Chapter 0

The Study of Computer Science

0.1 Why Computer Science?

It is a fair question to ask. Why should anyone bother to study computer science? Furthermore, what is "computer science"? Isn't this all just about programming? All good questions. We think it is worth discussing them before you forge ahead with the rest of the book.

0.1.1 Importance of Computer Science

Let's be honest. We wouldn't be writing the book and asking you to spend your valuable time if we didn't think studying computer science is important. There are a couple of ways to look at why this is true.

First, we all know that computers are everywhere, millions upon millions of them. What were once rare, expensive items are as common place as, well any commodity you can imagine (We were going to say the proverbial toaster, but there are many times more computers than toasters. In fact, there is likely a small computer *in* your toaster!). However, that isn't enough of a reason. There are millions and millions of cars and universities don't require auto mechanics as an area of study.

A second aspect is that computers are not only common, but they are also more universally applicable than any other commodity in history. A car is good for transportation, but a computer can be used in so many situations. In fact there is almost no area one can imagine where a computer *would not* be useful. That is a key element. No matter what your area of interest, a computer could be useful there as a *tool*. The computer's universal utility is unique, and learning how to use such a tool is important.

0.1.2 Computer "Science"

> *Any field that has the word science in its name is guaranteed thereby not to be a science.*
>
> Frank Harary

A popular view of the term "computer science" is that it is a glorified way to say "computer programming." It is true that computer programming is often the way that people are introduced to computing in general, and that computer programming is the primary reason many take computing courses. However, there is indeed more to computing than programming, hence the term computer science. Here are a few examples.

Theory of Computation

Before there were the vast numbers of computers that are available today, scientists were thinking about what it means to do computing and what the limits might be. They would ask questions, such as do there exist problems that we can conceive of but cannot compute? It turns out there are. One of these problems, called the "Halting Problem"[1], cannot be solved by a program running on any computer. Knowing what you can and cannot solve on a computer is an important issue and a subject of study among computer scientists that focus on the theory of computation.

Computational Efficiency

The fact that a problem is computable does not mean it is easily computed. Knowing roughly how difficult a problem is to solve is also very important. Determining a meaningful measure of difficulty is, in itself, an interesting issue, but imagine we are concerned only with time. Consider designing a solution to a problem that, as part of the solution, required you to sort 100,000 items (say cancer patient records, or asteroid names, or movie episodes, etc). A slow algorithm, such as the sorting algorithm called the Bubble Sort might take approximately 800 seconds (about 13 minutes) while another sorting algorithm called Quick Sort might take approximately 0.3 seconds. That is a difference of around 2400 times! That large a difference might determine whether it is worth doing or not. If you are creating a solution, it would be good to know what makes your solution slow or what makes it fast.

Algorithms and Data Structures

Algorithms and Data Structures are the currency of the computer scientist. Discussed more in Chapter 3, algorithms are the methods used to solve problems, and data structures are the organizations of data that the algorithms use. These two concepts are distinct: a general approach to solving a problem (such as searching for a particular value, sorting a list of objects, encrypting a message, etc.) differs from the organization of the data that is being processed (as a list of objects, as a dictionary of key-value pairs, as a "tree" of records). However, they are also tightly coupled. Furthermore, both algorithms and data structures can be examined independently of how they might be programmed. That is, one designs algorithms and data structures and then actually implements them in a particular computer program. Understanding abstractly how to design both algorithms and data structures independent of the programming language is critical for writing correct and efficient code.

Parallel Processing

It may seem odd to include what many consider an advanced topic, but parallel processing, using multiple computers to solve a problem, is an issue for everyone these days. Why? As it turns out, most computers come with at least two processors or CPUs (see Section 0.6), and many with four or more. The present day PlayStation3™ uses a special IBM chip that has eight processors, and Intel has recently announced prototypes of chips that have eighty processors! What does this mean to us, both as consumers and new computer scientists?

The answer is that new algorithms, data structures and programming paradigms will be needed to take advantage of this new processing environment. Orchestrating many processors to solve a problem is an exciting and challenging task.

[1] http://en.wikipedia.org/wiki/Halting_problem

Software Engineering

Even the process of writing programs itself has developed its own sub discipline within computer science. Dubbed "Software Engineering", it concerns the process of creating programs: from designing the algorithms they use, to supporting testing and maintenance of the program once created. There is even a discipline interested in representing a developed program as a mathematical entity so that one can **prove** what a program will do once written.

