itertools import combinations

numlist = [ 3, 8, -1, 4, -5, 6 ]
solution = []

for i in range( 1, len( numlist )+1 ):
    seq = combinations( numlist, i )
    for item in seq:
        if sum( item ) == 0:
            solution = item
            break
    if len( solution ) > 0:
        break

if len( solution ) <= 0:
    print( "There is no subset which adds up to zero" )
else:
    for i in range( len( solution ) ):
        if solution[i] < 0 or i == 0:
            print( solution[i], end="" )
        else:
            print( "+{}".format( solution[i] ), end="" )
    print( "=0" )
```

Note: While this code seems to create all possible subsets, which is a number that rises exponentially with the size of `numlist`, since iterators are used no more than one subset is in memory at any time. So this solution works fine even for large lists of integers (though it may get slow, which cannot be helped as this is an NP-hard problem).

Answer 23.6

answer2306.py

```
from itertools import combinations

testdict = {"a":1, "b":2, "c":3, "d":4 }
result = [ {} ]

keylist = list( testdict.keys() )
for length in range( 1, len( testdict)+1 ):
```

```
    for item in combinations( keylist, length ):
        subdict = {}
        for key in item:
            subdict[key] = testdict[key]
        result.append( subdict )

print( result )
```

Answer 23.7 If rows and columns are numbered 0 to 7, then every row and column number will occur exactly once in the solution. Take a list of the eight possible column numbers, and consider the index in the list as the row number. All potential solutions are represented by permutations of this list. You only need to check these permutations until you find one where never a diagonal is shared by two of the queens, i.e., where the absolute difference between the row numbers of any two queens is equal to the absolute difference of their column numbers.

answer2307.py

```
from itertools import permutations

SIZE = 8 # Board size.

def display_board( columns ):
    for i in columns :
        for j in range( len( columns ) ):
            if i == j:
                print( 'Q', end=" " )
            else:
                print( '-', end=" " )
        print()

def is_solution( columns ):
    for row in range( len( columns ) ):
        col = columns[row]
        for i in range( row+1, len( columns ) ):
            if i - row == abs( columns[i] - col ):
                return False
    return True

columns = list( range( SIZE ) )

for p in permutations( columns ):
    if is_solution( p ):
        display_board( p )
        break
else:
    print( "No solutions found" ) # Should not happen.
```

Chapter 24

Answer 24.1

answer2401.py

```
import sys

total = 0
for i in sys.argv[1:]:
    try:
        total += float( sys.argv[i] )
    except TypeError:
        print( sys.argv[i], "is not a number." )
        sys.exit(1)

print( "The arguments add up to", total )
```

Chapter 25

Answer 25.1

answer2501.py

```
import re

sentence = "The price of a 2-room apartment in Manhattan \
starts at 1 million dollars, and may actually be the 10-fold \
of that on 42nd Street."

pword = re.compile( r"[A-Za-z]+" )
wordlist = pword.findall( sentence )
for word in wordlist:
    print( word )
```

Answer 25.2

answer2502.py

```
import re

sentence = "The word ether can be found in my thesaurus \
using the archaic spelling 'aether'."

pthe = re.compile( r"\bthe\b", re.I )
thelist = pthe.findall( sentence )
print( len( thelist ) )
```

Answer 25.3

answer2503.py

```
import re

sentence = "Michael Jordan, Bill Gates, and the Dalai Lama \
decided to take a plane trip together."

pname = re.compile( r"\b([A-Z][a-z]*\s+[A-Z][a-z]*)\b" )
namelist = pname.findall( sentence )
for name in namelist:
    print( name )
```

Answer 25.4

answer2504.py

```
import re

sentence = "William Randolph Hearst attempted to destroy all \
copies of Orson Welles' masterpiece 'Citizen Kane', because he \
did not appreciate the fact that the protagonist Charles \
Foster Kane was a thinly disguised caricature of himself. I \
wonder whether William Henry Gates The Third would attempt to \
do the same."

