How to Think Like a Computer Scientist (Volume One)

Learning with Python
UKZN Local Version
2016

Brad Miller, David Ranum, Jeffrey Elkner, Peter Wentworth, Allen B. Downey, Chris Meyers, Dario Mitchell and Anban Pillay.

How To Think Like a Computer Scientist: Learning Python (Volume One)

Anban Pillay and others

2016

Contents

Frontmatter

Copyright Notice

Preface to the Interactive Edition

by Brad Miller and David Ranum

There are many books available if you want to learn to program in Python. In fact, we have written a few of them ourselves. However, none of the books are like this one.

Programming is not a "spectator sport". It is something you do, something you participate in. It would make sense, then, that the book you use to learn programming should allow you to be active. That is our goal.

This book is meant to provide you with an interactive experience as you learn to program in Python. You can read the text, watch videos, and write and execute Python code. In addition to simply executing code, there is a unique feature

called 'codelens' that allows you to control the flow of execution in order to gain a better understanding of how the program works..

We have tried to use these different presentation techniques where they are most appropriate. In other words, sometimes it might be best to read a description of some aspect of Python. On a different occasion, is might be best to execute a small example program. Often we give you many different options for covering the material. Our hope is that you will find that your understanding will be enhanced because you are able to experience it in more than just one way.

Acknowledgements

This interactive textbook is a triumph of open source. Not only have we stood on the shoulders of Jeffrey Elkner et. al. as the starting point for the prose but the interactive features of this book also make use of open source software. We are indebted to Scott Graham (skulpt.org) for his open source Python interpreter written in Javascript. In addition we are indebted to Philip Guo (`https://github.com/pgbovine/OnlinePythonTutor/`) for the Online Python Tutor which forms the basis for Codelens.

We would also like to acknowledge the Sphinx project for their fine work in creating a documentation system that allows us to focus on writing not endless hours of Javascript and html coding. In particular without the extension architecture of Sphinx this book would not have made it off the ground.

Foreword

By David Beazley

As an educator, researcher, and book author, I am delighted to see the completion of this book. Python is a fun and extremely easy-to-use programming language that has steadily gained in popularity over the last few years. Developed over ten years ago by Guido van Rossum, Python's simple syntax and overall feel is largely derived from ABC, a teaching language that was developed in the 1980's. However, Python was also created to solve real problems and it borrows a wide variety of features from programming languages such as C++, Java, Modula-3, and Scheme. Because of this, one of Python's most remarkable features is its broad appeal to professional software developers, scientists, researchers, artists, and educators.

Despite Python's appeal to many different communities, you may still wonder why Python? or why teach programming with Python? Answering these questions is no simple task—especially when popular opinion is on the side of more masochistic alternatives such as C++ and Java. However, I think the most

direct answer is that programming in Python is simply a lot of fun and more productive.

When I teach computer science courses, I want to cover important concepts in addition to making the material interesting and engaging to students. Unfortunately, there is a tendency for introductory programming courses to focus far too much attention on mathematical abstraction and for students to become frustrated with annoying problems related to low-level details of syntax, compilation, and the enforcement of seemingly arcane rules. Although such abstraction and formalism is important to professional software engineers and students who plan to continue their study of computer science, taking such an approach in an introductory course mostly succeeds in making computer science boring. When I teach a course, I don't want to have a room of uninspired students. I would much rather see them trying to solve interesting problems by exploring different ideas, taking unconventional approaches, breaking the rules, and learning from their mistakes. In doing so, I don't want to waste half of the semester trying to sort out obscure syntax problems, unintelligible compiler error messages, or the several hundred ways that a program might generate a general protection fault.

One of the reasons why I like Python is that it provides a really nice balance between the practical and the conceptual. Since Python is interpreted, beginners can pick up the language and start doing neat things almost immediately without getting lost in the problems of compilation and linking. Furthermore, Python comes with a large library of modules that can be used to do all sorts of tasks ranging from web-programming to graphics. Having such a practical focus is a great way to engage students and it allows them to complete significant projects. However, Python can also serve as an excellent foundation for introducing important computer science concepts. Since Python fully supports procedures and classes, students can be gradually introduced to topics such as procedural abstraction, data structures, and object-oriented programming — all of which are applicable to later courses on Java or C++. Python even borrows a number of features from functional programming languages and can be used to introduce concepts that would be covered in more detail in courses on Scheme and Lisp.

In reading Jeffrey's preface, I am struck by his comments that Python allowed him to see a higher level of success and a lower level of frustration and that he was able to move faster with better results. Although these comments refer to his introductory course, I sometimes use Python for these exact same reasons in advanced graduate level computer science courses at the University of Chicago. In these courses, I am constantly faced with the daunting task of covering a lot of difficult course material in a blistering nine week quarter. Although it is certainly possible for me to inflict a lot of pain and suffering by using a language like C++, I have often found this approach to be counterproductive—especially when the course is about a topic unrelated to just programming. I find that using Python allows me to better focus on the actual topic at hand while allowing students to

complete substantial class projects.

Although Python is still a young and evolving language, I believe that it has a bright future in education. This book is an important step in that direction. David Beazley University of Chicago Author of the *Python Essential Reference*

Contributor List

To paraphrase the philosophy of the Free Software Foundation, this book is free like free speech, but not necessarily free like free pizza. It came about because of a collaboration that would not have been possible without the GNU Free Documentation License. So we would like to thank the Free Software Foundation for developing this license and, of course, making it available to us.

We would also like to thank the more than 100 sharp-eyed and thoughtful readers who have sent us suggestions and corrections over the past few years. In the spirit of free software, we decided to express our gratitude in the form of a contributor list. Unfortunately, this list is not complete, but we are doing our best to keep it up to date. It was also getting too large to include everyone who sends in a typo or two. You have our gratitude, and you have the personal satisfaction of making a book you found useful better for you and everyone else who uses it. New additions to the list for the 2nd edition will be those who have made on-going contributions.

If you have a chance to look through the list, you should realize that each person here has spared you and all subsequent readers from the confusion of a technical error or a less-than-transparent explanation, just by sending us a note.

Impossible as it may seem after so many corrections, there may still be errors in this book. If you should stumble across one, we hope you will take a minute to contact us. The email address is `jeff@elkner.net` . Substantial changes made due to your suggestions will add you to the next version of the contributor list (unless you ask to be omitted). Thank you!

Second Edition

- An email from Mike MacHenry set me straight on tail recursion. He not only pointed out an error in the presentation, but suggested how to correct it.
- It wasn't until 5th Grade student Owen Davies came to me in a Saturday morning Python enrichment class and said he wanted to write the card game, Gin Rummy, in Python that I finally knew what I wanted to use as the case study for the object oriented programming chapters.
- A *special* thanks to pioneering students in Jeff's Python Programming class at GCTAA during the 2009-2010 school year: Safath Ahmed, Howard

Batiste, Louis Elkner-Alfaro, and Rachel Hancock. Your continual and thoughtfull feedback led to changes in most of the chapters of the book. You set the standard for the active and engaged learners that will help make the new Governor's Academy what it is to become. Thanks to you this is truly a *student tested* text.

- Thanks in a similar vein to the students in Jeff's Computer Science class at the HB-Woodlawn program during the 2007-2008 school year: James Crowley, Joshua Eddy, Eric Larson, Brian McGrail, and Iliana Vazuka.
- Ammar Nabulsi sent in numerous corrections from Chapters 1 and 2.
- Aldric Giacomoni pointed out an error in our definition of the Fibonacci sequence in Chapter 5.
- Roger Sperberg sent in several spelling corrections and pointed out a twisted piece of logic in Chapter 3.
- Adele Goldberg sat down with Jeff at PyCon 2007 and gave him a list of suggestions and corrections from throughout the book.
- Ben Bruno sent in corrections for chapters 4, 5, 6, and 7.
- Carl LaCombe pointed out that we incorrectly used the term commutative in chapter 6 where symmetric was the correct term.
- Alessandro Montanile sent in corrections for errors in the code examples and text in chapters 3, 12, 15, 17, 18, 19, and 20.
- Emanuele Rusconi found errors in chapters 4, 8, and 15.
- Michael Vogt reported an indentation error in an example in chapter 6, and sent in a suggestion for improving the clarity of the shell vs. script section in chapter 1.

First Edition

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote horsebet.py, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.

- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken catTwice function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word unconsciously in Chapter 1 needed to be changed to subconsciously .
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the increment function in Chapter 13.
- John Ouzts corrected the definition of return value in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the printTime function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have make numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between *gleich* and *selbe*.

- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.
- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and is helping us prepare to update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.

Chapter

1 Introduction

If you give someone a program, you will frustrate them for a day; if you teach them how to program, you will frustrate them for a lifetime.

- David Leinweber

First, solve the problem. Then, write the code.

— John Johnson

Figure 1.1: *al-Kwārizmī. The word algorithm originates from the name of this Persian Mathematician.*

1.1 Algorithms

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about

al-Kwārizmī was a Persian mathematician. In the 12th century, his work introduced the decimal positional number system to the Western world. He presented the first systematic solution of linear and quadratic equations in Arabic.

solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. The idea of an *algorithm* is central to programming [1]. Programmers devise solutions to problems and express these solutions with algorithms. These algorithms can be executed by computers if they are written in programming languages. The word **algorithm**

originates from the name of a Persian mathematician al-Kwārizmī.

GCD is also known as HCF: Highest Common Factor.

Consider the algorithm to find the *greatest common divisor* or **GCD** of two numbers . The algorithm is known as Euclid's algorithm and is named after the Greek mathematician Euclid. The problem may be stated as follows:

> Given two postive integers m and n, find their *greatest common divisor*, i.e. the largest postive integer that divides both m and n without remainder.

For example the GCD of 12 and 8 is 4. Euclid's algorithm to do this is:

```
1: Divide m by n and store the remainder in r (r = remainder).
2: If r == 0, n is the answer and the algorithm terminates.
3: Let m = n and n = r and go back to Step 1.
```

Note that the = means that we replace whatever is in the variable on the left hand side with whatever is on the right hand side. This replacement operation is also known as **assignment**. Do not confuse == which means "is equal to". *r==0* is asking if r is equal to 0.

To find the GCD of 12 and 8 using the algorithm we proceed as follows:

```
1. Divide 12 by 8 and the remainder, r , is 4.
3. m is now 8 and n is now 4
1. Divide 8 by 4 and the remainder is now 0
2. Since r equals 0, the answer is 4 and algorithm terminates.
```

What is an Algorithm?

We build up to answering this question by first making some important observations about the Euclid's algorithm.

There are two styles of programming as well. Some programming languages are declarative e.g. functional programming languages while others are procedural e.g. Python

1. Algorithms are procedures.

   We can make the distinction between two kinds of knowledge. The first kind we call declarative. This kind of knowledge tells us what **is**. For example we can say that the GCD of two numbers is the largest positive integer that divides both. This tells us what the GCD is but not how to calculate

[1] The discussion on algorithms is based on that given in Donald Knuth's book: The Art of Computer Programming-Volume One.

it. The other kind of knowledge we shall call procedural knowledge. This kind of knowledge tells us **how-to** do something. The GCD algorithm above captures procedural knowledge about how to calculate the GCD

2. Algorithms are executed

   An algorithm consists of various steps or instructions that are meant to be followed one after another. Algorithms can be executed by humans or computers. The human or computer must be able to carry out each of the instructions (in the order specified). Euclid's algorithm can be executed if the human or computer knows how to divide one number by another and how to calculate remainders etc. Thus, when writing algorithms, it is important to understand the capabilities of the human or machine that will execute it.

   Computers have a set of basic built-in instructions.

3. Algorithms are built from a set of simple operations

   When designing algorithms we assume that there exists a set of basic operations and that each operation can be carried out by the computer or human that will execute the algorithms. The set is surprisingly small and consists of a few **arithmetic operations** that include addition, subtraction, multiplication, division, modulus, and power. Modulus means "the remainder after division" i.e. 3 mod 5 means the remainder after dividing 3 by 5 (the answer is 3). There are also operations for manipulating variables.

   Note that different algorithm writers may use different symbols. For example you may see `<-` *being used for assignment and* `=` *being used for "is equal to".*

   Algorithms often make use of variables that denote particular values or numbers. They are similar to the use of variables in Mathematics. In the algorithm above we use the variables m, n, and r. Note that a variable refers to any number (hence their name) but at any time they each have a particular value. The **value** of a variable changes as the algorithm executes. This is done with the **assignment** operator, i.e. "=". The = means that we replace the value of of variable on the left hand side with the value of whatever is on the right hand side. When we executed the algorithm, m and n were given the values 12 and 8. In the second step, m is **assigned** the value 8 and n is assigned the value 4.

   There are also operations for comparing numbers – the **comparison operators**. These would include `==` (is equal to), `<` (is less than), `<=` (is less than or equal to), `>` (is greater than), `>=` (is greater than or equal to), `!=` (is not equal to). These operators allow us to ask true/false questions such as *Is this number greater than that one or is this number equal to this one*.

4. Algorithms use control structures

   When algorithms are executed the computer or human follows and completes each step. Its makes sense to control the order in which the steps are

executed. In the GCD algorithm you can see that we have three such ways to control execution of the steps. First, we specify that the steps must be carried out in the **sequence** they are written (i.e. one after another). The order of the steps are thus very important. See step 3. $m = n$ and then $n = r$ is very different from $n = r$ followed by $m = n$. If we followed the second sequence then whatever value is in n would be overwritten by whatever is in r. Thus, the value of n is lost before we can assign it m.

The second way to control the execution of statements is **conditional branching**. For example, in the GCD algorithm, we see that if r is 0, the answer is n and we stop; otherwise (if r is anything else, we continue to step 3. This type of control is called **branching** because we have to take one of two paths through the algorithm. The path we take depends on the answer to a true-false question (i.e. is r equal to 0) – this question is called a **condition**.

Repetition is also known as iteration

The third way to control the execution of statements is **repetition** . In this case we specify that some statements (maybe all) must be repeated. In the GCD algorithm, step 3 ends with *go back to Step 1*. After doing the assignments in step 3, we go back to step 1 and repeat the steps. Repetition is tricky - we must take care that the algorithm will eventually terminate. Algorithms that go on and on and on are problematic. Loops that do not eventually terminate are known as **infinite loops**.

5. Algorithms are finite.

   An algorithm must terminate after a finite number of steps. Proving that an algorithm eventually terminates is not always straight forward – we need to show that it will **always** terminate. To show that the GCD algorithm will terminate we argue as follows: After step 1 the value of r is *less* than n so if r *!= 0* the value of r decreases the next time we get to step 1. So a decreasing sequence of positive integers must eventually terminate so step 1 will only be executed a finite number of times. We should note that *finite* does not mean *small* or *few* as steps in algorithms may run any number of times - millions or billions or more.

In general no procedure exists that will determine if a particular algorithm will terminate or not.

6. Algorithms must be unambiguous.

   Each step of an algorithm must be precisely defined and unambiguous i.e. there must be no doubt or uncertainty as to what needs to be done. Steps in an algorithm should leave no room for interpretation. This ensures that the algorithm will always produce the same results if given the same inputs even if executed by different computers or humans.

This is why natural languages such as English are are not good for writing algorithms.

7. Algorithms have inputs.

   Most algorithms require some values to be input either before it starts or while executing it. The GCD algorithm takes two inputs viz. m and n.

Note that the set of input values may be restricted i.e. not every value input to it will produce correct results: the GCD algorithm takes only positive integer values.

8. Algorithms have outputs.

   An algorithm has one or more outputs. The output of the GCD algorithm is the greatest common divisor of the two positive integers input.

To close off this section on algorithms we give another example of an algorithm - an algorithm to determine the day of the week for a given month, day and year :

Use this algorithm to determine on which day of the week did November 18, 1992 fall. (You should get Wednesday.)

Note that `floor(x)` is the largest integer not greater than `x` and `ceiling(x)` is the smallest integer not less than `x`. Also `x mod y` is the remainder after dividing `x` by `y`.

1.2 Languages

Algorithms are written in some language. The GCD algorithm was written in a natural human language (English) and could be suitable for execution by any human that understands the language and understands the instructions i.e. they know how to divide, how to find remainder etc. If we want algorithms to be executed by computers, then we need to use computer languages. Computers are (not yet) able to understand human languages. Computer languages define a precise set of instructions that can be executed by a computer. For example, the instruction divide 35 by 7 and store the answer in the variable x may be written as `x = 35 / 7`.

Languages such English, Zulu, French, Hindi etc. are **natural languages**. They are the languages that people speak and were not designed by people (although people try to impose some order on them); they evolved naturally .

Natural languages can be ambiguous: For e.g. there are at least 5 different interpretations of "I saw a man on a hill with a telescope." And, "Eats, shoots and leaves" may refer to a panda or a disgruntled diner.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And:

> Programming languages are formal languages that have been designed to express algorithms.

There are two important ideas to consider when discussing languages: **syntax** and **semantics**.

Formal languages tend to have strict rules of syntax. The syntax of a language is a set of rules that must be followed to construct valid sentences (or statements)

Listing 1.1: *Day of the Week Algorithm*

```
DAY_OF_THE_WEEK ALGORITHM

INPUTS: day, month, year (all positive integers)
OUTPUT: Day of the week

1. Let century_digits = the first two digits of the year
2. Let year_digits = the last two digits of the year
3. Let value = year_digits + floor(year_digits / 4)
4. if century_digits = 18 then value = value + 2
   else if century_digits = 20 then value = value + 6
5. if month == January and year is not a leap year
     then value = value + 1
   else if month == February and the year is a leap year
           then value = value + 3
        otherwise (year is not a leap year)
           then value = value + 4
   else if month == March or November
     then value = value + 4
   else if month == May
     then value = value + 2
   else if month == June
     then value = value + 5
   else if month == August
     then value = value + 3
   else if month == October
     then value = value + 1
   else if month == September or December
     then value = value + 6
6. value = (value + day) mod 7
7. if value == 1 then day is Sunday
   else if value == 2 then day is Monday
   else if value == 3 then day is Tuesday
   else if value == 4 then day is Wednesday
   else if value == 5 then day is Thursday
   else if value == 6 then day is Friday
   else if value == 0 then day is Saturday
```

in the language. For example, 3 + 3 = 6 is a syntactically correct mathematical statement, but 3 = +6@ is not. H_2O is a syntactically correct chemical name, but ${}_2Zz$ is not. Each programming language has its own rules of syntax. For example, `x = 3 + 5` may be syntactically valid statement but `x + y = 5` may not be valid. A part of the process of learning to program involves learning the syntax rules of the language.

Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with `3=+6@` is that @ is not a legal token in mathematics (at least as far as we know). Similarly, ${}_2Zz$ is not legal because there is no element with the abbreviation `Zz`.

The second type of syntax rule pertains to the **structure** of a statement— that is, the way the tokens are arranged. The statement 3 = +6 is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell", you understand that the other shoe is the subject and fell is the verb. Once you have parsed a sentence, you can figure out what it means, or the **semantics** of the sentence. Assuming that you know what a shoe is and what it means to fall, you will understand the general implication of this sentence.

The semantics of programming languages are clearly and unambiguously specified by the language designers. For example, the syntactically correct statement `x = 3 + 5` may mean *evaluate the expression on the right (work out what 3 + 5 is) and assign the answer to the variable x.*

Although formal and natural languages have many features in common — tokens, structure, syntax, and semantics — there are many differences:

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Programming Languages

Learning to program a computer then requires the ability to write algorithms using a particular programming language. Many programming languages have been invented. The programming language you will be learning is Python. Python is an example of a **high-level language**; other high-level languages you might have heard of are C++, PHP, and Java.

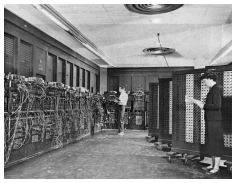


Figure 1.2: *The first computers were programmed by setting jumper cables.*

As you might infer from the term "high-level language", there are also **low-level languages**, sometimes referred to as machine languages or assembly languages. Machine language is the encoding of instructions in numbers so that they can be directly executed by the computer. A machine language program is thus a long list of numbers with each number specifying a particular instruction. The instructions themselves are very simple and usually involve the movement of numbers from one part of the computer to another, adding two numbers and storing the answer somewhere else. Programming in machine code is very tedious – the programmer needs to know what each number of the code does and a long list of instructions is required even for very simple operations. Each kind of computer (actually each kind of processor) has its own machine language.

Assembly language uses a slightly easier format to refer to the low level instructions. Instead of using numbers a mnemonic is used for each instruction. For example instead of a numeric code that says add the number in location R1 to the number in location R2 and store the result in location R3 we write:

```
1 ADD R1 R2 R3
```

This language is easier to use than machine language but is still tedious to use.

See: `http://www.99-bottles-of-beer.net` *that has a program to generate the lyrics of the song in 1500 different programming languages.*

High-level languages were invented to make the writing of programs easier for humans. Computers can only execute programs written in machine language. Thus, programs written in a high-level language (and even those in assembly language) have to be translated into machine language instructions before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages. However, the advantages to high-level languages are enormous.

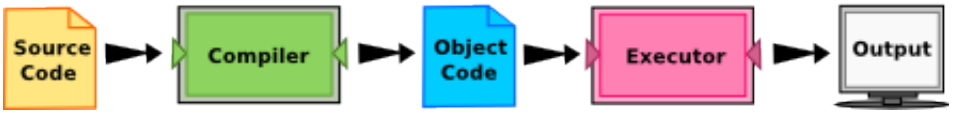
First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications.

Two kinds of programs process high-level languages into low-level languages: **interpreters** and **compilers**. An interpreter reads a high-level program and exe-

cutes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.

A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation.



Many modern languages use both processes. They are first compiled into a lower level language, called **byte code**, and then interpreted by a program called a **virtual machine**. Python uses both processes, but because of the way programmers interact with it, it is usually considered an interpreted language.

Figure 1.3: *Guido van Rossum: Author of the Python Programming language.*

Python was named for the Guido's favorite television show: "Monty Python's Flying Circus."

Using Python

We want to now get started on designing algorithms to solve problems and then expressing them (writing them) in Python. Before doing that, we look at various ways of using Python.

Python at the command line

The first way, that we will **not** use frequently is **command line interaction** with Python. All major operating systems (Windows, Linux and OSX on the Macintosh) have a command line interpreter. This is a special program that allows the user to type commands that the interpreter then runs. The program on Windows is called *cmd*, and on Linux and OSX it is called *Terminal*. The picture below shows the terminal in OSX (it is similar to the one on Linux). The picture below shows the Terminal program with the first command `pwd` that prints the current directory followed by the command *ls* that lists the directories and files in the current directory and the copy command (*cp*) that copies files and directories. There are many commands that are available in the terminal, including moving

Before 1973 (when the first Graphical User Interface was developed) the terminal was the only way to interact with a computer.

and deleting files, changing directory, editing text files etc. The next command shown below *python3* starts the Python interpreter.

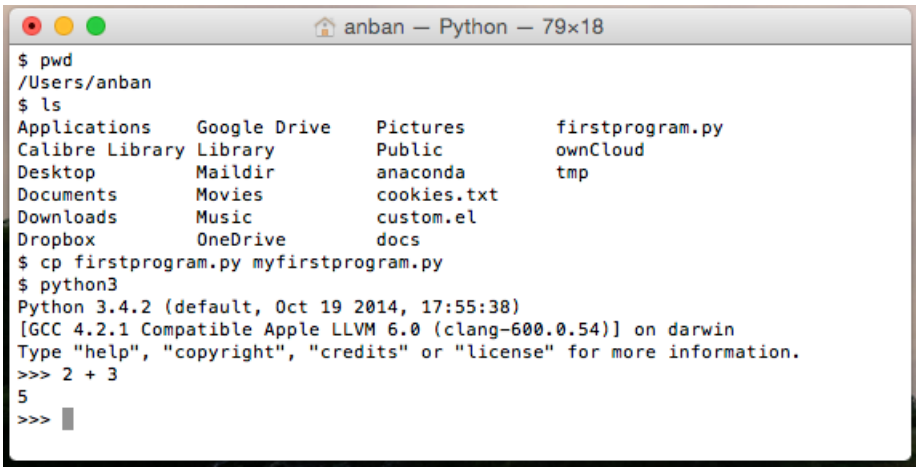


Figure 1.4: *Python at the command line*

The Python interpreter is a program that waits for the user to type python statements that it then executes and it then displays the results. When we use the Python interpreter in this way we say we are using it in *shell mode*. In shell mode, you type Python expressions into the **Python shell**, and the interpreter immediately shows the result. In the picture above the terminal command `python3` starts the interpreter. When the interpreter starts it prints out some helpful information and then presents the user with the **Python prompt**. The `>>>` is called the **Python prompt**. The interpreter uses the prompt to indicate that it is ready for instructions. We typed `2 + 3`. The interpreter evaluated our expression and replied `5`. On the next line it gave a new prompt indicating that it is ready for more input.

Working directly in the interpreter is convenient for testing short bits of code because you get immediate feedback.

Another way to use the Python interpreter is to write an entire program by placing lines of Python instructions in a file and then use the interpreter to execute the contents of the file as a whole. Such a file is often referred to as **source code**. For example, we use a text editor to create a source code file named `firstprogram.py` with the following contents:

```
1 print(”My first program adds two numbers, 2 and 3:”)
2 print(2 + 3)
```

By convention, files that contain Python programs have names that end with `.py`. Following this convention will help your operating system and other pro-

grams identify a file as containing python code. We then ask the Python interpreter to execute the instructions in the file as shown here:

```
$ python firstprogram.py
My first program adds two numbers, 2 and 3:
5
```

Writing programs on your Desktop/Laptop

We will use a program called WingIDE 101 . This is an IDE (an integrated development platform). It includes an editor for typing python programs and allows you to save the programs on disc and to run and debug them. It also provides a way to interact with the python interpreter. All this in one program - which is why its called an **integrated** development environment.