Many Others

We have provided but a taste of the many fields that make computer science such a wonderfully rich area to explore. Every area that uses computation brings its own problems to be explored.

0.1.3 Computer Science Through Computer Programming

We have tried to make the point that computer science is not just programming. However, it is also true that for much of the book we will focus on just that aspect of computer science: programming. Beginning with "problem solving through programming" allows one to explore pieces of the computer science landscape as they naturally arise.

0.2 The Difficulty and Promise of Programming

If computer science, particularly computer programming, is so interesting, why doesn't everybody do it? The truth is, it can be hard. We are often asked by beginning students, "Why is programming so hard"? Even grizzled programming veterans, when honestly looking back at their first experience, remember how difficult that first programming course was. Why? Understanding why it might be hard gives you an edge on what you can do to control the difficulty.

0.2.1 Difficulty 1: two things at once

Let's consider an example. Let us say that, when you walk into that first day of programming 101, you discover the course is not about programming but French poetry. French poetry? Yes, French poetry. Imagine that you come in and the professor posts the following excerpt from a poem on the board.

A une Damoyselle malade

Ma mignonne,
Je vous donne
Le bon jour;
Le séjour
C'est prison.

Clément Marot

Your assigned task is to translate this poetry into English (or German, or Russian, whatever language is your native tongue). Let us also assume, for the moment, that:

a) You do not know French.

b) You have never studied poetry.

You have two problems on your hands. First, you have to gain a better understanding of the syntax and semantics (the form and substance) of the French language. Second, you need to learn more about the "rules" of poetry and what constitutes a good poem.

Lest you think that this is a trivial matter, an entire book has been written by Douglas Hofstadter on the very subject of the difficulty of translating this one poem ("Le Ton beau de Marot").

So what's your first move? Most people would break out a dictionary and, line-by-line, try to translate the poem. Hofstadter, in his book, does exactly that, producing the crude translation in Figure 1.

My Sweet/Cute [One]
(Feminine)

My sweet/cute [one]
(feminine)
I [to] you (respectful)
give/bid/convey
The good day (i.e., a
hello, i.e., greetings).
The stay/sojourn/visit
(i.e., quarantine)
{It} is prison.

A une Damoyselle malade

Ma mignonne,
Je vous donne
Le bon jour;
Le séjour
C'est prison.

Figure 1: Crude Translation of excerpt.

The result is hardly a testament to beautiful poetry. This translation does capture the syntax and semantics, but not the poetry, of the original. If we take a closer look at the poem, we can discern some features that a good translation should incorporate. For example:

- Each line consists of three syllables.
- Each line's main stress falls on its final syllable.
- The poem is a string of rhyming couplets: **AA, BB, CC,** ...
- The semantic couplets are out of phase with the rhyming couplets: **A, AB, BC,** ...

Taking some of these ideas (and many more besides) into account, Hofstadter comes up with the translation in Figure 2.

Not only does this version sound far more like poetry, but it also matches the original poem, following the rules and conveying the intent. It is a pretty good translation!

Poetry to Programming?

How does this poetry example help? Actually, the analogy is pretty strong. In coming to programming for the first time, you face exactly the same issues:

- You are not yet familiar with the syntax and semantics of the language you are working with—in this case, the programming language Python and perhaps not with **any** programming language.

My Sweet Dear	A une Damoyselle malade
My sweet dear, I send cheer – All the best! Your forced rest Is like jail.	Ma mignonne, Je vous donne Le bon jour; Le séjour C'est prison.

Figure 2: Improved Translation of excerpt.

- You do not know how to solve problems using a computer—similar to not knowing how to write poetry.

Just like the French poetry neophyte, you are trying to solve two problems simultaneously. On one level, you are just trying to get familiar with the syntax and semantics of the language. At the same time you are tackling a second, very difficult task: creating poetry in the example above; solving problems using a computer in this course.

Working at two levels, the meaning of the programming words and then the intent of the program (what the program is trying to solve) are the two problems the beginning programmer has to face. Just like the French poetry neophyte, our first programs will be a bit clumsy as we learn both the programming language and how to use that language to solve problems. For example, to a practiced eye, many first programs look similar in nature to the literal translation of Hofstadter's in Figure 1. Trying to do two things simultaneously is difficult for anyone, so be gentle on yourself as you go forward with the process.

You might ask, isn't there a better way? Perhaps, but we have not found it yet. The way to learn programming is to program, just like swinging a baseball bat, playing the piano, playing well at bridge; you can hear the rules and talk about the strategies, but learning is best done by doing.