pname = re.compile( r"\b([A-Z][a-z]*(\s+[A-Z][a-z]*)+)\b" )
namelist = pname.finditer( sentence )
for name in namelist:
    print( name.group(1) )
```

Answer 25.5

answer2505.py

```
import re

sentence = "Client: \"I wish to register a complaint! \
Hello miss!\"\n\
Shopkeeper: \"What do you mean, miss?\"\n\
Client: \"I am sorry, I have a cold.\"\n"

pspoken = re.compile( r"\"[^\"]*\"" )
spokenlist = pspoken.findall( sentence )
for text in spokenlist:
    print( text )
```

Answer 25.6

answer2506.py

```
import re

text = "<html><head><title>List of persons with ids</title>\
</head><body>\
<p><id>123123123</id><name>Groucho Marx</name>\
<p><id>123123124</id><name>Harpo Marx</name>\
<p><id>123123125</id><name>Chico Marx</name>\
<randomcrap>Etaoin<id>Shrdlu</id>qwerty</name></randomcrap>\
<nocrap><p><id>123123126</id><name>Zeppo Marx</name></nocrap>\
<address>Chicago</address>\
<morerandomcrap><id>999999999</id>nonametobeseen!\
</morerandomcrap>\
<p><id>123123127</id><name>Gummo Marx</name>\
<note>Look him up on <a href=\"http://www.google.com\">\
Google.</a></note>\
</body></html>"

pidname = re.compile( r"<id>([^<]+)</id><name>([^<]+)</name>" )
mlist = pidname.finditer( text )
for m in mlist:
    print( m.group(1), m.group(2) )
```

Chapter 26

Answer 26.1

answer2601.py

```
from csv import reader, writer

fp = open( "pc_inventory.csv", newline='' )
fpo = open( "pc_writetest.csv", "w", newline='' )
csvreader = reader( fp )
csvwriter = writer( fpo, delimiter=' ', quotechar="'" )
for line in csvreader:
    csvwriter.writerow( line )
fp.close()
fpo.close()

fp = open( "pc_writetest.csv" )
print( fp.read() )
fp.close()
```

If you did it correctly, you notice the quotes around “Blue Stilton,” which are there because it contains a space, which is the delimiter.

Answer 26.2

answer2602.py

```
from csv import reader
from json import dump

data = []

fp = open( "pc_inventory.csv", newline='' )
csvreader = reader( fp )
for line in csvreader:
    data.append( line )
fp.close()

fp = open( "pc_writetest.json", "w" )
dump( data, fp )
fp.close()

fp = open( "pc_writetest.json" )
print( fp.read() )
fp.close()
```

Chapter 27

Answer 27.1

answer2701.py

```
from collections import Counter

sentence = "Your mother was a hamster and \
your father smelled of elderberries."
sentence2 = ""
for c in sentence.lower():
    if c >= 'a' and c <= 'z':
        sentence2 += c

clist = Counter( sentence2 ).most_common( 5 )
for c in clist:
    print( "{}: {}".format( c[0], c[1] ) )
```

Answer 27.2

answer2702.py

```
from collections import Counter
from pcinput import getInteger
from statistics import mean, median
from sys import exit
```

```
numlist = []
while True:
    num = getInteger( "Enter a number: " )
    if num == 0:
        break
    numlist.append( num )

if len( numlist ) <= 0:
    print( "No numbers were entered" )
    exit()

print( "Mean:", mean( numlist ) )
print( "Median:", median( numlist ) )

clist = Counter( numlist ).most_common()
if clist[0][1] <= 1:
    print( "There is no mode" )
else:
    mlist = [str( x[0] ) for x in clist if x[1] == clist[0][1] ]
    s = ", ".join( mlist )
    print( "Mode:", s )
```

For the mode I did a reasonably smart list comprehension, but you can write this out as multiple lines of code, of course.

Index