Download for free from `http://wingware.com/downloads/wingide-101/`

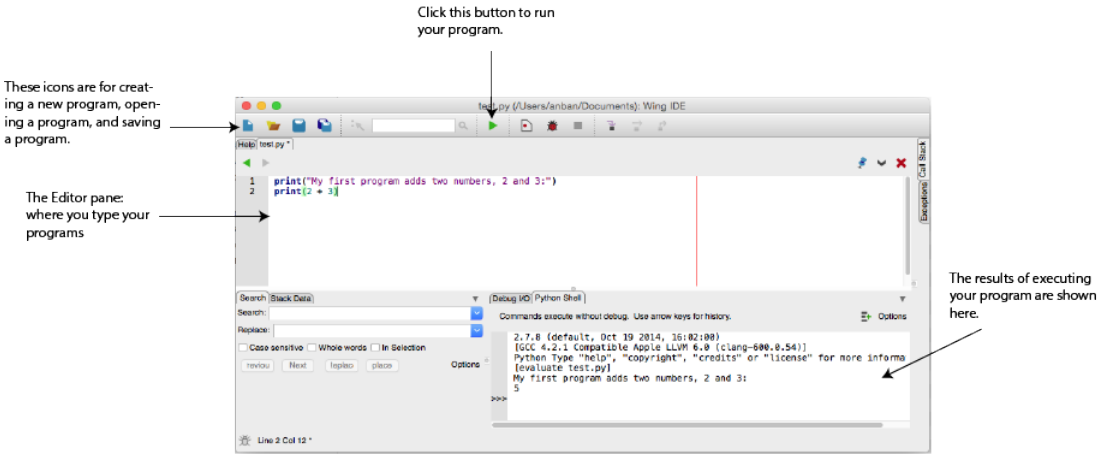


Figure 1.5: *Wing IDE*

Your First Program

For almost as long as programming has been taught, the first program written in a new language is called *Hello, World!* because all it does is display the words, **Hello, World!**. A beginner programmers' first program is thus to greet the world. In Python, the source code looks like this.

```
1 print("Hello, World!")
```

This is an example of using the **print function**, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result is the phrase:

```
1 Hello, World!
```

printed on the screen. The quotation marks in the program mark the beginning and end of the value we want printed. They don't appear in the result.

1.3 Exercises

1. Design an algorithm to find an approximation of $\sqrt{x}$. The following approach is known as the Babylonian method:

   - ▷ Make a guess of the square root, call it G.
   - ▷ Improve the guess by averaging G and x/G.
   - ▷ Keep improving the guess until it is good enough.

   Use this approach as the basis of your algorithm.

2. Write an algorithm to compute $x \div y$ using only addition, subtraction and multiplication.

3. Write an algorithm that takes three integers as input and calculates the sum of the squares of the larger numbers.

Chapter

2 Elements of Programs

Theory is when you know something, but it doesn't work. Practice is when something works, but you don't know why. Programmers combine theory and practice: Nothing works and they don't know why.

Unknown

Code never lies, comments sometimes do.

- Ron Jeffries

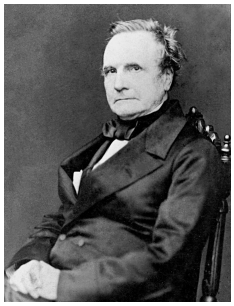


Figure 2.1: *Charles Babbage (1791-1871) Originated the concept of a programmable computer by inventing the first mechanical computer.*

2.1 Introduction

Recall that we want to use the Python programming language to write algorithms that may be executed by computers.

Algorithm
An algorithm is a self contained step-by-step set of operations that begins with some input and produces some output when executed.

Python provides the fundamental building blocks to use in constructing algorithms. The building blocks required for algorithms include the following:

It is surprising how small the list of building blocks required to write any algorithm possible!

Literals are things that stand for themselves, such as actual numbers and characters.

1. LITERALS: to express the data that the algorithms manipulate. These are the actual values (numeric and text values such as "42", and "Hello!".
2. VARIABLES: to **refer** to particular values. The language must provide a way to create and assign variables. This provides a way to name values.
3. ARITHMETIC EXPRESSIONS: The language must provide operators that are used together with literals and variables to form arithmetic expressions such as:
   `(v * 0.5 * (a / t))`.

Booleans are named after the British logician George Boole who invented an algebra of logic we now call a Boolean algebra.

4. BOOLEAN EXPRESSIONS: The language must provide comparison operators that are used together with literals and variables to form boolean expressions that evaluate to `true` or `false`. Boolean expressions are used to test values and variables, for example `(x + y)>= 0`.
5. INPUT FUNCTIONS: to allow us to read input values from the user.
6. OUTPUT FUNCTIONS: to allow us to write results to the screen.
7. CONTROL STRUCTURES: that allow us to control the execution of the algorithm.

The goal of this chapter is to introduce you to the basic vocabulary of programming and some of the fundamental building blocks of Python.

2.2 Output with the print function

The idea of a function in programming is a very important idea that we explore in more detail later.

The format of a function is thus: name(arg1,[arg2,...])

Python provides several built-in functions. In a later chapter we show how you could write your own functions.

Output from a program is used communicate the results produced from running the program to the outside world i.e. users of the program. Python provides a **function**, the **print function** for output. In Python, a function is a named sequence of instructions that accomplishes some task . Functions take one or more inputs, processes the inputs and produces some output or accomplishes some task. Other words for *input* are **parameter** and **argument**. So we say that a function has parameters or arguments. Having parameters or arguments for functions is a very powerful idea—it means that we can call the same function on different values to get different outcomes. Python provides many functions for programmers to use . To use them, the programmer needs to know the name of the function and the kind of input it takes.

```
1 print("Hello, cruel world")
2 print(42)
3 print("The answer is ",42,"!")
```

Functions have this format `name(inputs)`. The inputs to the print function are the things we want to print (for example, the first statement has the text we want printed as input while the second has the number we want printed. When executed, each print function displays its input on the screen. The last example

shows that the print function accepts multiple inputs separated by commas. In this example the print function takes 3 inputs.

So the print function takes a variable number of arguments (0, 1 or more).

2.3 Literals

A **literal** is one of the fundamental things – like a word or a number – that a program manipulates. A literal is a sequence of characters that stand for itself. Things like `0`, `42`, `3.14`, `Hello` are literals – they stand for themselves i.e. what you see is what you get.

A **character** is any single thing we can type at the keyboard. Simply put, each key on the keyboard allows us to type one character. We use the numeric characters (and optionally the `.`, `+`, `-` and `e`) to type numbers and all the characters (alphabetic, punctuation, numeric, etc.) to type words or text. Note that we also have some non-printable characters–characters that do not show up when we type them such as the space, the new-line, the tab etc.

Numeric Literals

A **numeric literal** is used to type numbers and it is a literal that contains only the digits 0-9, an optional sign character (+ or -) and an optional decimal point (`.`) and optionally the character `e` for exponential notation. No other characters (and especially not the comma `,`) are allowed when writing numeric literals.

Programming languages, generally, distinguish between two kinds of numbers. If a numeric literal has a decimal point then it is a **floating point** number (or simply a **float**), e.g. `42.42`; otherwise it is an **integer**. Humans do not make such a big deal of this distinction but programming languages do, due to the way numbers are represented inside the computer .

Integers are stored differently from floating point numbers inside a computer.

For each of the following examples it is easy to tell which are integers and which are floats.

```
1 5        5.       5.0       0.00000005
2 +5       -5       +5.0      -0.00000005
```

Numeric values may also be expressed using exponential notation. For example, to show $42.004x10^4$ we type `42.004e+4` and to type $0.03x10^{-2}$ we type `0.03e-2`.

Remember to **not** put commas or spaces in your integers, no matter how big they are. Also revisit what we said in the previous chapter: formal languages are strict, the notation is concise, and even the smallest change might mean something quite different from what you intended.

String Literals

To type normal text (words and sentences) we use various quotation marks to separate them from the rest of the program. In programming, we use the word **string** to refer to text. String literals, then, is any sequence of characters surrounded by single quotes (') or double quotes (") or three of each (''' or """). String literals are usually used to communicate with users – for example to output answers or to ask for specific input.

We should take care to understand that a string is **any** sequence of characters surrounded by quotes. What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks–so they are strings! This is an important point so 42 and "42" are not the same thing.

```
print('This is a string.')
print("And so is this.")
print("""and this.""")
print('''and even this...''')
```

Double quoted strings can contain single quotes inside them, as in "Bruce's beard", and single quoted strings can have double quotes inside them, as in this string:
'The knights who say "Ni!"'. Strings enclosed with three occurrences of either quote symbol are called triple quoted strings. They can contain either single or double quotes:

```
print('''"Oh no", she exclaimed, "Ben's bike is broken!"''')
```

Triple quoted strings can even span multiple lines:

```
print("""This message will span
several lines
of the text.""")
```

Python doesn't care whether you use single or double quotes or the three-of-a-kind quotes to surround your strings. Once it has parsed the text of your program or command, the way it stores the value is identical in all cases, and the surrounding quotes are not part of the value.

```
print('This is a string.')
print("""And so is this.""")
```

2.4 Variables

Algorithms often make use of **variables** that denote particular values. They are similar to the use of variables in Mathematics. A variable is a name that refers to a value. Note that a variable can refer to any value (hence their name) but

at any time it refers to a particular value. The value of a variable changes as the algorithm executes.

Assignment statements create new variables and also give them values to refer to.

```
1 message = "What's up, Doc?"
2 n = 17
3 pi = 3.14159
```

This example makes three assignments. The first assigns the string value `"What's up, Doc?"` to a new variable named `message`. Note carefully that the variable's name is `message` and the value it refers to is the string `"What's up, Doc?"`

Ensure that you are not confused by this legal statement: `message = "message"`

The second assignment assigns the integer `17` to `n`, and the third assigns the floating-point number `3.14159` to a variable called `pi`. The assignment operator, i.e. `=` means that we make the variable on the left hand side point to the value of whatever is on the right hand side.

The **assignment token**, `=`, should not be confused with *equality* (we will see later that equality uses the `==` token). The assignment statement links a *name*, on the left hand side of the operator, with a *value*, on the right hand side. This is why you will get an error if you enter :

What error do you get?

```
1 17 = n
```

> **Tip**
>
> When reading or writing code, say to yourself "n is assigned 17" or "n gets the value 17" or "n is a reference to the object 17" or "n refers to the object 17". Don't say "n equals 17".

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure, known as a **reference diagram**, is often called a **state snapshot** because it shows what state each of the variables is in at a particular instant in time. (Think of it as the variable's state of mind). This diagram shows the result of executing the assignment statements shown above.

We use variables in a program to "remember" things, like the current score at the football game. But variables are *variable*. This means they can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign a different value to the same variable. But it is important to note that at any particular time, the variable refers to one particular value.

> **Note**
>
> This is different from math. In math, if you give x the value

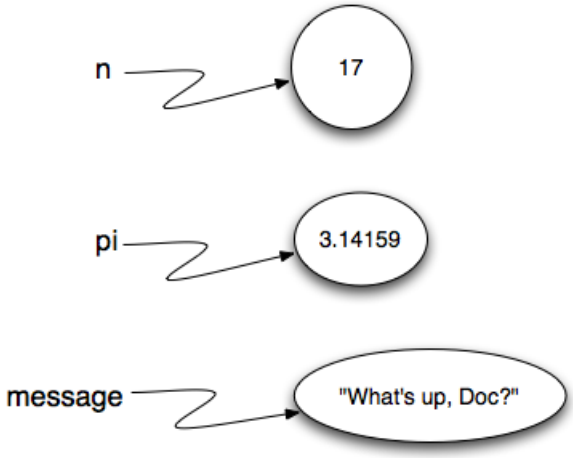


Figure 2.2: *A Reference Diagram*

3, it cannot change to refer to a different value half-way through your calculations!

To see this, read and then run the following program. You'll notice we change the value of `day` three times, and on the third assignment we even give it a value that is of a different type .

You should avoid assigning different types of things to a variable (even though it is possible). It is confusing and is source of errors in programs.

```
1 day = "Thursday"
2 print(day)
3 day = "Friday"
4 print(day)
5 day = 21
6 print(day)
```

A great deal of programming is about having the computer remember things. For example, we might want to keep track of the number of missed calls on your phone. Each time another call is missed, we will arrange to update or change the variable so that it will always reflect the correct value.

Reassignment

As we have mentioned previously, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
1 bruce = 5
2 print(bruce)
3 bruce = 7
4 print(bruce)
```

The first time `bruce` is printed, its value is 5, and the second time, its value is 7. The assignment statement changes the value (the object) that `bruce` refers to.

Here is what **reassignment** looks like in a reference diagram:

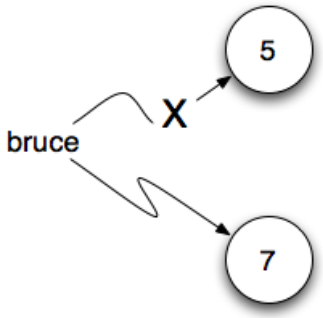


Figure 2.3: *Reassignment*

It is important to note that in mathematics, a statement of equality is always true. If "`a` is equal to `b`" now, then `a` will always equal to `b`. In Python, an assignment statement can make two variables refer to the same object and therefore have the same value. They appear to be equal. However, because of the possibility of reassignment, they don't have to stay that way:

```
1 a = 5
2 b = a    # after executing this line, a and b are now equal
3 print(a, b)
4 a = 3    # after executing this line, a and b are no longer equal
5 print(a, b)
```

Line 4 changes the value of `a` but does not change the value of `b`, so they are no longer equal. We will have much more to say about equality in a later chapter.

> **Note**
>
> In some programming languages, a different symbol is used for assignment, such as `<-` or `:=`. The intent is that this will help to avoid confusion. Python chose to use the tokens `=` for assignment, and `==` for equality. This is a popular choice also found in languages like C, C++, Java, and C#.

Updating Variables

One of the most common forms of reassignment is an **update** where the new value of the variable depends on the old. For example,

```
1 x = x + 1
```

This means get the current value of x, add one, and then update x with the new value. The new value of x is the old value of x plus 1. Although this assignment statement may look a bit strange, remember that executing assignment is a two-step process. First, evaluate the right-hand side expression. Second, let the variable name on the left-hand side refer to this new resulting object. The fact that x appears on both sides does not matter. The semantics of the assignment statement makes sure that there is no confusion as to the result.

```
1 x = 6         # initialize x
2 print(x)
3 x = x + 1     # update x
4 print(x)
```

If you try to update a variable that doesn't exist, you get an error because Python evaluates the expression on the right side of the assignment operator before it assigns the resulting value to the name on the left. Before you can update a variable, you have to **initialize** it, usually with a simple assignment. In the above example, x was initialized to 6.

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**. Sometimes programmers also talk about **bumping** a variable, which means the same as incrementing it by 1.

2.5 Identifiers and Keywords

An **identifier** is a name given to an entity in a program. Variables are one kind of entity we name and the names given to variables are referred to as identifiers. Like all programming, Python has rules for identifiers. Identifiers (names) can be arbitrarily long, can contain both letters and digits, but they have to begin with a letter or an underscore. Python is case sensitive – it distinguishes between upper case and lower case characters. num and Num are different variables.

> **Caution**
> Variable names can never contain spaces.

The underscore character (_) can also appear in a name. It is often used in names with multiple words, such as my_name or price_of_tea_in_china. There are some situations in which names beginning with an underscore have special meaning, so a safe rule for beginners is to start all names with a letter.

Some programmers do not like the use of underscores in identifiers and find these identifiers difficult to read. They prefer camel case where the underscores are not used but the first letter of each word (except the first) is capitalized. Thus we can have myName or priceOfTeaInChina. Note that there is no rule in Python

that insists that the first character of the name should be lower case but all Python programmers do this.

> **Caution**
>
> Do not confuse strings with identifiers (names). In the statement `day = "Thursday"`, `day` is the identifier (name) of a variable and `Thursday` is the value the variable refers to.

If you give a variable an illegal name, you get a syntax error. In the example below, each of the variable names is illegal.

Note that String values do not follow the rules for identifiers. They can contain any character.

```
1 76trombones = "big parade"
2 more$ = 1000000
3 class = "Computer Science 101"
```

`76trombones` is illegal because it does not begin with a letter. `more$` is illegal because it contains an illegal character, the dollar sign. But what's wrong with `class`?

It turns out that `class` is one of the Python **keywords**. Keywords define the language's syntax rules and structure, and they cannot be used as identifiers. Python has thirty-something keywords (and every now and again improvements to Python introduce or eliminate one or two):

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

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

Table 2.1: *Python Keywords*

Programmers generally choose names for their variables that are meaningful to the human readers of the program — they help the programmer document, or remember, what the variable is used for. This is an important point. Although we write programs for computers, writing programs that humans can read helps programmers find problems in code, helps them to understand how the code works so they can improve the programs or learn from them. So naming your variables after your boyfriend or girlfriend may be cute and impress him or her, programmers who read your code will not be impressed. So variable names **must** be meaningful and give an indication of what value the variable is referring to. So a name such as `x` is meaningless, `sum` may be better and `sumOfMarks` may be even better.

Caution

Beginners sometimes confuse "meaningful to the human readers" with "meaningful to the computer". So they'll wrongly think that because they've called some variable `average` or `pi`, it will somehow automagically calculate an average, or automagically associate the variable `pi` with the value 3.14159. No! The computer doesn't attach meaning to your variable names.

2.6 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments.

A **comment** in a computer program is text that is intended only for the human reader - it is completely ignored by the interpreter. In Python, the # token starts a comment. The rest of the line is ignored. Here is a new version of *Hello, World!*.

```
#-----------------------------------------------
# This demo program shows off how elegant Python is!
# Written by Joe Soap, December 2010.
# Anyone may freely copy or modify this program.
#-----------------------------------------------

print("Hello, World!")     # Isn't this easy!
```

Notice that when you run this program, it still only prints the phrase Hello, World! None of the comments appear. You'll also notice that we've left a blank line in the program. Blank lines are also ignored by the interpreter, but comments and blank lines can make your programs much easier for humans to parse.

Robert C. Martin said "The proper use of comments is to compensate for our failure to express ourself in code."

Comments are meant for other programmers to help them understand your code. So it is not necessary to comment every line: If your code can be understood then do not write a comment–write comments only for code that is difficult to understand.

2.7 Arithmetic Operators

An **operator** is a symbol, such as `+`, that represents an **operation** or **computation** that may be performed on one or more things (for e.g. values or variables) called operands. An **operand** are the things that the operator operates on. An **expression** is a combination of operands and operators. For example, `2 + 3` is

an expression with operator `+` and two operands `2` and `3`. Syntactically correct expression are **evaluated** by Python and a value results. The value is the result of applying the operator to the operand/s. A **unary** operator operates on only one operand. For example, in the expression `-x`, `-` is the unary **negation** operator. Operators that operate on two operands are known as **binary** operators.

The following are all legal Python expressions whose meaning is more or less clear:

```
20 + 32
hour - 1
hour * 60 + minute
minute / 60
5 ** 2
(5 + 9) * (15 - 7)
```

The tokens `+`, `-`, and `*`, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (`*`) is the token for multiplication, and `**` is the token for exponentiation. Addition, subtraction, multiplication, and exponentiation all do what you expect. If the input to the print function is an expression then Python evaluates the expression before printing out the resulting value (answer).

```
print(2 + 3)
print(2 - 3)
print(2 * 3)
print(2 ** 3)
print(3 ** 2)
```

When a variable name appears in the place of an operand, it is replaced with the value that it refers to before the operation is performed. For example, what if we wanted to convert 645 minutes into hours. In Python 3, division is denoted by the operator token `/` which always evaluates to a floating point result.

```
minutes = 645
hours = minutes / 60
print(hours)
```

What if, on the other hand, we had wanted to know how many *whole* hours there are and how many minutes remain. To help answer this question, Python gives us a second flavor of the division operator. This version, called **truncating division**, uses the token `//`. It always *truncates* its result down to the next smallest integer (to the left on the number line).

```
print(7 / 4)
print(7 // 4)
minutes = 645
hours = minutes // 60
print(hours)
```

Pay particular attention to the first two examples above. Notice that the result of floating point division is `1.75` but the result of the integer division is simply `1`. Take care that you choose the correct flavor of the division operator. If you're working with expressions where you need floating point values, use the division operator `/`. If you want an integer result, use `//`.

The **modulus operator**, sometimes also called the **remainder operator** or **integer remainder operator** works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (`%`). The syntax is the same as for other operators.

```
quotient = 7 // 3     # This is the integer division operator
print(quotient)
remainder = 7 % 3
print(remainder)
```

In the above example, 7 divided by 3 is 2 when we use integer division and there is a remainder of 1.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by `y`. Also, you can extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x`. Similarly `x % 100` yields the last two digits.

Finally, returning to our time example, the remainder operator is extremely useful for doing conversions, say from seconds, to hours, minutes and seconds. If we start with a number of seconds, say 7684, the following program uses integer division and remainder to convert to an easier form. Step through it to be sure you understand how the division and remainder operators are being used to compute the correct values.

```
total_secs = 7684
hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes =  secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining  % 60
```

Here is a summary of the arithmetic operators where `x` and `y` are either numeric literals or variables referring to numbers:

The division `/` and the truncating division `//` operators can cause you some grief if you do not understand them well. The application of the division operator **always** results in a float. The results of truncating division operator depends on the type of the operands (i.e. whether they are floats or integers). When the truncating division operator is applied to operands that are both integers then the result is an integer (i.e. the integer obtained by chopping off the decimal part). This truncated division is also known as **integer division**. If one **or** both

Arithmetic Operators		Example	Result
`-x`	negation	`-42`	-42
`x + y`	addition	`33 + 9`	42
`x - y`	subtraction	`52 - 10`	42
`x * y`	multiplication	`6 * 7`	42
`x / y`	division	`84 / 2`	42.0
`x // y`	truncating div	`428 // 10`	42
		`428 // 10.0`	42.0
`x % y`	modulus	`42 % 53`	42
`x ** y`	exponentiation	`4 ** 2`	16

Table 2.2: *Arithmetic Operators in Python*

of the operands are then floats, then the result is a truncated float (i.e. the float obtained by chopping off the decimal part of the result). The table below shows this:

	Operands	result type	example	result
`/` (Division)	int, int	float	`7 / 5`	1.4
	int, float	float	`7 / 5.0`	1.4
	float, float	float	`7.0 / 5.0`	1.4
`//` (Truncating Division)	int, int	truncated int	`7 // 5`	1
	int, float	truncated float	`7 // 5.0`	1.0
	float, float	truncated float	`7.0 // 5.0`	1.0

Table 2.3: *The Division Operators*

Order of Operations

Recall that an expression is a syntactically correct combination of operators and operands that evaluates to a value. Interesting expressions have many operators and operands.

```
1 (v ** 2) + (2 * a * d)
2 P * ((1 + (r / n)) ** ( n * t))
```

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does.

Perhaps you were taught the BODMAS *rules in school. The idea is the same but the rules are different.*

1. Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use

If you use brackets a lot then you do not have worry too much about precedence rule.

parentheses to make an expression easier to read, as in `(minute * 100)/ 60`, even though it doesn't change the result.

2. Exponentiation has the next highest precedence, so `2**1+1` is 3 and not 4, and `3*1**3` is 3 and not 27. Can you explain why?
3. Multiplication, both division operators, and modulus have the same precedence, which is higher than addition and subtraction, which also have the same precedence. So `2*3-1` yields 5 rather than 4, and `5-2*2` is 1, not 6.
4. Operators with the *same* precedence (i.e. on the same level) are evaluated from left-to-right except for exponentiation which are evaluated from right-to-left. So in the expression `6-3+2`, the subtraction happens first, yielding 3. We then add 2 to get the result 5. But in the expression `2 ** 3 ** 2` the right-most operator gets done first and the result of the expression is `512`. You may use parentheses to force a different order. In the expression `(2 ** 3)** 2` the left most operator evaluates first to give 8 and then the right most one is applied to give a final value of `64`.

This table summarizes the precedence rules in Python:

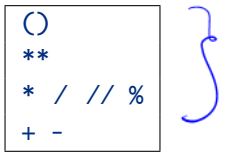


Table 2.4: *Operator Precedence*

A useful hint is to use parentheses to make the order you want clear. Here are some examples:

```
print(2 ** 3 ** 2)   # the right-most ** operator
                     # is done first!
print((2 ** 3) ** 2) # use brackets to force
                     # the order you want!
4 + 2 ** // 10
```

2.8 Data Types

We have previously studied values that programs manipulate. Values manipulated by programs are also referred to as **data** or **objects**. We encountered different types of values (or data) thus far: the two numeric types (integers and floats) and strings. We looked at how to write literals of these types and how to assign them to variables. In this section we look more closely at types.

These objects are classified into different **classes**, or **data types**: `4` is an *integer*, and `"Hello, World!"` is a *string*, so-called because it contains a string or sequence

of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

If you are not sure what class a value falls into, Python has a function called **type** which can tell you. The input to the function is a literal or variable and the output is its type.