0.2.2 Difficulty 2: What is a good program?

Having mastered some of the syntax and semantics of a programming language, how do we write a good program? That is, how do we create a program that is more like poetry than like the mess arrived at through literal translation.

It is difficult to discuss a good program when, at this point, we know so little, but there are a couple of points that are worth noting even before we get started.

It's all about problem solving

If the rules of poetry are what guides writing good poetry, what are the guidelines for writing good programs? That is, what is it we have to learn in order to transition from a literal translation to a good poem?

For programming, it is **problem solving**. When you write a program, you are creating, in some detail, how it is that **you** think a particular problem some class of problems, should be solved. Thus, the program represents, in a very accessible way, your thoughts on problem solving. Your thoughts! That means, before you write the program you must **have** some thoughts.

It is a common practice, even among veteran programmers, to get a problem and **immediately** sit down and start writing a program. Typically that approach results in a mess, and, for the beginning programmer,

it results in an unsolved problem. Figuring out how to solve a problem requires some initial thought. If you think before you program, you better understand what the problem requires as well as the best strategies you might use to solve that problem.

Remember the two-level problem? Writing a program as you figure out how to solve a problem means you are working at two levels at once: the problem solving level and the programming level. That is more difficult than doing things sequentially. You should sit down and think about the problem and how you want to solve it **before** you start writing the program. We will talk more about this later, but the rule is:

> **RULE 1:** Think before you program!

A program as an essay

When students are asked "What is the most important feature a program should have", many answer "It should run". By "run", they mean that the program executes and actually does something.

Wrong. As with any new endeavor, it is important to get the fundamentals correct right at the beginning. So Rule 2 is

> **RULE 2:** A program is a human-readable essay on problem solving that also happens to execute on a computer.

A program is an object to be read by another person, just as any other essay. While it is true that a program is written in such a way that a computer can execute it, it is still a human-readable essay. If your program is written so that it runs, even runs correctly (notice we have not discussed "correctly" yet!) but is unreadable, then it is really fairly worthless.

The question is why? Why should it be that people must read it? Why isn't running good enough? Who's going to read it anyway? Let's answer the last question first. The person who is going to read it the most is you! That's correct, **you** have to read the programs you are writing all the time. Every time you put your program away for some period of time and come back to it, you have to re-read what you wrote and understand what you were thinking. Your program is a record of your thoughts on solving the problem and you have to be able to read your program in order to work with it, update it, add to it, etc.

Furthermore, once you get out of the academic environment where you write programs solely for yourself, you will be writing programs with other people as a group. Your group mates have to be able to read what you wrote! Think of the process as developing a newspaper edition. Everyone has to be able to read each others' content so that the edition, as a whole, makes sense. Just writing words on paper isn't enough—they have to fit together.

Our goal is to write programs that other people can read, as well as be run.

0.2.3 The Promise of a Computer Program

A program is an essay on problem solving and that will be our major focus. However, it is still interesting that programs do indeed run on a computer. That is, in fact, one of the unique, and most impressive parts about a program. Consider that idea for a moment. You can think of a way to solve a problem, write that thought down in detail as a program, and (assuming you did it correctly) that problem gets solved. More importantly, the problem can be solved again and again because the program can be used **independent of you**. That is, your thoughts not only live on as words (because of the essay-like nature of the program), but also as an entity that actually implements those thoughts. How amazing is that! Do you have some thoughts about how to make a robot dance? Write the program, and the robot dances. Do you have some ideas on how to create music? Write the program, and music is written automatically. Do you have an idea of how to solve a Sudoku puzzle? Write the program, and every puzzle is solved.

So a computer program (and the computer it runs on) offers a huge leap forward, perhaps the biggest since Gutenberg in the mid 1400's. Gutenberg invented moveable type, so that the written word of an individual, i.e. their thoughts, could be reproduced independently of the writer. Now, not only can those thoughts be copied, but also implemented to be **used** over and over again.

Programming is as open, as universally applicable, as the thoughts of the people who do the programming, because a program is the manifest thought of the programmer.

0.3 Choosing a Computer Language

We have selected a particular programming language, the language called Python, for this introductory text. You should know that there are lots of programming languages out there. Wikipedia [2] lists more than 500 such languages. Why so many languages, and why Python for this text?