```
1 print(type("Hello, World!"))
2 print(type(17))
3 print("Hello, World")
4 x = 42.3
5 print(type(x))
```

Not surprisingly, strings belong to the class **str** and integers belong to the class **int**.

> **Note**
>
> When we show the value of a string using the `print` function, such as in the third example above, the quotes are no longer present. The value of the string is the sequence of characters inside the quotes. The quotes are only necessary to help Python know what the value is.

In the Python shell, it is not necessary to use the `print` function to see the values shown above. The shell evaluates the Python function and automatically prints the result. For example, consider the shell session shown below. When we ask the shell to evaluate `type("Hello, World!")`, it responds with the appropriate answer and then goes on to display the prompt for the next use.

```
1 Python 3.1.2 (r312:79360M, Mar 24 2010, 01:33:18)
2 [GCC 4.0.1 (Apple Inc. build 5493)] on darwin
3 Type "help", "copyright", "credits" or "license" for more
      information.
4 >>> type("Hello, World!")
5 <class 'str'>
6 >>> type(17)
7 <class 'int'>
8 >>> "Hello, World"
9 'Hello, World'
10 >>>
```

Note that in the last example, we simply ask the shell to evaluate the string "Hello, World". The result is as you might expect, the string itself.

Continuing with our discussion of data types, numbers with a decimal point belong to a class called **float**, because these numbers are represented in a format called *floating-point*. At this stage, you can treat the words *class* and *type* interchangeably.

```
print(type(3.2))
```

What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings.

```
print(type("17"))
print(type("3.2"))
```

They are strings!

So what exactly is a data type? A **data type** is a set of values together with a set of operators that may be applied to those values. So for the `int` data type the whole numbers is the set of values and the operators would include addition, subtraction, multiplication, and division etc. Similarly for floats, the set is the set of real numbers and the usual operators.

Programming languages need to carefully distinguish between types to ensure that programmers do not use values incorrectly for example by trying to divide a string by a number. Languages deal with data typing in two ways. In languages with **static typing**, a variable must be declared as a certain type before it is used and is only allowed to refer to values of that type throughout the life of the program. For example, in a language such as Java we have:

```
int num = 5;
num = 2.0;  //ERROR - cannot assign a float to an int
```

Python, however, is **dynamically typed** - the data type of a variable is the type of the value it is currently referring to. This means that the type of a variable may change as the program executes.

```
num = 5    # num is of type int
num = 2.0  # num is now of type float
num = "Hello" # num is not type string
answer = num / 2 # ERROR
```

To avoid errors like this, do not use a variable to hold more than one type of data.

So dynamically typed languages do not provide the safety net that statically typed languages provide. You only find the error when the program is running but in languages such as Java, the program will not compile until the error is fixed. You have to be careful about the use of variables in any programming language–more so in dynamically typed languages like Python since the interpreter/compiler does not check if you are using variables correctly.

Type conversion functions

Sometimes it is necessary to convert values from one type to another. Python provides a few simple functions that will allow us to do that. The functions `int`, `float` and `str` will (attempt to) convert their arguments into types `int`, `float` and `str` respectively. We call these **type conversion** functions.

The `int` function can take a floating point number or a string, and turn it into an int. For floating point numbers, it *discards* the decimal portion of the number - a process we call *truncation towards zero* on the number line. Let us see this in action:

```
>>> print(3.14, int(3.14))
(3.14, 3)
>>> print(3.9999, int(3.9999))  # This doesn't round to the
    closest int!
(3.9999, 3)
>>> print(3.0, int(3.0))
(3.0, 3)
>>> print(-3.999, int(-3.999))
(-3.999, -3)
>>> print("2345", int("2345"))        # parse a string to
    produce an int
('2345', 2345)
>>> print(17, int(17))               # int even works on
    integers
(17, 17)
>>> print(int("23bottles"))  #ERROR
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23bottles'
>>>
```

The last case shows that a string has to be a syntactically legal number, otherwise you'll get one of those pesky runtime errors. Modify the example by deleting the `bottles` and rerun the program. You should see the integer `23`.

The type converter `float` can turn an integer, a float, or a syntactically legal string into a float.

```
print(float("123.45"))
print(type(float("123.45")))
```

The type converter `str` turns its argument into a string. Remember that when we print a string, the quotes are removed. However, if we print the type, we can see that it is definitely `str`.

```
print(str(17))
print(str(123.45))
print(type(str(123.45)))
```

2.9 Input

The program in the previous section works fine but is very limited in that it only works with one value for `total_secs`. What if we wanted to rewrite the program

so that it was more general. One thing we could do is allow the user to enter any value they wish for the number of seconds. The program could then print the proper result for that starting value.

In order to do this, we need a way to get **input** from the user. Python provides a built-in function to accomplish this task. As you might expect, it is called `input`.

```
n = input("Please enter your name: ")
```

User friendly programs will have clear prompt strings so the user knows exactly what kind of input is required.

The input function allows the programmer to provide a **prompt string**. When the function is evaluated, the prompt is shown. The user of the program can enter the name and press return. When this happens the text that has been entered is returned from the input function, and in this case assigned to the variable `n`. Make sure you run this example a number of times and try some different names in the input box that appears.

```
n = input("Please enter your name: ")
print("Hello", n)
```

It is very important to note that the `input` function returns a string value. Even if you asked the user to enter their age, you would get back a string like `"17"`. It would be your job, as the programmer, to convert that string into an int or a float, using the `int` or `float` converter functions we saw earlier.

To modify our previous program, we will add an input statement to allow the user to enter the number of seconds. Then we will convert that string to an integer. From there the process is the same as before. To complete the example, we will print some appropriate output.

```
str_seconds = input("Please enter the number of seconds you wish
    to convert")
total_secs = int(str_seconds)

hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes =  secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining  % 60

print("Hrs=", hours, "mins=", minutes, "secs=",
    secs_finally_remaining)
```

The variable `str_seconds` will refer to the string that is entered by the user. As we said above, even though this string may be `7684`, it is still a string and not a number. To convert it to an integer, we use the `int` function. The result is referred to by `total_secs`. Now, each time you run the program, you can enter a new value for the number of seconds to be converted.

2.10 The math module

We have already used functions such as `print`, `input`, and `type`. Recall that a function is a named sequence of instructions that accomplishes some task. Functions take one or more inputs (or *parameters* or *arguments*), processes the inputs and produces some output or accomplishes some task. We use a function by *calling* or *invoking* it i.e. we type its name and the arguments. For example to print something we did this:

```
1 print("The answer is: ", 42)
```

The three functions we mentioned above are **built-in** functions - they are part of the Python language. Python provides hundreds of useful functions that we may use in our programs.

Some functions are not built-in but are collected in **modules**. A module is a collection of useful functions that are available to programmers. Modules may also contain **constants** which are named values. A constant then, is a name of a particular value and unlike a variable, the value referred to by a constant cannot be changed. An example of a constant is `pi` i.e. the mathematical constant pi.

Python has a huge collection of modules that programmers may use. See the online documentation for details.

The `math` module contains the kinds of mathematical functions you would typically find on your calculator and some mathematical constants like π and e.

In order to use any python module, such as the `math` module, we need to *import* it into our programs. We do this by typing `import math`.

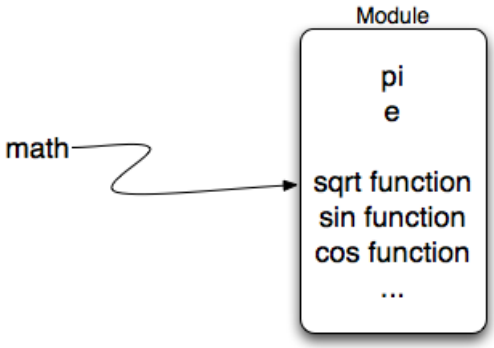


Figure 2.4: *The Python Math Module*

Here are some items from the math module in action. If you want more information, you can check out the documentation online by clicking this link: math module documentation.

```
1 import math
2
3 print(math.pi)
4 print(math.e)
5 print(math.sqrt(2.0))
6 print(math.sin(math.radians(90)))   # sin of 90 degrees
```

In order to use the functions in a module, we need to import the module. We do this with the `import` statement (see line 1). Once we have imported the math module, anything defined there can be used in our program. Notice that we always use the name of the module followed by a dot followed by the specific item from the module (`math.sqrt`). This is to tell Python where the function is located.

Notice that the input to a function can be another function. For example, the input to the third `print` function is the `sqrt` function. The innermost function is evaluated first i.e. Python finds the `sqrt` of 2.0 and the print function prints out the answer. In the fourth print statement, we first convert 90 degrees into radians using the `radians` function, then we calculate the sine of this and finally print out the answer. Thus we print out the sine of 90 degrees. We need to do it like this because the argument to the `sin` function must be in radians.

Here are some useful functions in the `math` module. More details can be found in the documentation.

2.11 The random module

We often want to use **random numbers** in programs. Here are a few typical uses:

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin,
- To shuffle a deck of playing cards randomly,
- To randomly allow a new enemy spaceship to appear and shoot at you,
- To simulate possible rainfall when we make a computerized model for estimating the environmental impact of building a dam,
- For encrypting your banking session on the Internet.

Python provides a module `random` that helps with tasks like this. You can take a look at it in the documentation. Here are the key things we can do with it.

```
1 import random
2
3 prob = random.random()
4 print(prob)
5
```

Function	Meaning
`ceil(x)`	Return the ceiling of x as a float, the smallest integer value greater than or equal to x.
`floor(x)`	Return the floor of x as a float, the largest integer value less than or equal to x.
`exp(x)`	Return e^x.
`log(x, base)`	Return the logarithm of x to the given base.
`log10 (x)`	Return the base-10 logarithm of *x*
`pow (x, y)`	Return x*y.
`sqrt (x)`	Return the square root of x.
`degrees (x)`	Converts angle x from radians to degrees.
`radians (x)`	Converts angle x from degrees to radians.
`sin (x)`	Return the sine of x radians.
`cos (x)`	Return the cosine of x radians.
`tan (x)`	Return the tangent of x radians.
`asin (x)`	Return the arc sine of x, in radians.
`acos (x)`	Return the arc cosine of x, in radians.
`atan (x)`	Return the arc tangent of x, in radians.
`pi`	The mathematical constant pi.
`e`	The mathematical constant e.

Table 2.5: *Functions and constants in the math module*

```
6 diceThrow = random.randrange(1, 7) # return an int, one of
      1,2,3,4,5,6
7 print(diceThrow)
```

Run this program several times and note that the values change each time. These are random numbers.

The `randrange` function generates an integer between its lower and upper argument – the lower bound is included, but the upper bound is excluded. All the values have an equal probability of occurring (i.e. the results are *uniformly* distributed).

The `random()` function returns a floating point number in the range [0.0, 1.0) — the square bracket means "closed interval on the left" and the round parenthesis means "open interval on the right". In other words, 0.0 is possible, but all returned numbers will be strictly less than 1.0. It is usual to *scale* the results after calling this method, to get them into a range suitable for your application.

In the case shown here, we've converted the result of the method call to a number in the range [0.0, 5.0). Once more, these are uniformly distributed numbers — numbers close to 0 are just as likely to occur as numbers close to 0.5, or numbers close to 1.0. If you run the program several times you will see

random values between 0.0 and up to but not including 5.0.

```
import random

prob = random.random()
result = prob * 5
print(result)
```

2.12 Exercises

1. What is the order of the arithmetic operations in the following expression. Evaluate the expression by hand and then check your work.

   2 + (3 - 1) * 10 / 5 * (2 + 3)

2. It is possible to name the days 0 through 6 where day 0 is Sunday and day 6 is Saturday. If you go on a wonderful holiday leaving on day number 3 (a Wednesday) and you return home after 10 nights. Write a general version of the program which asks for the starting day number, and the length of your stay, and it will tell you the number of day of the week you will return on.

3. Add parenthesis to the expression `6 * 1 - 2` to change its value from 4 to -6.

4. Write a program that will compute the circumference and area of a circle. Prompt the user to enter the radius and print a nice message back to the user with the answer.

5. Write a program that will compute MPG for a car. Prompt the user to enter the number of miles driven and the number of gallons used. Print a nice message with the answer.

6. Write a program that will convert degrees fahrenheit to degrees celsius.

7. Write a program which calculates the value of sin(x) where x is expressed in degrees. The input is just the value of x.

8. At the beginning of a journey the reading on a car's odometer is S kilometres and the fuel tank is full. After the journey the reading is F kilometres and it takes L litres to fill the tank.

   Write a program which reads values of S,F, and L and outputs the rate of fuel consumption.

9. In the "old days" measurements used to be expressed in yards, feet and inches (12 inches = 1 foot, 3 feet = 1 yard). Write a program that inputs the number of inches and outputs the number of yards, feet and inches.

10. If an amount of money A earns R% interest over a period of P years then at the end of that time the sum will be:

$$T = A \times (\frac{(100+R)}{100})^P$$

Write a program which inputs A,R and P and outputs T.

11. An object falls to the ground from a height h in time t given by

$$t = (2h/g)^{0.5}$$

where g is the gravitational constant (= 9.81 metres/sec).

- ▷ Write a program which computes the time it would take an apple to hit Newton on the head assuming he was 1.37 metres high (when sitting) and the apple was on a branch 6.7 metres high.
- ▷ An computer with an Intel Core i7 processor can execute 117160 MIPS (million instruction per second). Write a program which outputs the number of instructions that it can execute during the time it takes for an egg to drop to the floor from a table 1 metre high.
- ▷ In order to payoff in N years a mortgage of P on which interest is charged at an annual rate of R% and computed annually, A must be repaid every year where :

$$A = \frac{P \times (1+r)^N \times r}{(1+r)^N - 1}$$

and $r = \frac{R}{100}$.
Write a program which reads P, N and R and outputs A.

12. A room is B metres wide, L metres long and H metres high. It has a door ($B1$ metres wide and $H1$ metres high) in one wall and a window ($B2$ metres wide and $H2$ metres high) in another. Wallpaper is available in rolls M metres long and F metres wide. Write a program which reads values for $B, L, H, Bl, HI, B2, H2, M$ and F and calculates how many rolls of wallpaper would be needed to paper the walls assuming no waste.

13. A farmer has a field B metres wide, L metres long.

    The field yields C cubic metres of grain per hectare (1 hectare = 10000 square metres). The farmer has a number of cylindrical grain silos, R metres in radius, H metres in height in which he stores the harvest. Write a program which reads B, L, C, R, H and outputs the number of completely filled silos and the height of grain in any unfilled silo

14. Write a program which reads the coordinates of the vertices of a triangle and outputs the area of the triangle.

15. Write a program that inputs the the cost of a customer's purchases and the amount of money paid by the customer and prints the change due to the customer. It also prints the number of each note (R200, R100, R50, R20, R10) and the number of coins (R5, R2, R1, 50c, 20c, 10c, 5c) that have to be returned to the customer.

Chapter

3 Selection

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

- Martin Golding

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

- C.A.R. Hoare

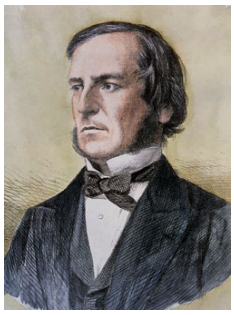


Figure 3.1: *George Boole (1815-1864): British Logician who invented an algebra of logic.*

3.1 Boolean Values and Boolean Expressions

One of the building blocks of algorithms are **boolean expressions**. Boolean expressions are used to *test* values and variables. They ask true/false questions: for example `x >= 0` is a boolean expression that asks (tests) whether the value referred to by the variable `x` is greater than or equal to `0`. The answer is either `True` or `False` depending on the value of `x`.

An expression is a sequence of symbols that evaluates to some value. We

There are two kinds of expressions we use in Python: boolean expressions and arithmetic expressions.

have encountered **arithmetic expressions** that are composed of literals, variables and arithmetic operators that evaluate to numeric values. For example `(2 + 3)* 2` is an arithmetic expression that evaluates to `10`. Boolean expressions are composed of literals, variables, arithmetic operators, **comparison operators**, and **logical operators**.

Arithmetic expressions evaluate to numbers and boolean expressions evaluate to True or False.

A boolean expression evaluates to one of two special values: `True` or `False`. The Python type for storing true and false values is called `bool`, named after the British mathematician, George Boole.There are only two **boolean values**. They are `True` and `False`. Capitalization is important, since `true` and `false` are not boolean values (remember Python is case sensitive).

```
print(True)
print(type(True))
print(type(False))
```

> **Note**
>
> Boolean values are not strings!
>
> It is extremely important to realize that True and False are not strings. They are **not** surrounded by quotes. They are the only two values in the data type `bool`. Take a close look at the types shown below.

```
print(type(True))
print(type("True"))
```

`bool` is another Python data type. We have already seen `int`, `float`, and `str` (string).

A **boolean expression** is an expression that evaluates to a boolean value i.e. `True` or `False`. The equality operator, `==`, compares two values and produces a boolean value related to whether the two values are equal to one another.

```
print(5 == 5)
print(5 == 6)
```

In the first statement, the two operands are equal, so the expression evaluates to `True`. In the second statement, 5 is not equal to 6, so we get `False`.

The `==` operator is one of six common **comparison operators**; the others are:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single

equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. Also, there is no such thing as =< or =>.

Note too that an equality test is symmetric, but assignment is not. For example, if `a == 7` then `7 == a`. But in Python, the statement `a = 7` is legal and `7 = a` is not. (Can you explain why?)

3.2 Logical operators

An operator is a special function that takes 1,2 or more inputs (operands) and produces an output value. The arithmentic operators include `+,-,/ *`. E.g. + takes two numbers as input and produces a number as output i.e. the number that results when the two input numbers are added together. Logical operators take take 1 or 2 inputs (the inputs are boolean expressions) and produces a boolean value (True or False).

There are three **logical operators**: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example, `x > 0 and x < 10` is true only if `x` is greater than 0 *and* at the same time, x is less than 10. How would you describe this in words? You would say that x is between 0 and 10, not including the endpoints.

There are many other logical operators that are not part of Python. A course on Logic will cover these.

`n % 2 == 0 or n % 3 == 0` is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* divisible by 3. In this case, one, or the other, or both of the parts has to be true for the result to be true.

Finally, the `not` operator negates a boolean expression, so `not x > y` is true if `x > y` is false, that is, if `x` is less than or equal to `y`.

```
1 x = 5
2 print(x > 0 and x < 10)
3
4 n = 25
5 print(n % 2 == 0 or n % 3 == 0)
```

Common Mistake!

There is a very common mistake that occurs when programmers try to write boolean expressions. For example, what if we have a variable `number` and we want to check to see if its value is 5,6, or 7. In words we might say: "number equal to 5 or 6 or 7". However, if we translate this into Python, `number == 5 or 6 or 7`, it will not be correct. The `or` operator must join the results of three equality checks. The correct way to write this is `number == 5 or number == 6 or number == 7`. This may seem like a

lot of typing but it is absolutely necessary. You cannot take a shortcut.

Operators and Truth Tables

To better understand the logical operators, we show their meaning with **truth tables**. A truth table is constructed by considering all possible assignments of `True` and `False` to each operand of the operator and in each case calculating the resulting evaluation of the whole statement.

Before we can construct a truth table for any particular expression, we must define truth tables for each operator.

The truth table for `not` is:

p	*not p*
`True`	`False`
`False`	`False`

and a truth table for `or` and `and` is:

p	*q*	*p or q*	*p and q*
`True`	`True`	`True`	`True`
`True`	`False`	`True`	`False`
`False`	`True`	`True`	`False`
`False`	`False`	`False`	`False`

Now we can construct truth tables for expressions. For example the truth table for:

(not p) or (not q)

There are two possibilities for p and two for q, so the truth table has 4 rows.

p	*q*	*not p*	*not q*	*not p or not q*
`True`	`True`	`False`	`False`	`False`
`True`	`False`	`False`	`True`	`True`
`False`	`True`	`True`	`False`	`True`
`False`	`False`	`True`	`True`	`True`

If there are n propositions in a statement, then its truth table has 2^n rows. The order of the rows does not matter, but it is easier to make sure one has them all by writing them down in a specific order!

3.3 Precedence of Operators

We have now added a number of additional operators to those we learned in the previous chapter. It is important to understand how these operators relate to the others with respect to operator precedence . Python will always evaluate the arithmetic operators first exponentiation (`**`) is highest, then multiplication/division, then addition/subtraction). Next comes the comparison operators. Finally, the logical operators are done last. This means that the expression `x*5 >= 10 and y-6 <= 20` will be evaluated so as to first perform the arithmetic and then check the relationships. The `and` will be done last. Although many programmers might place parenthesis around the two relational expressions, it is not necessary.

These precedence rules apply to boolean expressions.

The following table summarizes the operator precedence from highest to lowest. A complete table for the entire language can be found in the documentation.

Level	Category	Operators
7(high)	exponent	`**`
6	multiplication	`* / // %`
5	addition	`+ -`
4	relational	`== != = <= >= < >`
3	logical	`not`
2	logical	`and`
1(low)	logical	`or`

3.4 Conditional Execution: Binary Selection

Recall that a control structure allows us to specify the order of execution of the instructions in a program. There are three such control structures and they may be combined.

1. **Sequence** specifies that the instructions must be carried out in sequence–one after the other. In Python we specify this by writing the instructions one after another.

2. **Repetition** (also known as **Iteration**) specifies that a set of instructions should be repeated.

3. **Selection** or **Branching**. Selection allows us to choose different paths through the program based on conditions.

This chapter discusses the Selection control structure. It is also called branching since we reach a point in the program where we choose to execute one set of statements or another set of statements depending on certain conditions–its as if

we reach a branch in the road and we choose which one to take. We write conditions using boolean expressions: if the expression evaluates to `True` we execute one set of instructions otherwise we execute the other set.

binary means two.

The simplest form of selection is the **if statement**. This is sometimes referred to as **binary selection** since there are two possible paths of execution.

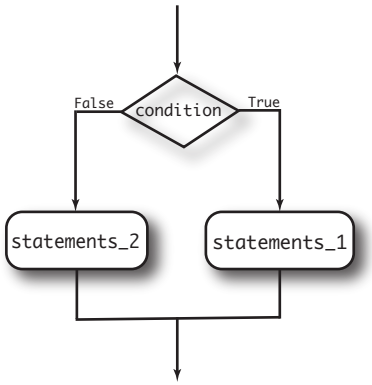


Figure 3.2: *Flowchart of a* ***if*** *statement with an* ***else***

```
x = 15

if x % 2 == 0:
    print(x, "is even")
else:
    print(x, "is odd")
```

The syntax for an `if` statement looks like this:

```
if BOOLEAN EXPRESSION:
    STATEMENTS_1        # executed if condition evaluates to True
else:
    STATEMENTS_2        # executed if condition evaluates to
        False
```

The boolean expression after the `if` statement is called the **condition**. If it is true, then the indented statements (`STATEMENTS_1`) get executed. If not, then the statements indented under the else clause (`STATEMENTS_1`) get executed.

The first line of the selection structure begins with the keyword `if` followed by a *boolean expression* and ends with a colon (:). The indented statements that follow are called a **block**. The first unindented statement marks the end of the block.

In Python a block is contiguous statements indented by the same amount away from the margin.

Each of the statements inside the first block of statements is executed in sequence if the boolean expression evaluates to `True`. The entire first block of statements is skipped if the boolean expression evaluates to `False`, and instead all the statements under the `else` clause are executed.

There is no limit on the number of statements that can appear under the two clauses of an `if` statement, but there has to be at least one statement in each block.

3.5 Omitting the else Clause: Unary Selection

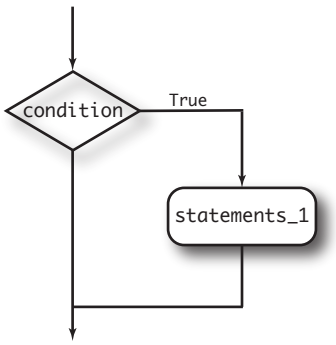


Figure 3.3: *Flowchart of an* ***if*** *with no* ***else***

Another form of the `if` statement is one in which the `else` part (or clause) is omitted entirely. This creates what is sometimes called **unary selection**.

In this case, when the condition evaluates to `True`, the statements are executed. Otherwise the flow of execution continues to the statement after the body of the `if`. Basically we instruct the computer: "Do this only if the condition is true" otherwise move on with the rest of the program".

```
1 x = 10
2 if x < 0:
3     print("The negative number ",  x, " is not valid here.")
4 print("This is always printed")
```

What would be printed if the value of `x` is negative? Try it.

3.6 Nested conditionals

One conditional can also be **nested** within another. The following pattern of selection shows how we might decide how two integer variables, `x` and `y` are related to each other.

The outer conditional contains two branches. The second branch (the else from the outer) contains another `if` statement, which has two branches of its own. Those two branches could contain conditional statements as well.

```
if x < y:
    print("x is less than y")
else:
    if x > y:
        print("x is greater than y")
    else:
        print("x and y must be equal")
```

The flow of control for this example can be seen in this flowchart illustration.

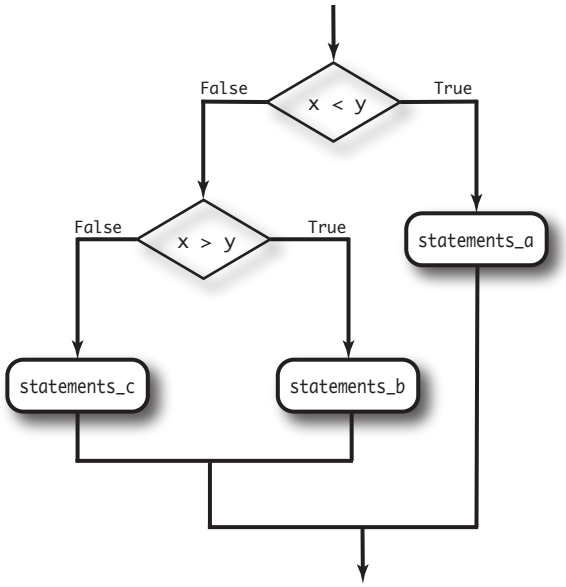


Figure 3.4: *Nested Selection Example*

Here is a complete program that defines values for `x` and `y`. Run the program and see the result. Then change the values of the variables to change the flow of control.