0.3.1 Different Computer Languages

If a program is a concrete, runnable realization of a person's thoughts, then it makes sense that different people would create languages that allow them to better reflect those thoughts. In fact, computer scientists are crazy about languages which is why there are so many. Whatever the reason that some language was created, and there can be many reasons, they all reflect some part of the creator's view of how to best express solving problems on a computer. In fact, computer scientists are specifically trained to write their own language to suit their needs—a talent few other disciplines support so fully!

So given all these languages, why did we pick Python?

0.3.2 Why Python?

> *"I set out to come up with a language that made programmers more productive."*
> Guido van Rossum, author of Python

There are three features that we think are important for an introductory programming language:

- The language should provide a low "cognitive load" on the student. That is, it should be as easy as possible to express your problem-solving thoughts in the mechanisms provided by the programming language.
- Having spent all your time learning this language, it should be easy to apply it to problems you will encounter. In other words, having learned a language you should be able to write short programs to solve problems that pop up in your life (sort your music, find a file on your computer, find the average temperature for the month, etc.).
- The programming language you use should have broad support across many disciplines. That is, the language should be embraced by practitioners from many fields (arts to sciences as they say) and that useful packages, collections of support programs, be available to many different types of users.

So how does Python match up against these criteria?

[2] `http://en.wikipedia.org/wiki/Alphabetical_list_of_programming_languages`

Python Philosophy

Python offers a philosophy: *There should be one—and preferably only one—obvious way to do it.* The language should offer, as much as possible, a one-to-one mapping between the problem-solving need and the language support for that need. This is not necessarily the case with all programming languages. Given the two-level problem the introductory programmer already faces, reducing the programming language load as much as possible is important. Though Python does have its short cuts, they are far fewer than many other languages, and there is less "language" one has to remember to accomplish your task. In fact, the Python developers are working to create new versions of the language (Python 3.0, Appendix D) that brings the language even more in line with this philosophy.

A "Best Practices" Language

One of our favorite descriptions of Python is that it is a "best practices" language. This means that Python provides many of the best parts of other languages directly to the user. Important data structures are provided as part of the standard language, iteration (described later) is introduced early and is available on standard data structures, packages for files, file paths, web, etc. are part of the standard language. Python is often described as a "batteries included" language where many commonly needed aspects are provided by default. This characteristic means that you can use Python to solve problems you will encounter.

Python is Open Source

One of Python's most powerful features is it's support by the various communities and the breadth of that support. One reason is that Python is developed under the **Open Source** model. Open Source is both a way to think about software and a culture or viewpoint used by those who develop software. Open Source for software is a way to make software freely available and to guarantee that free availability to those who develop new software based on Open Source software. Linux, a type of operating system, is such a project as are a number of other projects such as Firefox (web browser), Thunderbird (mail client), Apache (web server) and, of course, Python. As a culture, Open Source adherents want to make software as available and useful as possible. They want to share the fruits of their labor and, as a group, move software forward, including the application areas where software is used. This perspective can be summarized as follows:

> A rising tide lifts all boats.

Each person working explicitly (or implicitly) as part of a larger group towards a larger goal, making software available and useful. As a result of Python's Open Source development, there are free, specialized packages for almost any area of endeavor including: music, games, genetics, physics, chemistry, natural language, geography, etc. If you know Python and can do rudimentary programming, there are packages available that will support almost any area you care to choose.

0.3.3 Is Python the Best Language?

The answer to that question is that there is no "best" language. All computer programming languages are a compromise to some degree. After all, wouldn't it be easiest to describe the program in your own words and just have it run? Unfortunately, that isn't possible. Our present natural language (English, German, Hindi, whatever) is too difficult to turn into the precise directions a computer needs to execute a program. Each programming language has it's own strengths and weaknesses. New programmers, having mastered their first programming language, are better equipped to examine other languages and what they offer. For now, we think Python is a good compromise for the beginning programmer.

0.4 What is Computation?

It can be difficult to find a good definition for a broadly-used word like "computation." If you look, you will find definitions that include terms like "information processing," "sequence of operations," "numbers," "symbols," and "mathematical methods." Computer Scientists interested in the theory of computation formally define what a computation is, and what its limits are. It is a fascinating topic, but it is a bit beyond our scope.

We will use an English language definition that suits our needs. In this book we will define a computation as:

> *Computation is the manipulation of data by either humans or machines.*

The data that is manipulated may be numbers, characters, or other symbols.