```
x = 10
y = 10

if x < y:
    print("x is less than y")
else:
    if x > y:
        print("x is greater than y")
    else:
        print("x and y must be equal")
```

Note

In some programming languages, matching the `if` and the `else` is a problem. However, in Python this is not the case. The indentation pattern tells us exactly which else belongs to which if.

3.7 Chained conditionals

Python provides an alternative way to write nested selection such as the one shown in the previous section. This is sometimes referred to as a **chained conditional**

```
1 if x < y:
2     print("x is less than y")
3 elif x > y:
4     print("x is greater than y")
5 else:
6     print("x and y must be equal")
```

The flow of control can be drawn in a different orientation but the resulting pattern is identical to the one shown above.

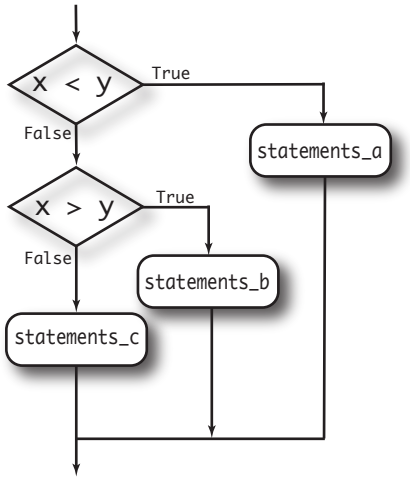


Figure 3.5: *Flowchart of a Chained Conditional*

`elif` is an abbreviation of `else if`. Again, exactly one branch will be executed. There is no limit of the number of `elif` statements but only a single (and optional) final `else` statement is allowed and it must be the last branch in the statement.

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the

statement ends. Even if more than one condition is true, only the first true branch executes.

Here is the same program using `elif`.

```
1 x = 10
2 y = 10
3
4 if x < y:
5     print("x is less than y")
6 elif x > y:
7     print("x is greater than y")
8 else:
9     print("x and y must be equal")
```

3.8 Exercises

1. Give the **logical opposites** of these conditions. You are not allowed to use the `not` operator.

   a) `a > b`
   b) `a >= b`
   c) `a >= 18 and day == 3`
   d) `a >= 18 or day != 3`

2. Write a program which inputs 3 numbers and outputs the value of the smallest.
3. The post office charges parcel-senders according to the weight of their parcel. For a parcel weighing 2 kilograms or less the charge is R 125. For each kilogram or part of a kilogram above 2 there is an additional charge of R 55 dollars.
   Write a program which inputs the weight of a parcel and outputs the amount the sender is charged.
4. A bus company has the following charges for a tour.
   If a person buys less than 5 tickets they cost R 1 500 each, otherwise they cost R 1250 each. Write a program which calculates a customers bill given the number of tickets.
5. Write a program which reads the co-ordinates of a point, the co-ordinates of the centre of a circle and the radius of the circle, and determines whether or not the point is inside the circle.
6. Write a program which reads the co-ordinates of three points and determines whether or not they lie on a straight line.

7. Write a program which reads in the 3 coefficients of a quadratic equation and determines whether it has (a) no real roots (b) a single real root - if so, prints it (c) 2 real roots - if so prints them.
8. A baby sitter charges R 25 an hour between 18:00 and 21:30 and R 35 an hour between 21:30 and midnight. She will not sit before 18:00 or after midnight. Write a program which reads the times at which she started and finished sitting and calculates how much she earned. Your program should check for invalid starting and finishing times.
9. A student sits three examinations.

   (i) He is awarded a pass if he scores at least 5O in each of the examinations.

   (ii) He is awarded supplementary exams in all three examinations if he passes in two, the average of the three marks is at least 50 and the lowest of the three marks is at least 40.

   (iii) He fails if neither (i) nor (ii) applies.

   Write a program which inputs the three marks and outputs either PASS or SUPP GRANTED or FAIL.
10. Write a program that will be given the length of two sides of a right-angled triangle and it should return the length of the hypotenuse. (Hint: `x ** 0.5` will return the square root, or use `sqrt` from the math module)
11. Write a program which, given the length of three sides of a triangle, will determine whether the triangle is right-angled.
    Hint: floating point arithmetic is not always exactly accurate, so it is not safe to test floating point numbers for equality. If a good programmer wants to know whether `x` is equal or close enough to `y`, they would probably code it up as

    ```
    1 if  abs(x - y) < 0.001:      # if x is approximately equal
          to y
    2     ...
    ```

12. A year is a **leap year** if it is divisible by 4 unless it is a century that is not divisible by 400. Write a function that takes a year as a parameter and returns `True` if the year is a leap year, `False` otherwise.
13. Write a program that reads in three positive integers representing day, month and year and indicate whether or not these form a legitimate date.

Chapter

4 Iteration

Perfection [in design] is achieved, not when there is nothing more to add, but when there is nothing left to take away.

- Antoine de Saint-Exupéry

Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little.

— Bertrand Meyer

Figure 4.1: *Ada, Countess of Lovelace: British mathematician and the world's first programmer.*

Iteration is also known as looping or repetition.

Sequence, selection and iteration are three control structures provided by all programming languages. In this chapter we discuss several language features that Python provides for specifying iteration. Iteration is also known as **repetition** or **looping**.

Repeated execution of a sequence of statements is called **iteration**. There are two main structures for iteration in Python. The `for` statement is a very common form of iteration in Python. The `while` statement is another way to have your program do iteration.

4.1 The `for` loop

A basic building block of all programs is to be able to repeat some code over and over again. In computer science, we refer to this repetitive idea as **iteration**. In this section, we will explore some mechanisms for basic iteration.

In Python, the **for** statement allows us to write programs that implement iteration. As a simple example, let's say we have some friends, and we'd like to send them each a message inviting them to our party. The following code will just print a message for each friend.

```
1 for name in ["Joe", "Amy", "Brad", "Zuki", "Thandi", "Paris"]:
2     print("Hi", name, "Please come to my party on Saturday!")
```

Take a look at the output produced when you run the program. There is one line printed for each friend. Here's how it works:

- **name** in this `for` statement is called the **loop variable**.
- The list of names in the square brackets is called a Python **list**. Lists are very useful. We will have much more to say about them later.
- Line 2 is the **loop body**. The loop body is always indented. The indentation determines exactly what statements are "in the loop". The loop body is performed one time for each name in the list.
- On each *iteration* or *pass* of the loop, first a check is done to see if there are still more items to be processed. If there are none left (this is called the **terminating condition** of the loop), the loop has finished. Program execution continues at the next statement after the loop body.
- If there are items still to be processed, the loop variable is updated to refer to the next item in the list. This means, in this case, that the loop body is executed here 6 times, and each time `name` will refer to a different friend.
- At the end of each execution of the body of the loop, Python returns to the `for` statement, to see if there are more items to be handled.

So the `for` loop processes each item in a list. Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed.

Flow of Execution of the for Loop

As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the **control flow**, or the **flow of execution** of the program. When humans execute programs, they often use their finger to point to each statement in turn. So you could think of control flow as "Python's moving finger".

Control flow until now has been strictly top to bottom, one statement at a time. We call this type of control **sequential**. Sequential flow of control is always

assumed to be the default behavior for a computer program. The for statement changes this.

Flow of control is often easy to visualize and understand if we draw a flowchart. This flowchart shows the exact steps and logic of how the for statement executes.

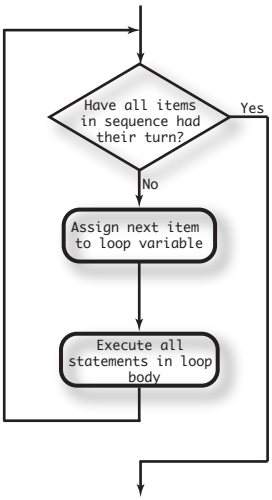


Figure 4.2: *Flowchart for Iteration*

The range Function

In the example below we use a list of four integers to control the iteration i.e. we cause the statements to be executed four times. We said that we could have used any four values.

```
1 for i in [0, 1, 2, 3]:   # repeat four times
2     print(i * i)         # square each number
```

It turns out that generating lists with a specific number of integers is a very common thing to do, especially when you want to write simple for loop controlled iteration. Even though you can use any four items, or any four integers for that matter, the conventional thing to do is to use a list of integers starting with 0. In fact, these lists are so popular that Python gives us a special built-in range function that can deliver a sequence of values to the for loop. The sequence provided by range always starts with 0.

```
1 for i in range(4):
2     # Executes the body with i = 0, then 1, then 2, then 3
3 for x in range(10):
4     # sets x to each of ... [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If you ask for `range(4)`, then you will get 4 values starting with 0. In other words, 0, 1, 2, and finally 3. Notice that 4 is not included since we started with 0. Likewise, `range(10)` provides 10 values, 0 through 9.

> **Note**
> Computer scientists like to count from 0!

So to repeat something four times, a good Python programmer would do this:

```
for i in range(4):
    print(i * i)
```

The range function (see the documentation) is actually a very powerful function when it comes to creating sequences of integers. It can take one, two, or three parameters. We have seen the simplest case of one parameter such as `range(4)` which creates `[0, 1, 2, 3]`. But what if we really want to have the sequence `[1, 2, 3, 4]`? We can do this by using a two parameter version of `range` where the first parameter is the starting point and the second parameter is the ending point. The evaluation of `range(1,5)` produces the desired sequence. What happened to the 5? In this case we interpret the parameters of the range function to mean range(start,stop+1).

> **Note**
> Why in the world would range not just work like `range(start, stop)`? Think about it like this. Because computer scientists like to start counting at 0 instead of 1, `range(N)` produces a sequence of things that is N long, but the consequence of this is that the final number of the sequence is N-1. In the case of start, stop it helps to simply think that the sequence begins with start and continues as long as the number is less than stop.

Here are a two examples for you to run. Try them and then add another line below to create a sequence starting at 10 and going up to 20 (including 20).

```
print(range(4))
print(range(1, 5))
```

In this example, the variable `i` will take on values produced by the `range` function.

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

Finally, suppose we want to have a sequence of even numbers. How would we do that? Easy, we add another parameter, a step, that tells range what to count

by. For even numbers we want to start at 0 and count by 2's. So if we wanted the first 10 even numbers we would use `range(0,19,2)`. The most general form of the range is `range(start, stop, step)`. You can also create a sequence of numbers that starts big and gets smaller by using a negative value for the step parameter.

```
print(range(0, 19, 2))
print(range(0, 20, 2))
print(range(10, 0, -1))
```

Accumulators

Iteration can be paired with the update idea to form the accumulator pattern. For example, to compute the sum of the first n integers, we could create a `for` loop using the `range` function to produce the numbers 1 through n. Using the accumulator pattern, we can start with a running total variable initialized to 0 and on each iteration, add the current value of the loop variable. A function to compute this sum is shown below.

To accumulate is to gather together or acquire an increasing number or quantity of something.

The running total variable is the accumulator, theSum in the example.

```
theSum = 0
aBound = 100

for aNumber in range(1, aBound + 1):
    theSum = theSum + aNumber

print(theSum)
```

To review, the variable `theSum` is called the accumulator. It is initialized to zero before we start the loop. The loop variable, `aNumber` will take on the values produced by the `range(1, aBound + 1)` function call. Note that this produces all the integers from 1 up to the value of `aBound`. If we had not added 1 to `aBound`, the range would have stopped one value short since `range` does not include the upper bound.

The assignment statement, `theSum = theSum + aNumber`, updates `theSum` each time through the loop. This accumulates the running total. Finally, we print the value of the accumulator.

4.2 The while Statement

There is another Python statement that can also be used to build an iteration. It is called the `while` statement. The `while` statement provides a much more general mechanism for iterating. Similar to the `if` statement, it uses a boolean expression to control the flow of execution. The body of `while` will be repeated as long as the controlling boolean expression evaluates to `True`.

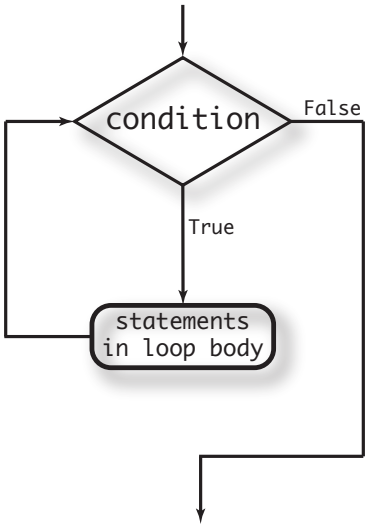


Figure 4.3: *Flowchart for the* `while` *statement*

The following figure shows the flow of control.

We can use the `while` loop to create any type of iteration we wish, including anything that we have previously done with a `for` loop. For example, the program in the previous section could be rewritten using `while`. Instead of relying on the `range` function to produce the numbers for our summation, we will need to produce them ourselves. To to this, we will create a variable called `aNumber` and initialize it to 1, the first number in the summation. Every iteration will add `aNumber` to the running total until all the values have been used. In order to control the iteration, we must create a boolean expression that evaluates to `True` as long as we want to keep adding values to our running total. In this case, as long as `aNumber` is less than or equal to the bound, we should keep going.

Here is a version of the summation program that uses a while statement.

```
1 theSum  = 0
2 aNumber = 1
3 aBound = 10
4 while aNumber <= aBound:
5       theSum = theSum + aNumber
6       aNumber = aNumber + 1
7
8 print(theSum)
```

The statements inside the while loop (i.e. the statements that are repeated) are known as the **body of the loop**. The boolean expression that controls the loop is also known as the **loop condition**.

You can almost read the `while` statement as if it were in natural language. It means, while `aNumber` is less than or equal to `aBound`, continue executing the body

of the loop. Within the body, each time, update `theSum` using the accumulator pattern and increment `aNumber`. After the body of the loop, we go back up to the condition of the `while` and reevaluate it. When `aNumber` becomes greater than `aBound`, the condition fails and flow of control continues to the `return` statement.

The variable that controls the iteration (i.e. that causes the loop to terminate or controls how many times the loop executes) is called a **control variable**. In the example, the control variable is `aNumber`. Note carefully that the control variable must be updated within the body of the loop. What would happen if the update statement (line 6) was omitted?

> **Note**
> The names of the variables have been chosen to help readability.

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `False` or `True`.
2. If the condition is `False`, exit the `while` statement and continue execution at the next statement.
3. If the condition is `True`, execute each of the statements in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is `False` the first time through the loop, the statements inside the loop are never executed.

The body of the loop should change the value of one or more variables so that eventually the condition becomes `False` and the loop terminates. Otherwise the loop will repeat forever. This is called an **infinite loop**.

An endless source of amusement for computer scientists is the observation that the directions written on the back of the shampoo bottle (lather, rinse, repeat) create an infinite loop.

In the case shown above, we can prove that the loop terminates because we know that the value of `aBound` is finite, and we can see that the value of `aNumber` increments each time through the loop, so eventually it will have to exceed `aBound`. In other cases, it is not so easy to tell.

Counting digits

The following program counts the number of decimal digits in a positive integer:

```
n = 710
count = 0
while n != 0:
    count = count + 1
    n = n // 10
print(count)
```

This function demonstrates an important pattern of computation called a **counter**. The variable `count` is initialized to 0 and then incremented each time the loop body is executed. When the loop exits, `count` contains the result — the total number of times the loop body was executed, which is the same as the number of digits.

If we wanted to only count digits that are either 0 or 5, adding a conditional before incrementing the counter will do the trick:

```
n = 1055030250
count = 0
while n > 0:
    digit = n % 10
    if digit == 0 or digit == 5:
        count = count + 1
    n = n // 10
print(count)
```

What is printed if `n` is assigned 1 in the first line?

4.3 Abbreviated assignment

Incrementing a variable (adding 1 to it) and decrementing (subtracting 1 from it is so common that Python provides an abbreviated syntax for it:

```
count = 0
count += 1
print(count)
count += 1
print(count)
count += 5
print(count)
```

`count += 1` is an abreviation for `count = count + 1` . We pronounce the operator as *"plus-equals"*. The increment value does not have to be 1:

There are similar abbreviations for `-=`, `*=`, `/=`, `//=` and `%=`:

```
n = 2
n *= 5
```

```
print(n)

n -= 4
print(n)

n //= 2
print(n)

n %= 2
print(n)
```

4.4 Infinite, Definite and Indefinite Iteration

Infinite Loops

An **infinite loop** is a loop structure that theoretically never terminates. This kind of loop causes your program to **hang** and you will notice that it continues running or runs for an extremely long time and does not produce any output or may cause your computer to crash. Infinite loops are usually due to programmer error. A common error is to forget to update the loop control variable.

```
# CAUTION: Infinite loop
theSum  = 0
aNumber = 1
aBound = 10
while aNumber <= aBound:
      theSum = theSum + aNumber

print(theSum)
```

Here we see that `aNumber` never changes and the loop condition will always be true.

Its not enough to check that you update the loop control variable within the body of the loop. The update must ensure that the loop condition will eventually become false. For example if your update statement was `aNumber = aNumber - 1` then notice that `aNumber` will always be less than `aBound` and will therefore never become false and you still have an infinite loop.

Indefinite and Definite Iteration

The `for` statement will always iterate through a sequence of values like the list of names for the party or the list of numbers created by `range`. Since we know that it will iterate once for each value in the collection, it is often said that a `for` loop creates a **definite iteration** because we definitely know how many times we are going to iterate.

`while` loops may also be used to provide definite iteration. The example above shows a definite iteration using the while loop. It is known how many times the loop will run.

```
1 theSum  = 0
2 aNumber = int(input('Enter a value: '))
3 aBound = 10
4 while aNumber <= aBound:
5       theSum = theSum + aNumber
6       aNumber = aNumber + 1
7 print(theSum)
```

We cannot know what value will be input by the user but when the while loop is executed we can tell how many times the loop will execute.

The `while` statement may also be used to construct **indefinite iteration**. Indefinite iteration simply means that we don't know how many times we will repeat but eventually the condition controlling the iteration will fail and the iteration will stop. (Unless we have an infinite loop which is of course a problem).

```
1 kmOrMiles = input("Enter K or M: ")
2
3 while kmOrMiles != 'K' and kmOrMiles != 'M':
4     kmOrMiles = input("Enter K or M: ")
```

This kind of loop is used to do input validation. It checks that the input provided by the user is valid. In this example, the user must enter either `K` or `M`. The while loop will run until the user enters one of these. In this case we cannot know how many times the loop runs — it depends on how many times the luser types incorrect values.

What you will notice here is that the `while` loop is more work for you — the programmer — than the equivalent `for` loop. When using a `while` loop you have to control the loop variable yourself. You give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates.

The 3n + 1 Sequence

As another example of indefinite iteration, let's look at a sequence that has fascinated mathematicians for many years. The rule for creating the sequence is to start from some given number, call it `n`, and to generate the next term of the sequence from `n`, either by halving `n`, whenever `n` is even, or else by multiplying it by three and adding 1 when it is odd. The sequence terminates when `n` reaches 1.

This Python function captures that algorithm. Try running this program several times supplying different values for n.

```
print(" Print the 3n+1 sequence from n, terminating when it
     reaches 1.")
n = 16   #you may use an input statement here
while n != 1:
    print(n)
    if n % 2 == 0:        # n is even
        n = n // 2
    else:                 # n is odd
        n = n * 3 + 1
print(n)                  # the last print is 1
```

The condition for this loop is `n != 1`. The loop will continue running until `n == 1` (which will make the condition false).

Each time through the loop, the program prints the value of `n` and then checks whether it is even or odd using the remainder operator. If it is even, the value of `n` is divided by 2 using integer division. If it is odd, the value is replaced by `n * 3 + 1`. Try some other examples.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program terminates. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even each time through the loop until it reaches 1.

You might like to have some fun and see if you can find a small starting number that needs more than a hundred steps before it terminates.

Particular values aside, the interesting question is whether we can prove that this sequence terminates for *all* values of `n`. So far, no one has been able to prove it *or* disprove it!

Think carefully about what would be needed for a proof or disproof of the hypothesis *"All positive integers will eventually converge to 1"*. With fast computers we have been able to test every integer up to very large values, and so far, they all eventually end up at 1. But this doesn't mean that there might not be some as-yet untested number which does not reduce to 1.

You'll notice that if you don't stop when you reach one, the sequence gets into its own loop: 1, 4, 2, 1, 4, 2, 1, 4, and so on. One possibility is that there might be other cycles that we just haven't found.

Choosing between `for` and `while`

Use a `for` loop if you know the maximum number of times that you'll need to execute the body. For example, if you're traversing a list of elements, or can formulate a suitable call to `range`, then choose the `for` loop.

So any problem like "iterate this weather model run for 1000 cycles", or "search this list of words", "check all integers up to 10000 to see which are prime" suggest that a `for` loop is best.

By contrast, if you are required to repeat some computation until some condition is met, as we did in this 3n + 1 problem, you'll need a `while` loop.

As we noted before, the first case is called **definite iteration** — we have some definite bounds for what is needed. The latter case is called **indefinite iteration** — we are not sure how many iterations we'll need — we cannot even establish an upper bound!

Newton's Method

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of `n`. If you start with almost any approximation, you can compute a better approximation with the following formula:

```
1 better =  1/2 * (approx + n/approx)
```

Execute this algorithm a few times using your calculator. Can you see why each iteration brings your estimate a little closer? One of the amazing properties of this particular algorithm is how quickly it converges to an accurate answer.

The following implementation of Newton's method requires two parameters. The first is the value whose square root will be approximated. The second is the number of times to iterate the calculation yielding a better result.

```
1 n = 18              #you may use an input statement here
2 howmany = 10        #you may use an input statement here
3
4 approx = 0.5 * n
5 for i in range(howmany):
6     betterapprox = 0.5 * (approx + n/approx)
7     approx = betterapprox
8
9 print(approx)
```

You may have noticed that the second and third calls to `newtonSqrt` in the previous example both returned the same value for the square root of 10. Using 10 iterations instead of 5 did not improve the the value. In general, Newton's algorithm will eventually reach a point where the new approximation is no better than the previous. At that point, we could simply stop. In other words, by repeatedly applying this formula until the better approximation gets close enough to the previous one, we can write a function for computing the square root that uses the number of iterations necessary and no more.

This implementation uses a `while` condition to execute until the approximation is no longer changing. Each time through the loop we compute a "better"

approximation using the formula described earlier. As long as the "better" is different, we try again. Step through the program and watch the approximations get closer and closer.

```
n = 18
approx = 0.5 * n
better = 0.5 * (approx + n/approx)
while better != approx:
    approx = better
    better = 0.5 * (approx + n/approx)

print(approx)
```

> **NOTE**
>
> The `while` statement shown above uses comparison of two floating point numbers in the condition. Since floating point numbers are themselves approximation of real numbers in mathematics, it is often better to compare for a result that is within some small threshold of the value you are looking for.

4.5 break and continue

The break statement

The **break** statement is used to immediately leave the body of its loop. The next statement to be executed is the first one after the body:

```
for i in [12, 16, 17, 24, 29]:
    if i % 2 == 1:  # If the number is odd
       break        #  ... immediately exit the loop
    print(i)
print("done")
```

The `continue` statement

This is a control flow statement that causes the program to immediately skip the processing of the rest of the body of the loop, *for the current iteration*. But the loop still carries on running for its remaining iterations:

```
for i in [12, 16, 17, 24, 29, 30]:
    if i % 2 == 1:      # If the number is odd
       continue         # Don't process it
    print(i)
print("done")
```

4.6 Other flavours of loops

`for` and `while` loops do their tests at the start, before executing any part of the body. They're called **pre-test** loops, because the test happens before (pre) the body. `break` and `return` are our tools for adapting this standard behaviour.

Sometimes we'd like to have the **middle-test** loop with the exit test in the middle of the body, rather than at the beginning or at the end. Or a **post-test** loop that puts its exit test as the last thing in the body. Other languages have different syntax and keywords for these different flavours, but Python just uses a combination of `while` and `if condition: break` to get the job done.

A typical example is a problem where the user has to input numbers to be summed. To indicate that there are no more inputs, the user enters a special value, often the value -1, or the empty string. This needs a middle-exit loop pattern: input the next number, then test whether to exit, or else process the number:

```
1 total = 0
2 while True:
3     response = input("Enter the next number. (Leave blank to
          end)")
4     if response == "":
5         break
6     total += int(response)
7 print("The total of the numbers you entered is ", total)
```

Convince yourself that this fits the middle-exit loop flowchart: line 3 does some useful work, lines 4 and 5 can exit the loop, and if they don't line 6 does more useful work before the next iteration starts.

The `while bool-expr:` uses the Boolean expression to determine whether to iterate again. `True` is a trivial Boolean expression, so `while True:` means *always do the loop body again*. This is a language *idiom* — a convention that most programmers will recognize immediately. Since the expression on line 2 will never terminate the loop, (it is a dummy test) the programmer must arrange to break (or return) out of the loop body elsewhere, in some other way (i.e. in lines 4 and 5 in this sample). A clever compiler or interpreter will understand that line 2 is a fake test that must always succeed, so it won't even generate a test, and our flowchart never even put the diamond-shape dummy test box at the top of the loop!

Similarly, by just moving the `if condition: break` to the end of the loop body we create a pattern for a post-test loop. Post-test loops are used when you want to be sure that the loop body always executes at least once (because the first test only happens at the end of the execution of the first loop body). This is useful, for example, if we want to play an interactive game against the user — we always want to play at least one game:

```
1 while True:
```

```
2     play_the_game_once()
3     response = input("Play again? (yes or no)")
4     if response != "yes":
5         break
6 print("Goodbye!")
```

Once you've recognized that you need a loop to repeat something, think about its terminating condition — when will I want to stop iterating? Then figure out whether you need to do the test before starting the first (and every other) iteration, or at the end of the first (and every other) iteration, or perhaps in the middle of each iteration. Interactive programs that require input from the user or read from files often need to exit their loops in the middle or at the end of an iteration, when it becomes clear that there is no more data to process, or the user doesn't want to play our game anymore.

4.7 A Guessing Game

The following program implements a simple guessing game:

```
1 import random                    # We cover random numbers in the
2 rng = random.Random()            #  modules chapter, so peek
      ahead.
3 number = rng.randrange(1, 1000) # Get random number between [1
      and 1000).
4
5 guesses = 0
6 msg = ""
7
8 while True:
9     guess = int(input(msg + "\nGuess my number between 1 and
          1000: "))
10    guesses += 1
11    if guess > number:
12        msg += str(guess) + " is too high.\n"
13    elif guess < number:
14        msg += str(guess) + " is too low.\n"
15    else:
16        break
17
18 input("\n\nGreat, you got it in {0}
      guesses!\n\n".format(guesses))
```

This program makes use of the mathematical law of **trichotomy** (given real numbers a and b, exactly one of these three must be true: a > b, a < b, or a == b).

At line 18 there is a call to the input function, but we don't do anything with the result, not even assign it to a variable. This is legal in Python. Here it has the

effect of popping up the input dialog window and waiting for the user to respond before the program terminates. Programmers often use the trick of doing some extra input at the end of a script, just to keep the window open.

Also notice the use of the `msg` variable, initially an empty string, on lines 6, 12 and 14. Each time through the loop we extend the message being displayed: this allows us to display the program's feedback right at the same place as we're asking for the next guess.

4.8 Nested Loops

These loops are the trickiest to read and write.

A **nested loop** is a loop structure in which a loop is found inside another loop i.e. the body of a loop contains another loop. For example consider the program below to print the minutes and seconds of a digital clock.

A good example of a nested loop is a digital clock.

The hour value loops through 24 hours (for a 24 hour clock) and for each hour the minute value loops 60 times. A program to simulate this is:

```
1 for hours in range(24):
2     for minutes in range(60):
3         print(hours, ':', minutes)
```

The output will be:

```
1 0:0
2 0:1
3 0:2
4 ...
5 1:0
6 1:1
7 1:2
8 ...
9 23:59
```

Notice that the hour value loops through from 0 to 23 and for each hour the minutes value loops through from 0 to 59. The outer loop runs 24 times for each of these the inner loop runs 60 times. In total, the print statement runs 24×60 times i.e. 1440 times.

It should not be surprising that we could nest a loop inside a nested loop and have a triple nested loop (or a multi-nested loop). For example one could modify the program above to also print out the seconds. In this case the outer loop runs 24 times, the inner loop 60 times and the inner-inner loop runs 60 times for time the inner loop runs.

```
for hours in range(24):
    for minutes in range(60):
        for seconds in range(60):
            print(hours, ':', minutes, ':' , seconds)
```

4.9 Tables

One of the things loops are good for is generating tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, *"This is great! We can use the computers to generate the tables, so there will be no errors."* That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium processor chip used to perform floating-point division.

Simple Tables

Although a power of 2 table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

```
print("n", '\t', "2**n")     #table column headings
print("---", '\t', "-----")

for x in range(13):        # generate values for columns
    print(x, '\t', 2 ** x)
```

The string `'\t'` represents a **tab character**. The backslash character in `'\t'` indicates the beginning of an **escape sequence**. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence `\n` represents a **newline**.

An escape sequence can appear anywhere in a string. In this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?

As characters and strings are displayed on the screen, an invisible marker called the **cursor** keeps track of where the next character will go. After a `print` function is executed, the cursor normally goes to the beginning of the next line.

In a nested loop structure such as the one below, we distinguish between 4 areas were we could write code: before the outer loop, in the outer loop but before the inner loop, in the inner loop, after the inner loop (but still in the outer loop). Ensure that you can identify each of these areas. Code in each these area is used for different purposes:

1. Before the outer loop: This code runs only once - we write code here to print the header for the table and the column labels.
2. In the outer loop but before the inner loop: This code runs every time the outer loop runs but before the inner loop runs. When we get to this point we are at the beginning of each row. We write code that we want to execute at the beginning of each row—for example to print row labels.
3. In the inner loop: Code here runs for every row and for every column. We write code here that executes as we visit every column of every row—for example to print the powers.
4. After the inner loop (but still in the outer loop): Code here runs when we have completed visiting every column in a particular row. We write code here that we want to execute at the end of every row—for example to print a newline (i.e. to go to the beginning of the next row).

```
1  # before loop
2  # print the table header
3
4  print '\t',
5  for i in range(1,5):
6          print 'x^',i, '\t'  # do not print a new line
7
8  print()                          # go to a new line
9
10 for x in range(1,11):  # outer loop: for each row
11
12         # before inner loop
13         # code here repeats for each row
14         # but before visiting the columns
15         # do something at the beginning of each row
16         print x, '\t',
17
18         for n in range(1,5):  # inner loop: for each column
19
20                 # inside inner loop
21                 # do something for each row and each column
22
23                 print x ** n,'\t' ,
24
```

The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program. Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

Two-dimensional tables

A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6.

A good way to start is to write a loop that prints the multiples of 2, all on one line:

```
1 for i in range(1, 7):
2     print(2 * i, end="   ")
3 print()
```

Here we've used the `range` function, but made it start its sequence at 1. As the loop executes, the value of `i` changes from 1 to 6. When all the elements of the range have been assigned to `i`, the loop terminates. Each time through the loop, it displays the value of `2 * i`, followed by three spaces.

The extra `end="   "` argument in the `print` function suppresses the newline, and uses three spaces instead. After the loop completes, the call to `print` at line 3 finishes the current line, and starts a new line.

Tables and nested loops

Nested loops are useful when working with tabular data (data in tables). The first example is a program that prints a table of the powers of $x^1, ..., x^n$. We want to print the following table:

	x^1	x^2	x^3	x^4
1	1	1	1	1
2	2	4	8	16
3	3	9	27	81
...	...	...	...	
10	10	100	1000	10000

The table (excluding the labels) has 4 columns and 10 rows. Each item in the table is sometimes called an *element* of the table. Study the code below and note the comments carefully.

We should think of this algorithm as a traversal of the table, i.e. it allows us to visit every element of the table. The outer loop takes us from row to row and the inner loop takes us from column to column (of each row).

```
25          # after inner loop
26          # do something at the end of each row
27          # after visiting every column in that row
28
29          print() #go to the next row
```

The general structure for working with two dimensional tables is thus:

```
1 # before loop: print headers
2
3 for each row
4      # do something at the beginning of each rows
5      for each column
6           # do something for every element
7      # do something at the end of each row
```

The "do something" does not have to involve only printing. The next example solves this problem: Print out the sums of each row of table above i.e. the sum of $x^1+x^2+x^3+x^4$ for each value of x from 1 to 10. We note that we should print the sum of powers at the end of each row. We need an accumulator variable to store the sum. So we may think that something like this will work:

```
1 sum = 0
2 for x in range (1,11):
3
4     for n in range(1,5):
5         sum = sum + (x ** n)
6
7     print('Sum of row', x, ' : ', sum)
```

You see the problem: the `sum` continues accumulating the sum of every element when we wanted only the sum of each row. So we have not reset sum. We should reset sum at the beginning of each row:

```
1 for x in range (1,11):
2     sum = 0
3     for n in range(1,5):
4         sum = sum + (x ** n)
5
6     print('Sum of row', x, ' : ', sum)
```

For our last example we want to print the following picture where the number of rows is input by the user:

```
1 *
2 **
3 ***
4 ****
5 *****
```

In problems like these we need to find patterns. In row 1 we print 1 star, row 2 we print 2 stars, row 3 we print 3 stars … So for each row, the number of stars to print depends on the row we are on:

```
numRows = input("How many rows: ")
numRows = int(numRows)

for row in range(numRows):
    for col in range (row + 1):
        print("*",end="")   #print without the new line
    print()
```

4.10 Exercises

1. Write a program that prints out the first n triangular numbers. A triangular number or triangle number counts the objects that can form an equilateral triangle, as in the diagram below. The n^{th} triangular number is the number of dots composing a triangle with n dots on a side. Your program should produce the following output for $n = 5$:

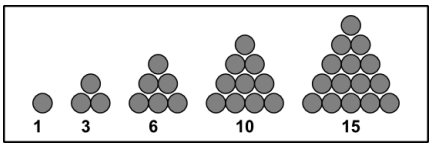


Figure 4.4: *Triangular Numbers*

```
1 1       1
2 2       3
3 3       6
4 4       10
5 5       15
```

2. Write a program which inputs N followed by the prices of N items. The program should output the price at which each item is offered in a sale. The sale price is calculated as follows. The original price is reduced by 10%, the resulting quantity is rounded to the nearest Rand and then 1 cent subtracted from it. If the resulting quantity is less than the original price then the new price is output, otherwise the old price is output together with a warning message. For e.g. if the original price is R5,69 then less 10% is R5,121. Rounded to the nearest Rand gives R5. Subtract 1c gives R4,99.

3. Write a program which reads N followed by N numbers and outputs the positive difference between the two largest numbers.

4. Write a program which inputs N followed by a set of N numbers and outputs the variance of the set given by

$$V = \frac{1}{N-1}(\sum_{i}^{N} x_i^2 - 2.A\sum_{i}^{N} x_i + N.A^2)$$

where N is the average of the set.

5. Write a program to evaluate the following series:

$$\sum_{i=1}^{100} \frac{1}{i^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{100^2}$$

6. Write a program which given the day of the week on which the first of a month falls, together with the number of days in the month, prints out a calendar for the month in the form shown below.

```
1
2 Mo   Tu   We   Th   Fr   Sa   Su
3            1    2    3    4    5
4  6    7    8    9   10   11   12
5 13   14   15   16   17   18   19
6 20   21   22   23   24   25   26
7 27   28   29   30
```

7. If x_n is a close approximation to the square root of A then a better one is x_{n+1} given by:

$$x_{n+1} = \frac{1}{2}(x_n + \frac{A}{x_n})$$

Write a program which reads some positive quantity A and calculates its square root to an accuracy which ensures that the result squared and A differ by at most 10^{-6}.

8. The Fibonacci numbers are F_0, F_1,... are defined as follows:

$$F_0 = 0 F_1 = 1 F_{n+2} = F_{n+1} + F_n \text{ for all} n \geq 0.$$

Thus the first 8 Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, \ldots$.

   a) Write a program that reads a positive integer M and prints the first Fibonacci number which is not less than M.

b) Write a program which reads an integer N and determines whether or not it is a Fibonacci number.

9. Consider

$$1^n + 6^n + 7^n + 17^n + 18^n + 23^n = 2^n + 3^n + 11^n + 13^n + 21^n + 22^n +$$

What is the first value of n for which the equation fails to hold?

10. Write a program which reads a sequence of positive integers terminated by -1 and outputs a sequence in which sub-sequences of repeated integers are replaced by a single instance of the integer preceded by a count of the integer and *. For example if the input contained

```
4  3  3  3 3  2  5  5  5  9 ...
```

the corresponding section of the output should be

```
4 4*3 2 3*5 9 ...
```

11. Write a program which reads in the three positive integer coefficients in the equation

$$ax^2 + by^2 = c$$

and outputs a solution (a and b are non-negative integers) if any exists.

12. The number of distinct ways in which r objects can be selected from n distinct objects is given by

$$nCr = \frac{n!}{(n-r)! \times r!}$$

where $x!$ is the factorial of x.

Write a program which reads n and r and outputs nCr.

13. In a certain ice skating event, a mark is awarded by each of N judges. The mark awarded to a competitor is obtained by ignoring the highest and lowest of the N marks and averaging the remainder. Write a program which reads N followed by the N marks and outputs the mark awarded.

14. Write a program which finds all the prime numbers between 1 and 1000.

15. Write a program which finds a four digit number $AABB$ which is a perfect square. A and B represent different digits.

16. Write a program which finds a four digit perfect square where the number represented by the first two digits and the number represented by the last two digits are both perfect squares.

17. Twin primes are consecutive odd numbers both of which are prime numbers. Write a program which inputs two positive integers A and B and outputs all the twin primes in the range A to B.

18. Write, for each of the series below, a program which reads N and x and outputs the sum of the first N terms.

    a)
    $$X = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots$$

    b)
    $$f(x) = \frac{X^2}{2} + \frac{x^4}{3} + \frac{x^6}{4} + \ldots$$

    c)
    $$X = \frac{1}{2!} - \frac{1}{3!} + \frac{1}{4!} - \frac{1}{5!} + \ldots$$

Chapter

5 Debugging

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

- Brian W. Kernighan

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

Edsger Dijkstra

Figure 5.1: *Linus Torvalds. Creator of the Linux operating System.*

5.1 Introduction

One of the most important skills a programmer needs is the ability to debug your programs. Debugging is a skill that you need to master over time, and some of the tips and tricks are specific to different aspects of Python programming. In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

Programming is an odd thing in a way. Here is why. As programmers we spend 99% of our time trying to get our program to work. We struggle, we stress, we spend hours deep in frustration trying to get our program to execute correctly. Then when we do get it going we celebrate, hand it in, and move on to the next homework assignment or programming task. But here is the secret, when you are successful, you are happy, your brain releases a bit of chemical that makes you feel good. You need to organize your programming so that you have lots of little successess. It turns out your brain doesn't care all that much if you have successfully written hello world, or a fast fourier transform (trust me its hard) you still get that little release that makes you happy. When you are happy you want to go on and solve the next little problem. Essentially I'm telling you once again, start small, get something small working, and then add to it.

5.2 Tracing a program

To write effective computer programs, and to build a good conceptual model of program execution, a programmer needs to develop the ability to **trace** the execution of a computer program. Tracing involves becoming the computer and following the flow of execution through a sample program run, recording the state of all variables and any output the program generates after each instruction is executed.

To understand this process, let's trace the following program we have seen already:

```
1 print(" Print the 3n+1 sequence from n, terminating when it
      reaches 1.")
2 n = 3   #you may use an input statement here
3 while n != 1:
4     print(n)
5     if n % 2 == 0:        # n is even
6         n = n // 2
7     else:                 # n is odd
8         n = n * 3 + 1
9 print(n)                  # the last print is 1
```

At the start of the trace, we have a variable, `n`with an initial value of 3. Since 3 is not equal to 1, the `while` loop body is executed. 3 is printed and `3 % 2 == 0` is evaluated. Since it evaluates to `False`, the `else` branch is executed and `3 * 3 + 1` is evaluated and assigned to `n`.

To keep track of all this as you hand trace a program, make a column heading on a piece of paper for each variable created as the program runs and another one for output. Our trace so far would look something like this:

```
n               output printed so far
--              ---------------------
3               3,
10
```

Since `10 != 1` evaluates to `True`, the loop body is again executed, and 10 is printed. `10 % 2 == 0` is true, so the `if` branch is executed and `n` becomes 5. By the end of the trace we have:

```
n               output printed so far
--              ---------------------
3               3,
10              3, 10,
5               3, 10, 5,
16              3, 10, 5, 16,
8               3, 10, 5, 16, 8,
4               3, 10, 5, 16, 8, 4,
2               3, 10, 5, 16, 8, 4, 2,
1               3, 10, 5, 16, 8, 4, 2, 1.
```

Tracing can be a bit tedious and error prone (that's why we get computers to do this stuff in the first place!), but it is an essential skill for a programmer to have. From this trace we can learn a lot about the way our code works. We can observe that as soon as `n` becomes a power of 2, for example, the program will require $\log_2(n)$ executions of the loop body to complete. We can also see that the final 1 will not be printed as output within the body of the loop, which is why we put the special `print` function at the end.

Tracing a program is, of course, related to single-stepping through your code and being able to inspect the variables. Using the computer to **single-step** for you is less error prone and more convenient. Also, as your programs get more complex, they might execute many millions of steps before they get to the code that you're really interested in, so manual tracing becomes impossible. Being able to set a **breakpoint** where you need one is far more powerful. So we strongly encourage you to invest time in learning using to use your programming environment to full effect.

5.3 Errors

Programming is a complex process. Since it is done by human beings, errors may often occur. Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

It is useful to distinguish between them in order to track them down more quickly.

Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without problems. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit. You will not be able to complete the execution your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. However, as you gain experience, you will make fewer errors and you will also be able to find your errors faster.

Runtime Errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

Semantic Errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages. However, your program will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

5.4 How to Avoid Debugging

Perhaps the most important lesson in debugging is that it is **largely avoidable** – if you work carefully.

1. **Start Small** This is probably the single biggest piece of advice for programmers at every level. Of course its tempting to sit down and crank out an entire program at once. But, when the program – inevitably – does not

work then you have a myriad of options for things that might be wrong. Where to start? Where to look first? How to figure out what went wrong? I'll get to that in the next section. So, start with something really small. Maybe just two lines and then make sure that runs ok. Hitting the run button is quick and easy, and gives you immediate feedback about whether what you have just done is ok or not. Another immediate benefit of having something small working is that you have something to turn in. Turning in a small, incomplete program, is almost always better than nothing.

2. **Keep it working** Once you have a small part of your program working the next step is to figure out something small to add to it. If you keep adding small pieces of the program one at a time, it is much easier to figure out what went wrong, as it is most likely that the problem is going to be in the new code you have just added. Less new code means its easier to figure out where the problem is.

This notion of **Get something working and keep it working** is a mantra that you can repeat throughout your career as a programmer. It's a great way to avoid the frustrations mentioned above. Think of it this way. Every time you have a little success, your brain releases a tiny bit of chemical that makes you happy. So, you can keep yourself happy and make programming more enjoyable by creating lots of small victories for yourself.

Ok, let's look at an example. Let's solve the problem posed in question 3 at the end of the Simple Python Data chapter. Ask the user for the time now (in hours 0 - 23), and ask for the number of hours to wait. Your program should output what the time will be on the clock when the alarm goes off.

So, where to start? The problem requires two pieces of input from the user, so let's start there and make sure we can get the data we need.

```
1 current_time = input("what is the current time (in hours)?")
2 wait_time = input("How many hours do you want to wait")
3
4 print(current_time)
5 print(wait_time)
```

So far so good. Now let's take the next step. We need to figure out what the time will be after waiting `wait_time` number of hours. A good first approximation to that is to simply add `wait_time` to `current_time` and print out the result. So lets try that.

```
1 current_time = input("Current time (in hours 0 - 23)?")
2 wait_time = input("How many hours do you want to wait")
3 print(current_time)
4 print(wait_time)
5 final_time = current_time + wait_time
6 print(final_time)
```

Hmm, when you run that example you see that something funny has happened.

This error was probably pretty simple to spot, because we printed out the value of `final_time` and it is easy to see that the numbers were just concatenated together rather than added. So what do we do about the problem? We will need to convert both `current_time` and `wait_time` to `int`. At this stage of your programming development, it can be a good idea to include the type of the variable in the variable name itself. So let's look at another iteration of the program that does that, and the conversion to integer.

```
1 current_time_str = input("What is the current time (in hours
      0-23)?")
2 wait_time_str = input("How many hours do you want to wait")
3
4 current_time_int = int(current_time_str)
5 wait_time_int = int(wait_time_str)
6
7 final_time_int = current_time_int + wait_time_int
8 print(final_time_int)
```

Now, thats a lot better, and in fact depending on the hours you chose, it may be exactly right. If you entered 8 for the current time and 5 for the wait time then 13 is correct. But if you entered 17 (5pm) for the hours and 9 for the wait time then the result of 26 is not correct. This illustrates an important aspect of **testing**, which is that it is important to test your code on a range of inputs. It is especially important to test your code on **boundary conditions**. In this case you would want to test your program for hours including 0, 23, and some in between. You would want to test your wait times for 0, and some really large numbers. What about negative numbers? Negative numbers don't make sense, but since we don't really have the tools to deal with telling the user when something is wrong we will not worry about that just yet.

So finally we need to account for those numbers that are bigger than 23. For this we will need one final step, using the modulo operator.

```
1 current_time_str = input("What is the current time (in hours
      0-23)?")
2 wait_time_str = input("How many hours do you want to wait")
3
4 current_time_int = int(current_time_str)
5 wait_time_int = int(wait_time_str)
6
7 final_time_int = current_time_int + wait_time_int
8
9 final_answer = final_time_int % 24
10
11 print("The time after waiting is: ", final_answer)
```

Of course even in this simple progression, there are other ways you could have gone astray. We'll look at some of those and how you track them down in the next section.

Know Your Error Messages

Many problems in your program will lead to an error message. For example as I was writing and testing this chapter of the book I wrote the following version of the example program in the previous section.

```
1 current_time_str = input("What is the current time (in hours
      0-23)?")
2 wait_time_str = input("How many hours do you want to wait")
3
4 current_time_int = int(current_time_str)
5 wait_time_int = int(wait_time_int)
6
7 final_time_int = current_time_int + wait_time_int
8 print(final_time_int)
```

Can you see what is wrong, just by looking at the code? Maybe, maybe not. Our brain tends to see what we think is there, so sometimes it is very hard to find the problem just by looking at the code. Especially when it is our own code and we are sure that we have done everything right!

Aha! Now we have an error message that might be useful. The name error tells us that `wait_time_int` is not defined. It also tells us that the error is on line 5. Thats **really** useful information. Now look at line five and you will see that `wait_time_int` is used on both the left and the right hand side of the assignment statement.

The most common errors are ParseError, TypeError, NameError, or ValueError. We will look at these errors in three stages:

- First we will define what these four error messages mean.
- Then, we will look at some examples that cause these errors to occur.
- Finally we will look at ways to help uncover the root cause of these messages.

ParseError

Parse errors happen when you make an error in the syntax of your program. Syntax errors are like making grammatical errors in writing. If you don't use periods and commas in your writing then you are making it hard for other readers to figure out what you are trying to say. Similarly Python has certain grammatical rules that must be followed or else Python can't figure out what you are trying to say.

Usually ParseErrors can be traced back to missing punctuation characters, such as parenthesis, quotation marks, or commas. Remember that in Python commas are used to separate parameters to functions. Paretheses must be balanced, or else Python thinks that you are trying to include everything that follows as a parameter to some function.

Finding Clues How can you help yourself find these problems? One trick that can be very valuable in this situation is to simply start by commenting out the line number that is flagged as having the error. If you comment out line four, the error message now changes to point to line 5. Now you ask yourself, am I really that bad that I have two lines in a row that have errors on them? Maybe, so taken to the extreme, you could comment out all of the remaining lines in the program. Now the error message changes to `TokenError: EOF in multi-line statement` This is a very technical way of saying that Python got to the end of file (EOF) while it was still looking for something. In this case a right parenthesis.

Finding Clues If you follow the same advice as for the last problem, comment out line one, you will immediately get a different error message. Here's where you need to be very careful and not panic. The error message you get now is:

```
1 NameError: name 'current_time_str' is not defined on line 4!.
```

You might be very tempted to think that this is somehow related to the earlier problem and immediately conclude that there is something wrong with the variable name `current_time_str` but if you reflect for a minute you will see that by commenting out line one you have caused a new and unrelated error. That is you have commented out the creation of the name `current_time_str`. So of course when you want to convert it to an `int` you will get the NameError. Yes, this can be confusing, but it will become much easier with experience. It's also important to keep calm, and evaluate each new clue carefully so you don't waste time chasing problems that are not really there.

Uncomment line 1 and you are back to the ParseError. Another trick is to eliminate a possible source of error. Rather than commenting out the entire line you might just try to assign `current_time_str` to a constant value. For example you might make line one look like this:

```
1  current_time_str = "10"
2  #input("What is the "current time" (in hours 0-23)?")
```

Now you have assigned `current_time_str` to the string 10, and commented out the input statement. And now the program works! So you conclude that the problem must have something to do with the input function.

TypeError

TypeErrors occur when you you try to combine two objects that are not compatible. For example you try to add together an integer and a string. Usually type

errors can be isolated to lines that are using mathematical operators, and usually the line number given by the error message is an accurate indication of the line.

Here's an example of a type error created by a Polish learner. See if you can find and fix the error.

```
1 a = input(uu'wpisz ęgodzin')
2 x = input(uu'wpisz ęliczb godzin')
3 int(x)
4 int(a)
5 h = x // 24
6 s = x % 24
7 print (h, s)
8 a = a + s
9 print ('godzina teraz %s' %a)
```

Finding Clues One thing that can help you in this situation is to print out the values and the types of the variables involved in the statement that is causing the error. You might try adding a print statement after line 4 `print(x, type(x))` You will see that at least we have confirmed that x is of type string. Now you need to start to work backward through the program. You need to ask yourself, where is x used in the program? x is used on lines 2, 3, and of course 5 and 6 (where we are getting an error). So maybe you move the print statement to be after line 2 and again after 3. Line 3 is where you expect the value of x to be changed to an integer. Could line 4 be mysteriously changine x back to a string? Not very likely. So the value and type of x is just what you would expect it to be after line 2, but not after line 3. This helps you isolate the problem to line 3. In fact if you employ one of our earlier techniques of commenting out line 3 you will see that this has no impact on the error, and is a big clue that line 3 as it is currently written is useless.

NameError

Name errors almost always mean that you have used a variable before it has a value. Often NameErrors are simply caused by typos in your code. They can be hard to spot if you don't have a good eye for catching spelling mistakes. Other times you may simply mis-remember the name of a variable or even a function you want to call. You have seen one example of a NameError at the beginning of this section. Here is another one. See if you can get this program to run successfully:

```
1 str_time = input("What time is it now?")
2 str_wait_time = input("What is the number of nours to wait?")
3 time = int(str_time)
4 wait_time = int(str_wait_time)
5 time_when_alarm_go_off = time + wait_time
6 print(time_when_alarm_go_off)
```

Finding Clues With name errors one of the best things you can do is use the editor, or browser search function. Quite often if you search for the exact word in the error message one of two things will happen:

1. The word you are searching for will appear only once in your code, its also likely that it will be on the right hand side of an assignment statment, or as a parameter to a function. That should confirm for you that you have a typo somewhere. If the name in question **is** what you thought it should be then you probably have a typo on the left hand side of an assignment statement on a line before your error message occurs. Start looking backward at your assignment statements. In some cases its really nice to leave all the highlighted strings from the search function visible as they will help you very quickly find a line where you might have expected your variable to be highlighted.
2. The second thing that may happen is that you will be looking directly at a line where you expected the search to find the string in question, but it will not be highlighted. Most often that will be the typo right there.

Here is another one for you to try:

```
1 n = input("What time is it now (in hours)?")
2 n = imt(n)
3 m = input("How many hours do you want to wait?")
4 m = int(m)
5 q = m % 12
6 print("The time is now", q)
```

And one last bit of code to fix.

```
1 present_time = input("Enter the present timein hours:")
2 set_alarm = input("Set the hours for alarm:")
3 int (present_time, set_time, alarm_time)
4 alarm_time = present_time + set_alarm
5 print(alarm_time)
```

ValueError

Value errors occur when you pass a parameter to a function and the function is expecting a certain type, but you pass it a different type. We can illustrate that with this particular program in two different ways.

```
1 current_time_str = input("Current time (in hours 0-23)?")
2 current_time_int = int(current_time_str)
3 wait_time_str = input("How many hours do you want to wait")
4 wait_time_int = int(wait_time_int)
5 final_time_int = current_time_int + wait_time_int
6 print(final_time_int)
```

Run the program but instead of typing in anything to the dialog box just click OK. You should see the following error message:

```
1   ValueError: invalid literal for int() with base 10: '' on
        line: 4
```

This error is not because you have made a mistake in your program. Although sometimes we do want to check the user input to make sure its valid, but we don't have all the tools we need for that yet. The error happens because the user did not give us something we can convert to an integer, instead we gave it an empty value. Try running the program again. Now this time enter "ten" instead of the number 10. You will get a similar error message.

ValueErrors are not always caused by user input error, but in this program that is the case. We'll look again at ValueErrors again when we get to more complicated programs. For now it is worth repeating that you need to keep track of the types of your variables, and understand what types your function is expecting. You can do this by writing comments in your code, or by naming your variables in a way that reminds you of their type.

Chapter

6 Functions

Controlling complexity is the essence of computer programming.

— *Brian W. Kernighan*

So much complexity in software comes from trying to make one thing do two things.

— *Ryan Singer*

Figure 6.1: *Dennis Ritchie. Created the C programming language and invented, together with Ken Thompson, the UNIX operating system.*

6.1 Functions

In Python, a **function** is a named sequence of statements that belong together and that accomplishes some task. Functions also accept 0, 1 or more inputs (also known as **parameters** or **arguments**) that it operates on. Usually the function produces and **returns** a value. Their primary purpose is to help us organize programs into chunks that match how we think about the solution to the problem. So basically a function contains Python code (using any of the statements and control structures we've seen already).

Calling functions

There are two aspects to functions in Python. First, a function has to be **defined**: the programmer creates a function by giving it a name, specifying its parameters and writing the code that accomplishes its task/s. Secondly, the function is used in your programs and can also be used in other functions. Using a function is a simple matter of typing its name and providing it with the parameters it requires. This is known as **calling a function** or **invoking a function**. We have already used (called or invoked) functions and here we summarize what we already know:

1. The functions that we have used already include: `print`, `input`, `type`, `random`, `log`, `sqrt`, `sin` and various other mathematical functions. All of these functions were defined by other programmers and made available for us to use. `print` , `input` and `type` were **built-in** functions while the others are found in **modules**. A module is a collection of functions made available to programmers. To use a function in a module we need to *import* the module first and we call the function by using dot notation by specifying the name of the module then the function name (The built-in functions are part of the python language so no import is necessary):

   ```
   1 import math
   2 squareRoot = math.sqrt(3.0)
   ```

2. Functions accomplish some task. The `print` function displays test on the screen, the `type` returns the type of a literal or variable, the `sqrt` function returns the square root of some value.

3. Functions are accept **parameters** (or **arguments** or **input** value/s). A powerful feature of functions is the ability to accept parameters. The parameters to the `print` function is a comma separated list of values to print, the parameter to the `input` function is a prompt string, the parameter to the `sqrt` function is the value we want the square root of. Arguments allow us to call the same function multiple times with different values. The same code is used to but we get different answers depending on the parameters we give the function. The parameters modifies the behavior of the function.

4. The syntax for calling functions is the `function_name(optional_parameters)`. If there are no parameters then the brackets are still required. When there are parameters, we say that *the parameters are passed to the function*. When the function is called, its code is executed.

5. Functions may (or may not) **return** values. For example the `sqrt` function returns the square root of its input, the `input` function returns the value typed by the user, and the `sin` function returns the sine of the input angle.

It is important to know the type of value returned so it may be used properly. The `input` function returns a string. If we know that the user inputs a number then we have to convert the returned string to a number before using it:

```
1 inputValue = input("Enter an integer")
2 myInteger = (int) inputValue
```

6. Before using functions we must know whether the function returns a string, an int, a float, a boolean or something else. The returned value may be used anywhere that type may be used. For example, the `sin` function returns a float so the `sin` function can be part of an arithmetic or boolean expression or it can be passed to another function that can accept a float. Note in particular how functions can be passed to other functions. In line 5, we use the `radians` function to first convert degrees to radians since `sin` works with radians:

```
1 b = 3.0
2 a = 5
3
4 # function is part of an expression
5 root = (-b + math.sqrt( (b * b) - 4 * a * c) ) / (2 * a)
6
7 # function is input to another function
8 print(math.sqrt(b * b))
9
10 # radians function is parameter to sin
11 sine = math.sin( math.radians(90))
```

7. Functions simplify program development. In the examples above, we do not write any code that actually calculates the sine or code to convert degrees to radians. These tasks are done by the functions. The code we wrote is easier to read and understand. Functions allow us to re-use code: we write the function once and we use it whenever we wish simply by calling it and even better, we can use code written by other programmers so we do not have to do the work that other programmers have already done.

Defining functions

We know want to write our functions that we may use and perhaps other programmers. Before writing a function we need to already know the following:

1. Task: What we want the function to accomplish.
2. Arguments: What arguments (if any) do we want input to the function.

3. Return: What value do we want the function to return, if any.

4. Processing: What processing is required in order to accomplish the task.

Let us begin with a really simple function: `cubeVolume`. We analyze it as follows:

1. Task: Calculate the volume of a cube.
2. Arguments: One could write a function to calculate the volume of a cube of a specific length such as 42. Then no arguments are required and the function calculates 42^3 and we are done. But such a function is not very useful. If we wanted to calculate the volume of another cube, we require another function. Its important to **generalize** our functions: writing functions that are useful in a wide variety of cases. We decide that the length of the side of the cube should be an argument to the function and that the function should calculate the volume of a cube using the length provided as argument. You may also be tempted to have no arguments but to have an input statement in the function that gets the length of the side from the user. You will see that this is not a very general function either: for example the programmer using such a function is not able to input values any other values to the function such as values she has calculated.
3. Return: The function should return the volume of the cube. We should also decide here whether the type of the returned value: float, int, bool etc.
4. Processing: We should have some idea about how to accomplish the task. For this problem its easy: $cubevolume = s^3$. Usually we need to design more interesting algorithms.

Now to **define** (or write) the function.
The syntax for a **function definition** is:

```
1 def name( parameters ):   # function header
2     statements            # body of the function
```

You can make up any names you want for the functions you create, except that you can't use a name that is a Python keyword, and the names must follow the rules for legal identifiers that were given previously.

The parameters specify what information, if any, you have to provide in order to use the new function. Another way to say this is that the parameters specify what the function needs to do it's work.

There can be any number of statements inside the function, but they have to be indented from the `def`. In the examples in this book, we will use the standard indentation of four spaces. Function definitions are **compound statements** that have the same pattern:

1. A header line which begins with a keyword and ends with a colon.

2. A **body** consisting of one or more Python statements, each indented the same amount – *4 spaces is the Python standard* – from the header line.

In a function definition, the keyword in the header is `def`, which is followed by the name of the function and some *parameters* enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters separated from one another by commas. In either case, the parentheses are required.

We need to say a bit more about the parameters. In the function definition, the parameter list is more specifically known as the **formal parameters**. This list of names describes those things that the function will need to receive from the user-programmer of the function. When you use a function, you provide values to the formal parameters.

The figure below shows this relationship. A function needs certain information to do its work. These values, often called **arguments** or **actual parameters**, are passed to the function by the user-programmer.

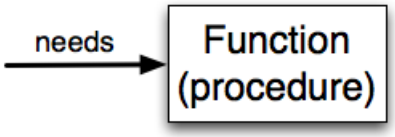


Figure 6.2: *Function as a Black Box*

This type of diagram is often called a **black-box diagram** because it only states the requirements from the perspective of the user-programmer. The user-prgrammer must know the name of the function and what arguments need to be passed. The details of how the function works are hidden inside the "black-box".

Here is our `cubeVolume` function.

```
1 # function definition
2 def CubeVolume (side):    # parameter is the length of one side
3     """return the volume of a cube with side."""
4     # function body
5     volume = side ** 3
6     return volume
7
8 # Using the function
9 print("A cube with side = 42 has volume ", cubeVolume(42))
10 aSide = input("enter a side for the cube")
11 side = float(aSide)
12 volume = cubeVolume(side)
13 print("Your cube with side ", side, " is ", volume)
```

Make sure you know where the body of the function ends — it depends on the indentation and the blank lines don't count for this purpose!

If the first thing after the function header is a string (some tools insist that it must be a triple-quoted string), it is called a **docstring** and gets special treatment in Python and in some of the programming tools. This is used to **document** the function to give user-programmers information about your function.

Another way to retrieve this information is to use the interactive interpreter, and enter the expression `<function_name>.__doc__`, which will retrieve the docstring for the function. So the string you write as documentation at the start of a function is retrievable by python tools *at runtime*. This is different from comments in your code, which are completely eliminated when the program is parsed.

By convention, Python programmers use docstrings for the key documentation of their functions.

Defining a new function does not make the function run. To do that we need a **function call**. This is also known as a **function invocation**. We've already seen how to call some built-in functions like `print`, `range` and `int`. Function calls contain the name of the function to be executed followed by a list of values, called *arguments*, which are assigned to the parameters in the function definition. You can see this after the function definition. See that you can identify **formal parameter** (side in the function definition) and the **actual parameter** (`aSide` in line 13 and `side` in line 14) being sent to the function. The **actual parameters** represent the specific data items that the function will use when it is executing. We deliberately used the same name on line 14: note that these are different variables (more about this later).

Once we've defined a function, we can call it as often as we like and its statements will be executed each time we call it.

6.2 Functions that Return Values

Most functions require arguments, values that control how the function does its job. For example, if you want to find the absolute value of a number, you have to indicate what the number is. Python has a built-in function for computing the absolute value:

```
1 print(abs(5))
2
3 print(abs(-5))
```

In this example, the arguments to the `abs` function are 5 and -5.

Some functions take more than one argument. For example the math module contains a function called `pow` which takes two arguments, the base and the exponent.

```
import math
print(math.pow(2, 3))

print(math.pow(7, 4))
```

> **NOTE**
> Of course, we have already seen that raising a base to an exponent can be done with the ** operator.

Another built-in function that takes more than one argument is `max`.

```
print(max(7, 11))
print(max(4, 1, 17, 2, 12))
print(max(3 * 11, 5 ** 3, 512 - 9, 1024 ** 0))
```

`max` can be sent any number of arguments, separated by commas, and will return the maximum value sent. The arguments can be either simple values or expressions. In the last example, 503 is returned, since it is larger than 33, 125, and 1.

Furthermore, functions like `range`, `int`, `abs` all return values that can be used to build more complex expressions.

So functions can have 0, 1 or more arguments.

Functions may also return a value or it may not. For example the `cubeVolume` function returns a value but the `print` function does not. Functions that do not return a value do not have a return value. We take a closer look at functions that return a value.

Functions that return values are sometimes called **fruitful functions**. In many other languages, a chunk that doesn't return a value is called a **procedure**, but we will stick here with the Python way of also calling it a function.

Fruitful functions still allow the user to provide information (arguments). However there is now an additional piece of data that is returned from the function.

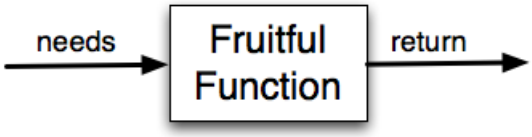


Figure 6.3: *Functions that return values*

How do we write our own fruitful function? Let's start by creating a very simple mathematical function that we will call `square`. The square function will take one number as a parameter and return the result of squaring that number. Here is the black-box diagram with the Python code following.

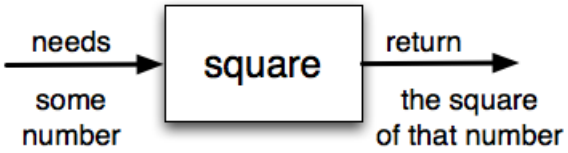


Figure 6.4: *The Square Function*

```
1 def square(x):
2     y = x * x
3     return y
4
5 toSquare = 10
6 result = square(toSquare)
7 print("The result of ", toSquare, " squared is ", result)
```

The **return** statement is followed by an expression which is evaluated. Its result is returned to the caller. Because the return statement can contain any Python expression we could have avoided creating the **temporary variable** `y` and simply used `return x*x`. Try modifying the square function above to see that this works just the same. On the other hand, using **temporary variables** like `y` in the program above makes debugging easier. These temporary variables are referred to as **local variables**.

Notice something important here. The name of the variable we pass as an argument — `toSquare` — has nothing to do with the name of the formal parameter — `x`. It is as if `x = toSquare` is executed when `square` is called. It doesn't matter what the value was named in the caller. In `square`, it's name is `x`.

The **return** statement not only causes the function to return a value, but it also returns the flow of control back to the place in the program where the function call was made.

Finally, there is one more aspect of function return values that should be noted. All Python functions return the value `None` unless there is an explicit return statement with a value other than `None`. Consider the following common mistake made by beginning Python programmers.

```
1 def square(x):
2     y = x * x
3     print(y)   # Bad! should use return instead!
4
5 toSquare = 10
6 squareResult = square(toSquare)
7 print("The result of ", toSquare, " squared is ", squareResult)
```

The problem with this function is that even though it prints the value of the square, that value will not be returned to the place where the call was done. Since

line 6 uses the return value as the right hand side of an assignment statement, the evaluation of the function will be `None`. In this case, `squareResult` will refer to that value after the assignment statement and therefore the result printed in line 7 is incorrect. Typically, functions will return values that can be printed or processed in some other way by the caller.

6.3 Variables and Parameters are Local

An assignment statement in a function creates a **local variable** for the variable on the left hand side of the assignment operator. It is called local because this variable only exists inside the function and you cannot use it outside. For example, consider again the `square` function:

```
1 def square(x):
2     y = x * x
3     return y
4
5 z = square(10)
6 print(y)  #ERROR
```

When we try to use `y` on line 6 (outside the function) Python looks for a global variable named `y` but does not find one. This results in the error: `Name Error: 'y'is not defined.`

The variable `y` only exists while the function is being executed — we call this its **lifetime**. When the execution of the function terminates (returns), the local variables are destroyed.

Formal parameters are also local and act like local variables. For example, the lifetime of `x` begins when `square` is called, and its lifetime ends when the function completes its execution.

So it is not possible for a function to set some local variable to a value, complete its execution, and then when it is called again next time, recover the local variable. Each call of the function creates new local variables, and their lifetimes expire when the function returns to the caller.

On the other hand, it is legal for a function to access a global variable. However, this is considered **bad form** by nearly all programmers and should be avoided. Look at the following, nonsensical variation of the square function.

```
1 def badsquare(x):
2     y = x ** power
3     return y
4
5 power = 2
6 result = badsquare(10)
7 print(result)
```

Although the `badsquare` function works, it is silly and poorly written. We have done it here to illustrate an important rule about how variables are looked up in Python. First, Python looks at the variables that are defined as local variables in the function. We call this the **local scope**. If the variable name is not found in the local scope, then Python looks at the global variables, or **global scope**. This is exactly the case illustrated in the code above. `power` is not found locally in `badsquare` but it does exist globally. The appropriate way to write this function would be to pass power as a parameter. For practice, you should rewrite the badsquare example to have a second parameter called power.

There is another variation on this theme of local versus global variables. Assignment statements in the local function cannot change variables defined outside the function. Consider the following example:

```
1 def powerof(x, p):
2     power = p   # Another dumb mistake
3     y = x ** power
4     return y
5
6 power = 3
7 result = powerof(10, 2)
8 print(result)
```

Now step through the code. What do you notice about the values of variable `power` in the local scope compared to the variable `power` in the global scope?

The value of `power` in the local scope was different than the global scope. That is because in this example `power` was used on the left hand side of the assignment statement `power = p`. When a variable name is used on the left hand side of an assignment statement Python creates a local variable. When a local variable has the same name as a global variable we say that the local shadows the global. A **shadow** means that the global variable cannot be accessed by Python because the local variable will be found first. This is another good reason not to use global variables. As you can see, it makes your code confusing and difficult to understand.

To cement all of these ideas even further lets look at one final example. Inside the `square` function we are going to make an assignment to the parameter `x` There's no good reason to do this other than to emphasize the fact that the parameter `x` is a local variable. If you step through the example you will see that although `x` is 0 in the local variables for `square`, the `x` in the global scope remains 2. This is confusing to many beginning programmers who think that an assignment to a formal parameter will cause a change to the value of the variable that was used as the actual parameter, especially when the two share the same name. But this example demonstrates that that is clearly not how Python operates.

```
1 def square(x):
2     y = x * x
3     x = 0       # assign a new value to the parameter x
4     return y
5
6 x = 2
7 z = square(x)
8 print(z)
```

6.4 Functions can Call Other Functions

It is important to understand that each of the functions we write can be used and called from other functions we write. This is one of the most important ways that computer scientists take a large problem and break it down into a group of smaller problems. This process of breaking a problem into smaller subproblems is called **functional decomposition**.

Here's a simple example of functional decomposition using two functions. The first function called `square` simply computes the square of a given number. The second function called `sum_of_squares` makes use of square to compute the sum of three numbers that have been squared.

```
1 def square(x):
2     y = x * x
3     return y
4
5 def sum_of_squares(x, y, z):
6     a = square(x)
7     b = square(y)
8     c = square(z)
9
10    return a + b + c
11
12 a = -5
13 b = 2
14 c = 10
15 result = sum_of_squares(a, b, c)
16 print(result)
```

Even though this is a pretty simple idea, in practice this example illustrates many very important Python concepts, including local and global variables along with parameter passing. Note that when you step through this example, codelens bolds line 1 and line 5 as the functions are defined. The body of square is not executed until it is called from the `sum_of_squares` function for the first time on line 6. Also notice that when `square` is called there are two groups of local variables, one for `square` and one for `sum_of_squares`. As you step through you

will notice that x, and y are local variables in both functions and may even have different values. This illustrates that even though they are named the same, they are in fact, very different.

Now we will look at another example that uses two functions. This example illustrates an important computer science problem solving technique called **generalization**. Consider the cubeVolume function we developed at the beginning of the chapter. One way to generalize the function is to realize that a cube is a special case of a rectangular prism:

See the function prismVolume:

```
# function definition
def prismVolume (l,b,h):    # parameters are length, breadth and
     height
    """return the volume of a rectangular prism."""

    # function body
    return l * b * h

# Using the function
print("Volume of the prism: 2 4 6 is ", prismVolume(2, 4, 6))
```

The parameter names are deliberately chosen as single letters to ensure they're not misunderstood. In real programs, once you've had more experience, we will insist on better variable names than this. The point is that the program doesn't "understand" that you're calculating the volume of prism or that the parameters represent the width and the height. Concepts like prism, width, and height are meaningful for humans. They are not concepts that the program or the computer understands.

Thinking like a computer scientist involves looking for patterns and relationships. In the code above, we've done that to some extent. We noted that a cube is just a special case of a rectangular prism. We already have a function that calculates the volume of a prism. To calculate the volume of a cube we note that $l = b = h$.

```
def cubeVolume(side):        # a new version of cubeVolume
    prismVolume(side, side, side)
```

Here is the entire example with the necessary set up code.

There are some points worth noting here:

- Functions can call other functions.
- A caller of the prismVolume function might say prismVolume(2,4,6). The parameters of this function, l, b and h are assigned the values 2, 4 and 6 respectively.
- In the body of the function, l, b and h are just like any other variable.

So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two:

1. Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command. The function (including its name) can capture your mental chunking, or *abstraction*, of the problem.
2. Creating a new function can make a program smaller by eliminating repetitive code.
3. Sometimes you can write functions that allow you to solve a specific problem using a more general solution.

6.5 Flow of Execution Summary

When you are working with functions it is really important to know the order in which statements are executed. This is called the **flow of execution** and we've already talked about it a number of times in this chapter.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order, from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called. Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is adept at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the body of the function until you reach a point where that function is called.

6.6 Using a Main Function

Using functions is a good idea. It helps us to modularize our code by breaking a program into logical parts where each part is responsible for a specific task. Here is the `prismVolume` program.

```
1 import random
2
3 def prismVolume (l,b,h):    # parameters are length, breadth and
      height
4     """return the volume of a rectangular prism."""
5
6     return l * b * h
7
8 def cubeVolume (s)
9     """return the volume of a cube."""
10
11 # Using the functions
12 side = random.randrange(1,100)
13
14 length = random.randrange(1,100)
15 breadth = random.randrange(1,50)
16 height = random.randrange(1,80)
17
18 print("Volume of the prism is ", prismVolume(length, breadth,
      height))
19 print("Volume of a cube is ", cubeVolume(side))
```

If you look closely at the structure of this program, you will notice that we first perform all of our necessary `import` statements, in this case to be able to use the `random` module. Next, we define the function `prismVolume`. At this point, we could have defined as many functions as were needed. Finally, there are statements that call the functions and do the work we wanted. These final statements perform the main processing that the program will do. Notice that much of the detail has been pushed inside the functions. However, there are still these six lines of code that are needed to get things done.

In many programming languages (e.g. Java and C++), it is not possible to simply have statements sitting alone like this at the bottom of the program. They are required to be part of a special function that is automatically invoked by the operating system when the program is executed. This special function is called **main**. Although this is not required by the Python programming language, it is actually a good idea that we can incorporate into the logical structure of our program. In other words, these five lines are logically related to one another in that they provide the main tasks that the program will perform. Since functions are designed to allow us to break up a program into logical pieces, it makes sense to call this piece `main`.

The following code shows this idea. In line 11 we have defined a new function called `main` that doesn't need any parameters. The five lines of main processing are now placed inside this function. Finally, in order to execute that main processing code, we need to invoke the `main` function (line 20). When you push run, you will see that the program works the same as it did before.

```
1 import random
2
3 def prismVolume (l,b,h):    # parameters are length, breadth and
      height
4     """return the volume of a rectangular prism."""
5
6     return l * b * h
7
8 def cubeVolume (s)
9     """return the volume of a cube."""
10
11
12 def main():                      # Define the main function
13    # Using the functions
14    side = random.randrange(1,100)
15
16    length = random.randrange(1,100)
17    breadth = random.randrange(1,50)
18    height = random.randrange(1,80)
19
20    print("Volume of the prism is ", prismVolume(length, breadth,
          height))
21    print("Volume of a cube is ", cubeVolume(side))
22
23 main()                           # Invoke the main function
```

Now our program structure is as follows. First, import any modules that will be required. Second, define any functions that will be needed. Third, define a `main` function that will get the process started. And finally, invoke the main function (which will in turn call the other functions as needed).

> **Note**
>
> In Python there is nothing special about the name `main`. We could have called this function anything we wanted. We chose `main` just to be consistent with some of the other languages.

Before the Python interpreter executes your program, it defines a few special variables. One of those variables is called `__name__` and it is automatically set to the string value `"__main__"` when the program is being executed by itself in a standalone fashion. On the other hand, if the program is being imported by

another program, then the __name__ variable is set to the name of that module. This means that we can know whether the program is being run by itself or whether it is being used by another program and based on that observation, we may or may not choose to execute some of the code that we have written.

For example, assume that we have written a collection of functions to do some simple math. We can include a main function to invoke these math functions. It is much more likely, however, that these functions will be imported by another program for some other purpose. In that case, we would not want to execute our main function.

The code below defines two simple functions and a main.

```
1 def squareit(n):
2     return n * n
3
4 def tripleit(n):
5     return n*n*n
6
7 def main():
8     anum = int(input("Please enter a number"))
9     print(squareit(anum))
10    print(tripleit(anum))
11
12 if __name__ == "__main__":
13    main()
```

Line 12 uses an if statement to ask about the value of the __name__ variable. If the value is "__main__", then the main function will be called. Otherwise, it can be assumed that the program is being imported into another program and we do not want to call main because that program will invoke the functions as needed. This ability to conditionally execute our main function can be extremely useful when we are writing code that will potentially be used by others. It allows us to include functionality that the user of the code will not need, most often as part of a testing process to be sure that the functions are working correctly.

6.7 Program Development

At this point, you should be able to look at complete functions and tell what they do. Also, if you have been doing the exercises, you have written some small functions. As you write larger functions, you might start to have more difficulty, especially with runtime and semantic errors.

To deal with increasingly complex programs, we are going to suggest a technique called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the Pythagorean theorem, the distance is:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_2)^2}$$

The first step is to consider what a `distance` function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value.

Already we can write an outline of the function that captures our thinking so far.

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated.

To test the new function, we call it with sample values.

```
def distance(x1, y1, x2, y2):
    return 0.0

print(distance(1, 2, 4, 6))
```

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be — in the last line we added.

A logical first step in the computation is to find the differences x_2- x_1 and y_2- y_1. We will store those values in temporary variables named `dx` and `dy`.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    return 0.0
```

Next we compute the sum of squares of `dx` and `dy`.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    return 0.0
```

Again, we could run the program at this stage and check the value of `dsquared` (which should be 25).

Finally, using the fractional exponent `0.5` to find the square root, we compute and return the result.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared**0.5
    return result

print(distance(1, 2, 4, 6))
```

If that works correctly, you are done. Otherwise, you might want to print the value of `result` before the return statement.

When you start out, you might add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger conceptual chunks. As you improve your programming skills you should find yourself managing bigger and bigger chunks: this is very similar to the way we learned to read letters, syllables, words, phrases, sentences, paragraphs, etc., or the way we learn to chunk music — from indvidual notes to chords, bars, phrases, and so on.

The key aspects of the process are:

1. Start with a working skeleton program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
2. Use temporary variables to hold intermediate values so that you can easily inspect and check them.
3. Once the program is working, you might want to consolidate multiple statements into compound expressions, but only do this if it does not make the program more difficult to read.

6.8 Composition

As we have already seen, you can call one function from within another. This ability to build functions by using other functions is called **composition**.

As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we've just written a function, `distance`, that does just that, so now all we have to do is use it:

```
radius = distance(xc, yc, xp, yp)
```

The second step is to find the area of a circle with that radius and return it. Again we will use one of our earlier functions:

```
result = area(radius)
return result
```

Wrapping that up in a function, we get:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared**0.5
    return result

def area(radius):
    b = 3.14159 * radius**2
    return b

def area2(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result

print(area2(0,0,1,1))
```

We called this function `area2` to distinguish it from the `area` function defined earlier. There can only be one function with a given name within a module.

Note that we could have written the composition without storing the intermediate results.

```
def area2(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.9 Boolean Functions

We have already seen that boolean values result from the evaluation of boolean expressions. Since the result of any expression evaluation can be returned by a function (using the `return` statement), functions can return boolean values. This turns out to be a very convenient way to hide the details of complicated tests. For example:

```
def isDivisible(x, y):
    if x % y == 0:
        result = True
    else:
        result = False

    return result

print(isDivisible(10, 5))
```

The name of this function is `isDivisible`. It is common to give **boolean functions** names that sound like yes/no questions. `isDivisible` returns either `True` or `False` to indicate whether the `x` is or is not divisible by `y`.

We can make the function more concise by taking advantage of the fact that the condition of the `if` statement is itself a boolean expression. We can return it directly, avoiding the `if` statement altogether:

```
def isDivisible(x, y):
    return x % y == 0
```

Boolean functions are often used in conditional statements:

```
if isDivisible(x, y):
    ... # do something ...
else:
    ... # do something else ...
```

It might be tempting to write something like `if isDivisible(x, y)== True:` but the extra comparison is not necessary. The following example shows the `isDivisible` function at work. Notice how descriptive the code is when we move the testing details into a boolean function. Try it with a few other actual parameters to see what is printed.

```
def isDivisible(x, y):
    if x % y == 0:
        result = True
    else:
        result = False

    return result

if isDivisible(10, 5):
    print("That works")
else:
    print("Those values are no good")
```

6.10 Exercises

1. Write a function `areaOfCircle(r)` which returns the area of a circle of radius r. Make sure you use the math module in your solution.

2. Write a function called `mySqrt` that will approximate the square root of a number, call it n, by using Newton's algorithm. Newton's approach is an iterative guessing algorithm where the initial guess is `n/2` and each subsequent guess is computed using the formula: `newguess = (1/2)* (oldguess + (n/oldguess))`.

3. Write a function called myPi that will return an approximation of PI (3.14159...). Use the following series known as the Madhava-Leibniz series`http://en.wikipedia.org/wiki/Madhava_of_Sangamagrama`:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + \frac{(-1)^n}{2n+1}$$

4. Write a boolean function `isPrime(x)` that returns `True` if `x` is prime, `False` otherwise. Use this function to write efficient programs to do the following:

   a) Print all primes between 2 numbers input by the user.

   b) Twin primes are consecutive odd numbers both of which are prime numbers. Write a program which inputs two positive integers A and B and outputs all the twin primes in the range A to B.

5. A **perfect cube** is an integer whose cube root is also an integer. Examples of perfect cubes are 8 (2^3), 64 (4^3) and 125 (5^3).

   A **Dudeney number** is a perfect cube such that the sum of its digits is equal to its cube root. For example, 512 is a Dudeney number since the sum of its digits ($5+1+2 = 8$) is equal to its cube root (8).

   The following are examples of Dudeney numbers:

```
1    1       = 1 x 1 x 1       and  1   = 1
2    512     = 8 x 8 x 8       and  8   = 5 + 1 + 2
3    4913    = 17 x 17 x 17    and  17  = 4 + 9 + 1 + 3
4    5832    = 18 x 18 x 18    and  18  = 5 + 8 + 3 + 2
```

   Write a program that finds all Dudeney numbers between 1 and 20000.

   Your solution must define and use a function `sumDigits(n)` that returns the sum of the digits of `n`.

6. Write a function `factorial(n)` that calculates and returns the factorial of `n`.

7. Write a function `gcd(m,n)` that calculates the Greatest Common Factor (GCD) of the two non-two non-negative integers `m` and `n`. Then write another function `lcm(m,n)` that calculates and returns the least common multiple of `m` and `n`. The `lcm` may be calculated using the fact that $lcm(m,n) = \frac{m \times n}{gcd(m,n)}$. Test both functions.

8. Write a function to find the smallest prime factor of an integer `n`. Use this function in a program that inputs an integer and prints out its prime decomposition.

Chapter

7 Turtle Graphics

If it doesn't work, it doesn't matter how fast it doesn't work.

— Mich Ravera

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

- Martin Fowler

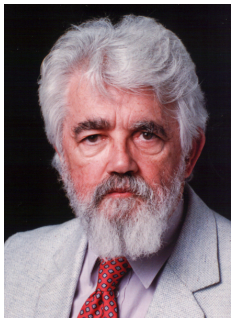


Figure 7.1: *John McCarthy. McCarthy was one of the founders of the discipline of artificial intelligence. He coined the term "artificial intelligence" (AI) and developed the Lisp programming language family.*

7.1 Hello Little Turtles!

Turtle graphics, as it is known, is based on a very simple idea. Imagine that you have a turtle that understands simple commands. You can tell your turtle to do simple commands such as go forward and turn right. As the turtle moves around, if its tail is touching the ground, it will draw a line (leave a trail behind) as it moves. If you tell your turtle to lift up its tail it can still move around but will not leave a trail. As you will see, you can make some pretty amazing drawings with this simple capability.

Note

The turtles are fun, but the real purpose of the chapter is to teach ourselves a little more Python and to develop our theme of *computational thinking*, or *thinking like a computer scientist*.

Our First Turtle Program

Let's try a couple of lines of Python code to create a new turtle and start drawing a simple figure like a rectangle. We will refer to our first turtle using the variable name `alex`, but remember that you can choose any name you wish as long as you follow the naming rules from the previous chapter.

The program as shown will only draw the first two sides of the rectangle. After line 4 you will have a straight line going from the center of the drawing canvas towards the right. After line 6, you will have a canvas with a turtle and a half drawn rectangle.

```
1 import turtle                 # allows us to use the turtles
      library
2 wn = turtle.Screen()          # creates a graphics window
3 alex = turtle.Turtle()        # create a turtle named alex
4 alex.forward(150)             # tell alex to move forward by 150
      units
5 alex.left(90)                 # turn by 90 degrees
6 alex.forward(75)              # complete the second side of a
      rectangle
```

Here are a couple of things you'll need to understand about this program.

The first line tells Python to load a **module** named `turtle`. A module is collection of python code that provides some functionality. It usually contains several functions that programmers may use in their programs. Recall that a Python function is a named sequence of instructions that takes 0, one or more inputs and accomplishes some task. Another word for input is **argument**. We say that the functions takes an argument. The turtle module contains several functions that allow us to create one or more turtles and to control their movement on the canvas. The import statement loads the turtle module and makes the functions available for us to use.

The next line creates and opens a screen (or window), which we assign to variable `wn`. Every window contains a **canvas**, which is the area inside the window on which we can draw.

In line 3 we create a turtle. The variable `alex` is made to refer to this turtle.

These first three lines set us up so that we are ready to do some drawing. It is not terribly important to understand the details of the first two lines. You need these two lines in all programs using the turtle module. You do need to learn

how to create turtles (line 3). Note that you are free to use any name for the variable (provided you follow the rules for identifiers).

In lines 4-6, we instruct `alex` to move and to turn. Notice that we use functions to this. The `forward` function takes as input an integer and when the function is executed it moves the turtle by that number of units (150 in this example). The `left` also takes an integer that represents an angle in degrees and turns the turtle by that number of degrees. So these functions are similar to the `print` and `input` functions in that they take in some input and accomplish some task.

Modify the program by adding the commands necessary to have *alex* complete the rectangle.

> **Note**
>
> The dot notation (having `alex.` in front of the function) means that we want the function to operate on that particular turtle. This allows us to create multiple turtles and give instructions to each of them:

```
1 import turtle                 # allows us to use the turtles
      library
2 wn = turtle.Screen()          # creates a graphics window
3 alex = turtle.Turtle()        # create a turtle named alex
4 alex.forward(150)             # tell alex to move forward by 150
      units
5 alex.left(90)                 # turn by 90 degrees
6 alex.forward(75)              # complete the second side of a
      rectangle
7 sally = turtle.Turtle()       # create another turtle name Sally
8 sally.right(45)                # turn sally by 45 degrees
9 sally.forward(100)            # move sally forward by 100 units
```

There are various functions available to instruct the turtles. For example, each turtle has a *color*. The function `alex.color("red")` will make alex red and the line that it draws will be red too. There is also a function to change the the width of its pen(tail). The window object also has functions for e.g. a function to change its background color.

```
1 import turtle
2
3 wn = turtle.Screen()
4 wn.bgcolor("lightgreen") # set the window background color
5
6 tess = turtle.Turtle()
7 tess.color("blue")  # make tess blue
8 tess.pensize(3)     # set the width of her pen
9
```

```
10 tess.forward(50)
11 tess.left(120)
12 tess.forward(50)
13
14 wn.exitonclick()  # wait for a user click on the canvas
```

The last line plays a very important role. The `wn` variable refers to the window shown above. When we invoke its `exitonclick` function, the program pauses execution and waits for the user to click the mouse somewhere in the window. When this click event occurs, the response is to close the turtle window and exit (stop execution of) the Python program.

Each time we run this program, a new drawing window pops up, and will remain on the screen until we click on it.

Do the following to extend the program:

1. Modify this program so that before it creates the window, it prompts the user to enter the desired background color. It should store the user's responses in a variable, and modify the color of the window according to the user's wishes. (Hint: you can find a list of permitted color names at `http://www.w3schools.com/html/html_colornames.asp`. It includes some quite unusual ones, like "PeachPuff" and "HotPink".)
2. Do similar changes to allow the user, at runtime, to set tess' color.
3. Do the same for the width of tess' pen. *Hint:* your dialog with the user will return a string, but tess' `pensize` method expects its argument to be an `int`. That means you need to convert the string to an int before you pass it to `pensize`.

A Bale of Turtles

Just like we can have many different integers in a program, we can have many turtles. Each of them is an independent object — so alex might draw with a thin black pen and be at some position, while tess might be going in her own direction with a fat pink pen. So here is what happens when alex completes a square and tess completes her triangle:

```
1 import turtle
2 wn = turtle.Screen()              # Set up the window and its
      attributes
3 wn.bgcolor("lightgreen")
4
5
6 tess = turtle.Turtle()            # create tess and set some
      attributes
7 tess.color("hotpink")
8 tess.pensize(5)
```

```
9
10 alex = turtle.Turtle()           # create alex
11
12 tess.forward(80)                 # Let tess draw an equilateral
       triangle
13 tess.left(120)
14 tess.forward(80)
15 tess.left(120)
16 tess.forward(80)
17 tess.left(120)                   # complete the triangle
18
19 tess.right(180)                  # turn tess around
20 tess.forward(80)                 # move her away from the origin
21
22 alex.forward(50)                 # make alex draw a square
23 alex.left(90)
24 alex.forward(50)
25 alex.left(90)
26 alex.forward(50)
27 alex.left(90)
28 alex.forward(50)
29 alex.left(90)
30
31 wn.exitonclick()
```

Here are some observations:

- ▷ There are 360 degrees in a full circle. If you add up all the turns that a turtle makes, *no matter what steps occurred between the turns*, you can easily figure out if they add up to some multiple of 360. This should convince you that alex is facing in exactly the same direction as he was when he was first created. (Geometry conventions have 0 degrees facing East and that is the case here too!)
- ▷ We could have left out the last turn for alex, but that would not have been as satisfying. If you're asked to draw a closed shape like a square or a rectangle, it is a good idea to complete all the turns and to leave the turtle back where it started, facing the same direction as it started in. This makes reasoning about the program and composing chunks of code into bigger programs easier for us humans!
- ▷ We did the same with tess: she drew her triangle and turned through a full 360 degress. Then we turned her around and moved her aside. Even the blank line 18 is a hint about how the programmer's *mental chunking* is working: in big terms, tess' movements were chunked as "draw the triangle" (lines 12-17) and then "move away from the origin" (lines 19 and 20).

- One of the key uses for comments is to record your mental chunking, and big ideas. They're not always explicit in the code.
- And, uh-huh, two turtles may not be enough for a herd, but you get the idea!

Iteration Simplifies our Turtle Program

To draw a square we'd like to do the same thing four times — move the turtle forward some distance and turn 90 degrees. We previously used 8 lines of Python code to have alex draw the four sides of a square. This next program does exactly the same thing but, with the help of the for statement, uses just three lines (not including the setup code). Remember that the `for` statement will repeat the forward and left four times, one time for each value in the list.

```
import turtle            # set up alex
wn = turtle.Screen()
alex = turtle.Turtle()

for i in [0, 1, 2, 3]:   # repeat four times
    alex.forward(50)
    alex.left(90)

wn.exitonclick()
```

While "saving some lines of code" might be convenient, it is not the big deal here. What is much more important is that we've found a "repeating pattern" of statements, and we reorganized our program to repeat the pattern. Finding the chunks and somehow getting our programs arranged around those chunks is a vital skill when learning *How to think like a computer scientist*.

The values [0,1,2,3] were provided to make the loop body execute 4 times. We could have used any four values. For example, consider the following program.

```
import turtle            # set up alex
wn = turtle.Screen()
alex = turtle.Turtle()

for aColor in ["yellow", "red", "purple", "blue"]:      # repeat
     four times
    alex.forward(50)
    alex.left(90)

wn.exitonclick()
```

In the previous example, there were four integers in the list. This time there are four strings. Since there are four items in the list, the iteration will still occur

four times. aColor will take on each color in the list. We can even take this one step further and use the value of aColor as part of the computation.

```
import turtle            # set up alex
wn = turtle.Screen()
alex = turtle.Turtle()

for aColor in ["yellow", "red", "purple", "blue"]:
    alex.color(aColor)
    alex.forward(50)
    alex.left(90)

wn.exitonclick()
```

In this case, the value of aColor is used to change the color attribute of alex. Each iteration causes aColor to change to the next value in the list.

```
for i in range(4):
    alex.forward(50)
    alex.left(90)
```

A Turtle Bar Chart

Recall from our discussion of modules that there were a number of things that turtles can do. Here are a couple more tricks (remember that they are all described in the module documentation).

- We can get a turtle to display text on the canvas at the turtle's current position. The method is called write. For example, alex.write("Hello") would write the string hello at the current position.
- One can fill a shape (circle, semicircle, triangle, etc.) with a fill color. It is a two-step process. First you call the method begin_fill, for example alex.begin_fill(). Then you draw the shape. Finally, you call end_fill (alex.end_fill()).
- We've previously set the color of our turtle - we can now also set it's fill color, which need not be the same as the turtle and the pen color. To do this, we use a method called fillcolor, for example, alex.fillcolor("red").

Ok, so can we get tess to draw a bar chart? Let us start with some data to be charted,

```
xs = [48, 117, 200, 240, 160, 260, 220]
```

Corresponding to each data measurement, we'll draw a simple rectangle of that height, with a fixed width. Here is a simplified version of what we would like to create.

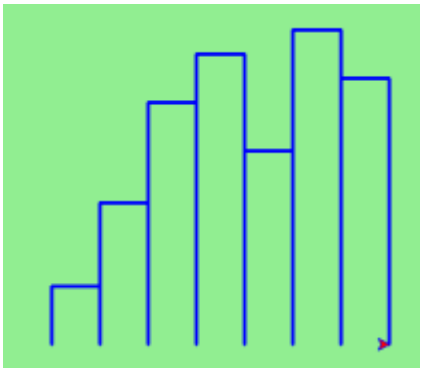


Figure 7.2: *Tess Drawing a Bar Graph*

We can quickly see that drawing a bar will be similar to drawing a rectangle or a square. Since we will need to do it a number of times, it makes sense to create a function, `drawBar`, that will need a turtle and the height of the bar. We will assume that the width of the bar will be 40 units. Once we have the function, we can use a basic for loop to process the list of data values.

```
def drawBar(t, height):
    """ Get turtle t to draw one bar, of height. """
    t.left(90)                # Point up
    t.forward(height)         # Draw up the left side
    t.right(90)
    t.forward(40)             # width of bar, along the top
    t.right(90)
    t.forward(height)         # And down again!
    t.left(90)                # put the turtle facing the way we
        found it.

...
for v in xs:                  # assume xs and tess are ready
    drawBar(tess, v)
```

It is a nice start! The important thing here was the mental chunking. To solve the problem we first broke it into smaller pieces. In particular, our chunk is to draw one bar. We then implemented that chunk with a function. Then, for the whole chart, we repeatedly called our function.

Next, at the top of each bar, we'll print the value of the data. We will do this in the body of `drawBar` by adding `t.write(str(height))` as the new fourth line of the body. Note that we had to turn the number into a string. Finally, we'll add the two methods needed to fill each bar.

The one remaining problem is related the fact that our turtle lives in a world where position (0,0) is at the center of the drawing canvas. In this problem, it would help if (0,0) were in the lower left hand corner. To solve this we can use

our `setworldcoordinates` method to rescale the window. While we are at it, we should make the window fit the data. The tallest bar will correspond to the maximum data value. The width of the window will need to be proportional to the number of bars (the number of data values) where each has a width of 40. Using this information, we can compute the coordinate system that makes sense for the data set. To make it look nice, we'll add a 10 unit border around the bars.

Here is the complete program. Try it and then change the data to see that it can adapt to the new values. Note also that we have stored the data values in a list and used a few list functions. We will have much more to say about lists in a later chapter.

```
1
2  import turtle
3
4  def drawBar(t, height):
5      """ Get turtle t to draw one bar, of height. """
6      t.begin_fill()               # start filling this shape
7      t.left(90)
8      t.forward(height)
9      t.write(str(height))
10     t.right(90)
11     t.forward(40)
12     t.right(90)
13     t.forward(height)
14     t.left(90)
15     t.end_fill()                 # stop filling this shape
16
17
18
19 xs = [48, 117, 200, 240, 160, 260, 220]  # here is the data
20 maxheight = max(xs)
21 numbars = len(xs)
22 border = 10
23
24 tess = turtle.Turtle()          # create tess and set some
       attributes
25 tess.color("blue")
26 tess.fillcolor("red")
27 tess.pensize(3)
28
29 wn = turtle.Screen()            # Set up the window and its
       attributes
30 wn.bgcolor("lightgreen")
31 wn.setworldcoordinates(0-border, 0-border, 40*numbars+border,
       maxheight+border)
32
```

```
33
34  for a in xs:
35      drawBar(tess, a)
36
37  wn.exitonclick()
```

Randomly Walking Turtles

Suppose we want to entertain ourselves by watching a turtle wander around randomly inside the screen. When we run the program we want the turtle and program to behave in the following way:

1. The turtle begins in the center of the screen.
2. Flip a coin. If its heads then turn to the left 90 degrees. If its tails then turn to the right 90 degrees.
3. Take 50 steps forward.
4. If the turtle has moved outside the screen then stop, otherwise go back to step 2 and repeat.

Notice that we cannot predict how many times the turtle will need to flip the coin before it wanders out of the screen, so we can't use a for loop in this case. In fact, although very unlikely, this program might never end, that is why we call this indefinite iteration.

So based on the problem description above, we can outline a program as follows:

```
create a window and a turtle

while the turtle is still in the window:
    generate a random number between 0 and 1
    if the number == 0 (heads):
        turn left
    else:
        turn right
    move the turtle forward 50
```

Now, probably the only thing that seems a bit confusing to you is the part about whether or not the turtle is still in the screen. But this is the nice thing about programming, we can delay the tough stuff and get *something* in our program working right away. The way we are going to do this is to delegate the work of deciding whether the turtle is still in the screen or not to a boolean function. Let's call this boolean function `isInScreen` We can write a very simple version of this boolean function by having it always return `True`, or by having it decide randomly, the point is to have it do something simple so that we can focus on

the parts we already know how to do well and get them working. Since having it always return true would not be a good idea we will write our version to decide randomly. Let's say that there is a 90% chance the turtle is still in the window and 10% that the turtle has escaped.

```
import random
import turtle


def isInScreen(w, t):
    if random.random() > 0.1:
        return True
    else:
        return False


t = turtle.Turtle()
wn = turtle.Screen()

t.shape('turtle')
while isInScreen(wn, t):
    coin = random.randrange(0, 2)
    if coin == 0:               # heads
        t.left(90)
    else:                       # tails
        t.right(90)

    t.forward(50)

wn.exitonclick()
```

Now we have a working program that draws a random walk of our turtle that has a 90% chance of staying on the screen. We are in a good position, because a large part of our program is working and we can focus on the next bit of work – deciding whether the turtle is inside the screen boundaries or not.

We can find out the width and the height of the screen using the `window_width` and `window_height` methods of the screen object. However, remember that the turtle starts at position 0,0 in the middle of the screen. So we never want the turtle to go farther right than width/2 or farther left than negative width/2. We never want the turtle to go further up than height/2 or further down than negative height/2. Once we know what the boundaries are we can use some conditionals to check the turtle position against the boundaries and return `False` if the turtle is outside or `True` if the turtle is inside.

Once we have computed our boundaries we can get the current position of the turtle and then use conditionals to decide. Here is one implementation:

```
def isInScreen(wn,t):
    leftBound = -(wn.window_width() / 2)
    rightBound = wn.window_width() / 2
    topBound = wn.window_height() / 2
    bottomBound = -(wn.window_height() / 2)

    turtleX = t.xcor()
    turtleY = t.ycor()

    stillIn = True
    if turtleX > rightBound or turtleX < leftBound:
        stillIn = False
    if turtleY > topBound or turtleY < bottomBound:
        stillIn = False

    return stillIn
```

There are lots of ways that the conditional could be written. In this case we have given `stillIn` the default value of `True` and use two `if` statements to possibly set the value to `False`. You could rewrite this to use nested conditionals or `elif` statements and set `stillIn` to `True` in an else clause.

Here is the full version of our random walk program.

```
import random
import turtle

def isInScreen(w,t):
    leftBound = - w.window_width() / 2
    rightBound = w.window_width() / 2
    topBound = w.window_height() / 2
    bottomBound = -w.window_height() / 2

    turtleX = t.xcor()
    turtleY = t.ycor()

    stillIn = True
    if turtleX > rightBound or turtleX < leftBound:
        stillIn = False
    if turtleY > topBound or turtleY < bottomBound:
        stillIn = False

    return stillIn

t = turtle.Turtle()
wn = turtle.Screen()

t.shape('turtle')
while isInScreen(wn,t):
```

```
26      coin = random.randrange(0, 2)
27      if coin == 0:
28          t.left(90)
29      else:
30          t.right(90)
31
32      t.forward(50)
33
34  wn.exitonclick()
```

We could have written this program without using a boolean function. You might want to try to rewrite it using a complex condition on the while statement. However, using a boolean function makes the program much more readable and easier to understand. It also gives us another tool to use if this was a larger program and we needed to have a check for whether the turtle was still in the screen in another part of the program. Another advantage is that if you ever need to write a similar program, you can reuse this function with confidence the next time you need it. Breaking up this program into a couple of parts is another example of functional decomposition.

A Few More turtle Functions and Observations

Here are a few more things that you might find useful as you write programs that use turtles.

- Turtle functions can use negative angles or distances. So `tess.forward(-100)` will move tess backwards, and `tess.left(-30)` turns her to the right. Additionally, because there are 360 degrees in a circle, turning 30 to the left will leave you facing in the same direction as turning 330 to the right! (The on-screen animation will differ, though — you will be able to tell if tess is turning clockwise or counter-clockwise!)

  This suggests that we don't need both a left and a right turn function — we could be minimalists, and just have one function. There is also a *backward* function. (If you are very nerdy, you might enjoy saying `alex.backward(-100)` to move alex forward!)

  Part of *thinking like a scientist* is to understand more of the structure and rich relationships in your field. So revising a few basic facts about geometry and number lines, like we've done here is a good start if we're going to play with turtles.

- A turtle's pen can be picked up or put down. This allows us to move a turtle to a different place without drawing a line. The functions are `up` and `down`. Note that the functions `penup` and `pendown` do the same thing.

```
alex.up()
alex.forward(100)     # this moves alex, but no line is drawn
alex.down()
```

- Every turtle can have its own shape. The ones available "out of the box" are `arrow`, `blank`, `circle`, `classic`, `square`, `triangle`, `turtle`.

```
...
alex.shape("turtle")
...
```

- You can speed up or slow down the turtle's animation speed. (Animation controls how quickly the turtle turns and moves forward). Speed settings can be set between 1 (slowest) to 10 (fastest). But if you set the speed to 0, it has a special meaning — turn off animation and go as fast as possible.

```
alex.speed(10)
```

- A turtle can "stamp" its footprint onto the canvas, and this will remain after the turtle has moved somewhere else. Stamping works even when the pen is up.

Let's do an example that shows off some of these new features.

```
import turtle
wn = turtle.Screen()
wn.bgcolor("lightgreen")
tess = turtle.Turtle()
tess.color("blue")
tess.shape("turtle")

print(range(5, 60, 2))
tess.up()                       # this is new
for size in range(5, 60, 2):    # start with size = 5 and grow by 2
    tess.stamp()                # leave an impression on the canvas
    tess.forward(size)          # move tess along
    tess.right(24)              # and turn her

wn.exitonclick()
```

The list of integers shown above is created by printing the `range(5,60,2)` result. It is only done to show you the distances being used to move the turtle forward. The actual use appears as part of the `for` loop.

One more thing to be careful about. All except one of the shapes you see on the screen here are footprints created by `stamp`. But the program still only has *one* turtle instance — can you figure out which one is the real tess? (Hint: if you're not sure, write a new line of code after the `for` loop to change tess' color, or to put her pen down and draw a line, or to change her shape, etc.)

Summary of Turtle Methods

Method	Parameters	Description
Turtle	None	Creates and returns a new turtle object
forward	distance	Moves the turtle forward
backward	distance	Moves the turle backward
right	angle	Turns the turtle clockwise
left	angle	Turns the turtle counter clockwise
up	None	Picks up the turtles tail
down	None	Puts down the turtles tail
color	color name	Changes the color of the turtle's tail
fillcolor	color name	Changes the color of the turtle will use to fill a polygon
heading	None	Returns the current heading
position	None	Returns the current position
goto	x,y	Move the turtle to position x,y
begin_fill	None	Remember the starting point for a filled polygon
end_fill	None	Close the polygon and fill with the current fill color
dot	None	Leave a dot at the current position
stamp	None	Leaves an impression of a turtle shape at the current location
shape	shapename	Should be 'arrow', 'classic', 'turtle', or 'circle'

Once you are comfortable with the basics of turtle graphics you can read about even more options on the . Note that we will describe Python Docs in more detail in the next chapter.

7.2 Exercises

1. Write a program that asks the user for the number of sides, the length of the side, the color, and the fill color of a regular polygon. The program should draw the polygon and then fill it in.

2. On a piece of scratch paper, trace the following program and show the drawing. When you are done, press `run` and check your answer.

3. Write a program to draw a face of a clock that looks something like this:

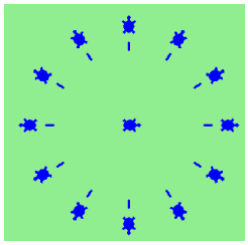


Figure 7.3: *Tess draws a clock*

4. Draw five stars, but between each, pick up the pen, move forward by 350 units, turn right by 144, put the pen down, and draw the next star. You'll get something like this (note that you will need to move to the left before drawing your first star in order to fit everything in the window):

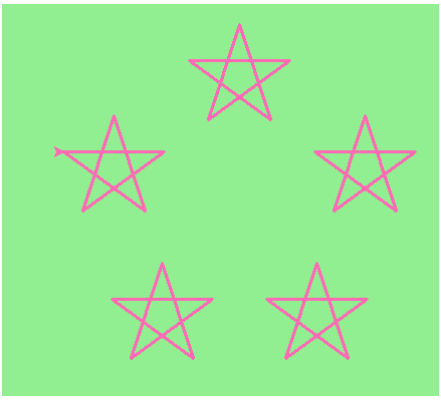


Figure 7.4: *Five Stars*

What would it look like if you didn't pick up the pen?

5. Write a program to draw this. Assume the innermost square is 20 units per side, and each successive square is 20 units bigger, per side, than the one inside it.

Figure 7.5: *Nested Squares*

6. Draw this pretty pattern.

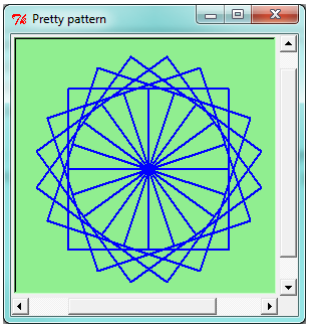


FIGURE 7.6: *Pretty Pattern*

7. Write a non-fruitful function `drawEquitriangle(someturtle, somesize)` which calls `drawPoly` from the previous question to have its turtle draw a equilateral triangle.

Chapter

8 Strings

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the universe trying to build bigger and better idiots. So far, the universe is winning.

- Rick Cook

Simplicity is prerequisite for reliability.

— Edsger W. Dijkstra

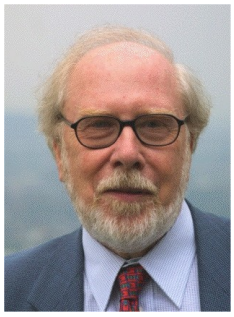


Figure 8.1: *Niklaus Emil Wirth (1934–) is a Swiss computer scientist, best known for designing several programming languages, including Pascal, and for pioneering several classic topics in software engineering.*

8.1 Strings Revisited

Throughout the first chapters of this book we have used strings to represent words or phrases that we wanted to print out. Our definition was simple: a string is simply some characters inside quotes. In this chapter we explore strings in much more detail.

So far we have seen built-in types like: `int`, `float`, `bool`, `str` and we've seen lists. `int`, `float`, and `bool` are considered to be simple or primitive data types because their values are not composed of any smaller parts. They cannot be broken down. On the other hand, strings and lists are different from the others

because they are made up of smaller pieces. In the case of strings, they are made up of smaller strings each containing one **character**.

Types that are comprised of smaller pieces are called **collection data types**. Depending on what we are doing, we may want to treat a collection data type as a single entity (the whole), or we may want to access its parts. This ambiguity is useful.

Strings can be defined as sequential collections of characters. This means that the individual characters that make up the string are assumed to be in a particular order from left to right. A character is any letter, number, space, punctuation mark, or symbol that can be typed on a computer.

A string that contains no characters, often referred to as the **empty string**, is still considered to be a string. It is simply a sequence of zero characters and is represented by ” or ”” (two single or two double quotes with nothing in between).

8.2 Operations on Strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that `message` has type string):

```
1 message - 1
2 ”Hello” / 123
3 message * ”Hello”
4 ”15” + 2
```

Interestingly, the `+` operator does work with strings, but for strings, the `+` operator represents **concatenation**, not addition. Concatenation means joining the two operands by linking them end-to-end. For example:

```
1 fruit = ”banana”
2 bakedGood = ” nut bread”
3 print(fruit + bakedGood)
```

The output of this program is `banana nut bread`. The space before the word `nut` is part of the string and is necessary to produce the space between the concatenated strings. Take out the space and run it again.

The `*` operator also works on strings. It performs repetition. For example, `'Fun'*3` is `'FunFunFun'`. One of the operands has to be a string and the other has to be an integer.

```
1 print(”Go” * 6)
2
3 name = ”Packers”
4 print(name * 3)
5
6 print(name + ”Go” * 3)
```

```
7
8 print((name + "Go") * 3)
```

This interpretation of `+` and `*` makes sense by analogy with addition and multiplication. Just as `4*3` is equivalent to `4+4+4`, we expect `"Go"*3` to be the same as `"Go"+"Go"+"Go"`, and it is. Note also in the last example that the order of operations for `*` and `+` is the same as it was for arithmetic. The repetition is done before the concatenation. If you want to cause the concatenation to be done first, you will need to use parenthesis.

Index Operator: Working with the Characters of a String

The **indexing operator** (Python uses square brackets to enclose the index) selects a single character from a string. The characters are accessed by their position or index value. For example, in the string shown below, the 14 characters are indexed left to right from postion 0 to position 13.

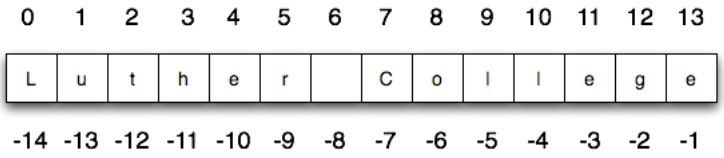


Figure 8.2: *Index Values of a String*

It is also the case that the positions are named from right to left using negative numbers where -1 is the rightmost index and so on. Note that the character at index 6 (or -8) is the blank character.

```
1 school = "University of KwaZulu-Natal"
2 m = school[2]
3 print(m)
4
5 lastchar = school[-1]
6 print(lastchar)
```

The expression `school[2]` selects the character at index 2 from `school`, and creates a new string containing just this one character. The variable `m` refers to the result.

Remember that computer scientists often start counting from zero. The letter at index zero is `U`. So at position `[2]` we have the letter `i`.

If you want the zero-eth letter of a string, you just put 0, or any expression with the value 0, in the brackets. Give it a try.

The expression in brackets is called an **index**. An index specifies a member of an ordered collection. In this case the collection of characters in the string. The index *indicates* which character you want. It can be any integer expression so long as it evaluates to a valid index value.

Note that indexing returns a *string* — Python has no special type for a single character. It is just a string of length 1.

8.3 Operations on Strings

Length

The `len` function, when applied to a string, returns the number of characters in a string.

```
fruit = "Banana"
print(len(fruit))
```

To get the last letter of a string, you might be tempted to try something like this:

```
fruit = "Banana"
sz = len(fruit)
last = fruit[sz]       # ERROR!
print(last)
```

That won't work. It causes the runtime error `IndexError: string index out of range`. The reason is that there is no letter at index position 6 in `"Banana"`. Since we started counting at zero, the six indexes are numbered 0 to 5. To get the last character, we have to subtract 1 from `length`. Give it a try in the example above.

```
fruit = "Banana"
sz = len(fruit)
lastch = fruit[sz-1]
print(lastch)
```

Typically, a Python programmer will access the last character by combining the two lines of code from above.

```
lastch = fruit[len(fruit)-1]
```

The Slice Operator

A substring of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
singers = "Peter, Paul, and Mary"
print(singers[0:5])
print(singers[7:11])
print(singers[17:21])
```

The slice operator `[n:m]` returns the part of the string from the n'th character to the m'th character, including the first but excluding the last. In other words,

start with the character at index n and go up to but do not include the character at index m. This behavior may seem counter-intuitive but if you recall the `range` function, it did not include its end point either.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string.

```
fruit = "banana"
print(fruit[:3])
print(fruit[3:])
```

What do you think `fruit[:]` means?

String Comparison

The comparison operators also work on strings. To see if two strings are equal you simply write a boolean expression using the equality operator.

```
word = "banana"
if word == "banana":
    print("Yes, we have bananas!")
else:
    print("Yes, we have NO bananas!")
```

Other comparison operations are useful for putting words in . This is similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters.

```
word = "zebra"

if word < "banana":
    print("Your word, " + word + ", comes before banana.")
elif word > "banana":
    print("Your word, " + word + ", comes after banana.")
else:
    print("Yes, we have no bananas!")
```

It is probably clear to you that the word apple would be less than (come before) the word `banana`. After all, a is before b in the alphabet. But what if we consider the words `apple` and `Apple`? Are they the same?

```
print("apple" < "banana")

print("apple" == "Apple")
print("apple" < "Apple")
```

It turns out, as you recall from our discussion of variable names, that uppercase and lowercase letters are considered to be different from one another. The way the computer knows they are different is that each character is assigned a unique integer value. "A" is 65, "B" is 66, and "5" is 53. The way you can find

out the so-called **ordinal value** for a given character is to use a character function called `ord`.

```
print(ord("A"))
print(ord("B"))
print(ord("5"))

print(ord("a"))
print("apple" > "Apple")
```

When you compare characters or strings to one another, Python converts the characters into their equivalent ordinal values and compares the integers from left to right. As you can see from the example above, "a" is greater than "A" so "apple" is greater than "Apple".

Humans commonly ignore capitalization when comparing two words. However, computers do not. A common way to address this issue is to convert strings to a standard format, such as all lowercase, before performing the comparison.

There is also a similar function called `chr` that converts integers into their character equivalent.

```
print(chr(65))
print(chr(66))

print(chr(49))
print(chr(53))

print("The character for 32 is", chr(32), "!!!")
print(ord(" "))
```

One thing to note in the last two examples is the fact that the space character has an ordinal value (32). Even though you don't see it, it is an actual character. We sometimes call it a *nonprinting* character.

The `in` and `not in` operators

The `in` operator tests if one string is a substring of another:

```
print('p' in 'apple')
print('i' in 'apple')
print('ap' in 'apple')
print('pa' in 'apple')
```

Note that a string is a substring of itself, and the empty string is a substring of any other string. (Also note that computer scientists like to think about these edge cases quite carefully!)

```
print('a' in 'a')
print('apple' in 'apple')
```

```
print('' in 'a')
print('' in 'apple')
```

The `not in` operator returns the logical opposite result of `in`.

```
print('x' not in 'apple')
```

Strings are Immutable

One final thing that makes strings different from some other Python collection types is that you are not allowed to modify the individual characters in the collection. It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example, in the following code, we would like to change the first letter of `greeting`.

```
greeting = "Hello, world!"
greeting[0] = 'J'            # ERROR!
print(greeting)
```

Instead of producing the output `Jello, world!`, this code produces the runtime error `TypeError: 'str'object does not support item assignment`.

Strings are **immutable**, which means you cannot change an existing string. The best you can do is create a new string that is a variation on the original.

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print(newGreeting)
print(greeting)             # same as it was
```

The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

8.4 Traversal

Traversal and the `for` Loop: By Item

A lot of computations involve processing a collection one item at a time. For strings this means that we would like to process one character at a time. Often we start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**.

We have previously seen that the `for` statement can iterate over the items of a sequence (a list of names in the case below).

```
for aname in ["Joe", "Amy", "Brad", "Angelina", "Zuki",
      "Thandi", "Paris"]:
    invitation = "Hi " + aname + ".  Please come to my party on
         Saturday!"
    print(invitation)
```

Recall that the loop variable takes on each value in the sequence of names. The body is performed once for each name. The same was true for the sequence of integers created by the `range` function.

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

Since a string is simply a sequence of characters, the `for` loop iterates over each character automatically.

```
for achar in "Go Spot Go":
    print(achar)
```

The loop variable `achar` is automatically reassigned each character in the string "Go Spot Go". We will refer to this type of sequence iteration as **iteration by item**. Note that it is only possible to process the characters one at a time from left to right.

Traversal and the `for` Loop: By Index

It is also possible to use the `range` function to systematically generate the indices of the characters. The `for` loop can then be used to iterate over these positions. These positions can be used together with the indexing operator to access the individual characters in the string.

Consider the following codelens example.

```
fruit = "apple"
for idx in range(5):
    currentChar = fruit[idx]
    print(currentChar)
```

The index positions in "apple" are 0,1,2,3 and 4. This is exactly the same sequence of integers returned by `range(5)`. The first time through the for loop, `idx` will be 0 and the "a" will be printed. Then, `idx` will be reassigned to 1 and "p" will be displayed. This will repeat for all the range values up to but not including 5. Since "e" has index 4, this will be exactly right to show all of the characters.

In order to make the iteration more general, we can use the `len` function to provide the bound for `range`. This is a very common pattern for traversing any sequence by position. Make sure you understand why the range function behaves correctly when using `len` of the string as its parameter value.

```
fruit = "apple"
for idx in range(len(fruit)):
    print(fruit[idx])
```

You may also note that iteration by position allows the programmer to control the direction of the traversal by changing the sequence of index values. Recall that we can create ranges that count down as well as up so the following code will print the characters from right to left.

```
fruit = "apple"
for idx in range(len(fruit)-1, -1, -1):
    print(fruit[idx])
```

Trace the values of `idx` and satisfy yourself that they are correct. In particular, note the start and end of the range.

Traversal and the `while` Loop

The `while` loop can also control the generation of the index values. Remember that the programmer is responsible for setting up the initial condition, making sure that the condition is correct, and making sure that something changes inside the body to guarantee that the condition will eventually fail.

```
fruit = "apple"

position = 0
while position < len(fruit):
    print(fruit[position])
    position = position + 1
```

The loop condition is `position < len(fruit)`, so when `position` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

Here is the same example in codelens so that you can trace the values of the variables.

```
fruit = "apple"

position = 0
while position < len(fruit):
    print(fruit[position])
    position = position + 1
```

8.5 The Accumulator Pattern with Strings

Combining the `in` operator with string concatenation using `+` and the accumulator pattern, we can write a function that removes all the vowels from a string. The idea is to start with a string and iterate over each character, checking to see if the character is a vowel. As we process the characters, we will build up a new string consisting of only the nonvowel characters. To do this, we use the accumulator pattern.

Remember that the accumulator pattern allows us to keep a "running total". With strings, we are not accumulating a numeric total. Instead we are accumulating characters onto a string.

```
1 def removeVowels(s):
2     vowels = "aeiouAEIOU"
3     sWithoutVowels = ""
4     for eachChar in s:
5         if eachChar not in vowels:
6             sWithoutVowels = sWithoutVowels + eachChar
7     return sWithoutVowels
8
9 print(removeVowels("compsci"))
10 print(removeVowels("aAbEefIijOopUus"))
```

Line 5 uses the `not in` operator to check whether the current character is not in the string `vowels`. The alternative to using this operator would be to write a very large `if` statement that checks each of the individual vowel characters. Note we would need to use logical `and` to be sure that the character is not any of the vowels.

```
1 if eachChar != 'a'  and eachChar != 'e'  and eachChar != 'i'  and
2    eachChar != 'o'  and eachChar != 'u'  and eachChar != 'A'  and
3    eachChar != 'E'  and eachChar != 'I'  and eachChar != 'O'  and
4    eachChar != 'U':
5
6      sWithoutVowels = sWithoutVowels + eachChar
```

Look carefully at line 6 in the above program (`sWithoutVowels = sWithoutVowels + eachChar`). We will do this for every character that is not a vowel. This should look very familiar. As we were describing earlier, it is an example of the accumulator pattern, this time using a string to "accumulate" the final result. In words it says that the new value of `sWithoutVowels` will be the old value of `sWithoutVowels` concatenated with the value of `eachChar`. We are building the result string character by character.

Take a close look also at the initialization of `sWithoutVowels`. We start with an empty string and then begin adding new characters to the end.

Step through the function using codelens to see the accumulator variable grow.

```
1 def removeVowels(s):
2     vowels = "aeiouAEIOU"
3     sWithoutVowels = ""
4     for eachChar in s:
5         if eachChar not in vowels:
6             sWithoutVowels = sWithoutVowels + eachChar
7     return sWithoutVowels
8
9 print(removeVowels("compsci"))
```

8.6 Looping and Counting

We will finish this chapter with a few more examples that show variations on the theme of iteration through the characters of a string. We will implement a few of the methods that we described earlier to show how they can be done.

The following program counts the number of times a particular letter, `aChar`, appears in a string. It is another example of the accumulator pattern that we have seen in previous chapters.

```
def count(text, aChar):
    lettercount = 0
    for c in text:
        if c == aChar:
            lettercount = lettercount + 1
    return lettercount

print(count("banana","a"))
```

The function `count` takes a string as its parameter. The `for` statement iterates through each character in the string and checks to see if the character is equal to the value of `aChar`. If so, the counting variable, `lettercount`, is incremented by one. When all characters have been processed, the `lettercount` is returned.

A `find` function

Here is function that returns the index of a given character in a string.

```
def find(astring, achar):
    """
    Find and return the index of achar in astring.
    Return -1 if achar does not occur in astring.
    """
    ix = 0
    found = False
    while ix < len(astring) and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1
    if found:
        return ix
    else:
        return -1

print(find("Compsci", "p"))
print(find("Compsci", "C"))
print(find("Compsci", "i"))
```

```
print(find("Compsci", "x"))
```

In a sense, `find` is the opposite of the indexing operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears for the first time. If the character is not found, the function returns `-1`.

The `while` loop in this example uses a slightly more complex condition than we have seen in previous programs. Here there are two parts to the condition. We want to keep going if there are more characters to look through and we want to keep going if we have not found what we are looking for. The variable `found` is a boolean variable that keeps track of whether we have found the character we are searching for. It is initialized to *False*. If we find the character, we reassign `found` to *True*.

The other part of the condition is the same as we used previously to traverse the characters of the string. Since we have now combined these two parts with a logical `and`, it is necessary for them both to be *True* to continue iterating. If one part fails, the condition fails and the iteration stops.

When the iteration stops, we simply ask a question to find out why and then return the proper value.

> **Note**
>
> This pattern of computation is sometimes called a eureka traversal because as soon as we find what we are looking for, we can cry Eureka! and stop looking. The way we stop looking is by setting `found` to True which causes the condition to fail.

Optional parameters

To find the locations of the second or third occurrence of a character in a string, we can modify the `find` function, adding a third parameter for the starting position in the search string:

```
def find2(astring, achar, start):
    """
    Find and return the index of achar in astring.
    Return -1 if achar does not occur in astring.
    """
    ix = start
    found = False
    while ix < len(astring) and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1
```

```
13      if found:
14          return ix
15      else:
16          return -1
17
18  print(find2('banana', 'a', 2))
```

The call `find2('banana', 'a', 2)` now returns `3`, the index of the first occurrence of 'a' in 'banana' after index 2. What does `find2('banana', 'n', 3)` return? If you said, 4, there is a good chance you understand how `find2` works. Try it.

Better still, we can combine `find` and `find2` using an **optional parameter**.

```
1  def find3(astring, achar, start=0):
2      """
3      Find and return the index of achar in astring.
4      Return -1 if achar does not occur in astring.
5      """
6      ix = start
7      found = False
8      while ix < len(astring) and not found:
9          if astring[ix] == achar:
10             found = True
11         else:
12             ix = ix + 1
13     if found:
14         return ix
15     else:
16         return -1
17
18 print(find3('banana', 'a', 2))
```

The call `find3('banana', 'a', 2)` to this version of `find` behaves just like `find2`, while in the call `find3('banana', 'a')`, `start` will be set to the **default value** of `0`.

Adding another optional parameter to `find` makes it search from a starting position, up to but not including the end position.

```
1  def find4(astring, achar, start=0, end=None):
2      """
3      Find and return the index of achar in astring.
4      Return -1 if achar does not occur in astring.
5      """
6      ix = start
7      if end == None:
8          end = len(astring)
9
10     found = False
```

```
    while ix < end and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1
    if found:
        return ix
    else:
        return -1

ss = "Python strings have some interesting methods."

print(find4(ss, 's'))
print(find4(ss, 's', 7))
print(find4(ss, 's', 8))
print(find4(ss, 's', 8, 13))
print(find4(ss, '.'))
```

The optional value for `end` is interesting. We give it a default value `None` if the caller does not supply any argument. In the body of the function we test what `end` is and if the caller did not supply any argument, we reassign `end` to be the length of the string. If the caller has supplied an argument for `end`, however, the caller's value will be used in the loop.

The semantics of `start` and `end` in this function are precisely the same as they are in the `range` function.

8.7 Exercises

1. In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop tries to output these names in order.

   ```
   prefixes = "JKLMNOPQ"
   suffix = "ack"

   for p in prefixes:
     print(p + suffix)
   ```

   Of course, that's not quite right because Ouack and Quack are misspelled. Can you fix it?

2. Print out a neatly formatted multiplication table, up to 12×12.

3. Write a function that reverses its string argument.

4. Write a function that removes all occurrences of a given letter from a string.

5. Write a function that counts how many times a substring occurs in a string.

6. Write a function `count(str,char)` that returns the number of occurrences of the character `char` in the string `str`.

7. Write a function that removes all occurrences of a string from another string.

8. Write a function that implements a substitution cipher. In a substitution cipher one letter is substituted for another to garble the message. For example A → Q, B → T, C → G etc. Your function should take two parameters, the message you want to encrypt, and a string that represents the mapping of the 26 letters in the alphabet. Your function should return a string that is the encrypted version of the message.

9. Write a function called `removeDups` that takes a string and creates a new string by only adding those characters that are not already present. In other words, there will never be a duplicate letter added to the new string.

10. Write a function to determine the most frequently occurring letter in a string.

11. Write a function for inserting one string at a particular position within another string.

12. Write a function to determine if all the characters in a string are in alphabetical order or not.