0.5 What is a computer?

The definition of a computer then is

> *A computer is something that does computation.*

That definition is purposefully vague on **how** a computer accomplishes computation, only that it does so. This imprecision exists because what counts as a computer is surprisingly diverse. However, there are some features that almost any system that does computation should have.

- A computer should be able to accept input. What counts as input might vary among computers, but data must be able to enter the computer for processing.
- If a computer is defined by its ability to do computation, then any object that is a computer must be able to manipulate data.
- A computer must be able to output data.

Another characteristic of many computers is their ability to perform computation using an assembly of only simple parts. Such computers are composed of huge numbers of simple parts assembled in complex ways. As a result, the complexity of many computers is the organization of their parts, not the parts themselves.

0.5.1 Computation in Nature

There are a number of natural systems that perform computation. These are areas of current, intense investigation in computer science as researchers try to learn more from existing natural systems.

The Human Brain

When electronic computers were first brought to the public's attention in the 1950's and even into the 1960's, they were often referred to as "electronic brains." The reason for this was obvious. The only computing object we had known up to that time was the human brain.

The human brain is in fact a powerful computing engine, though it is not often thought of in quite that way. It constantly takes in a torrent of input data—sensory data from the five senses—manipulates that data in a variety of ways, and provides output in the form of both physical action (both voluntary and involuntary) as well as mental action.

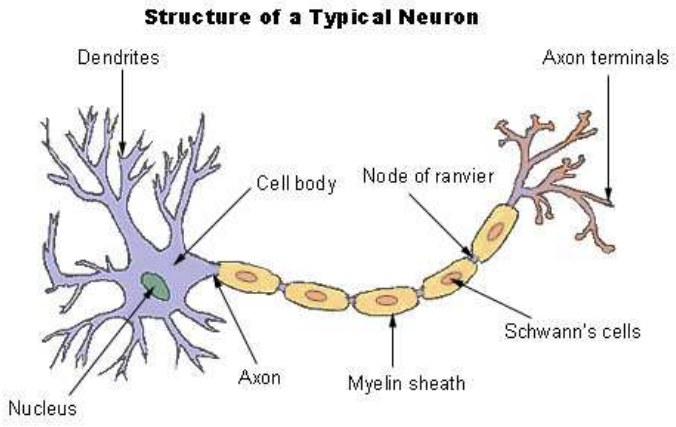


Figure 3: An annotated neuron.

The amazing thing about the human brain is that the functional element of the brain is a very simple cell called the *neuron*, Figure 3.

Though a fully functional cell, a neuron also acts as a kind of small switch. When a sufficient signal reaches the *dendrites* of a neuron, the neuron "fires" and a signal is transmitted down the *axon* of the neuron to the *axon terminals*. Neurons are not directly connected to one another, rather the dendrites of one neuron are located very close to the axon terminals of another neuron, separated by a space known as the *synapse*. When the signal reaches the axon terminal, chemicals called *neurotransmitters* are secreted across the synapse. It is the transmission of a signal, from neurotransmitters secreted by the axon terminals of one neuron to the dendrites of a connecting neuron that constitutes a basic computation in the brain.

Here are a few interesting facts about neuronal activity:

- The signal within a neuron is not transmitted by electricity (as in a wire), but by rapid chemical changes that propagate down the axon. The result is that the signal is very much slower than an electrical transmission down a wire—about a million time slower.
- Because the signal is chemical, a neuron must typically recover for a millisecond (one-thousandth of a second). before it can fire again. Therefore, there is a built-in time delay to neuronal firing.
- A neuron is "fired" in response to some combination of the number of signals that are received at its dendrite (how many other neurons are firing in proximity) and the strength of the signal received (how much neurotransmitter is being dumped into the synapse).

One thing to note is how **slow** the neuron is as a switch. As we shall see in Section 0.6.2, electronic switches can act many millions of times faster. However, what the brain lacks in terms of speedy switches it makes up for in sheer size and complexity of organization. By some estimates, the human brain consists of of 100 billion (10^{11}) neurons and 100 trillion (10^{14}) synapses. Furthermore, the organization of the brain is incredibly complicated. Scientists have spent hundred's of years identifying specific areas of the brain that are responsible for various functions, and how those areas are interconnected. Yet for all this complexity, the main operational unit is a tiny, slow switch.

Scientists have been fascinated by the brain, so much so that there is a branch of computer science that works with simulations of *neural networks*, networks consisting of simple switches such as those found in the brain. Neural networks have been used to solve many difficult problems.

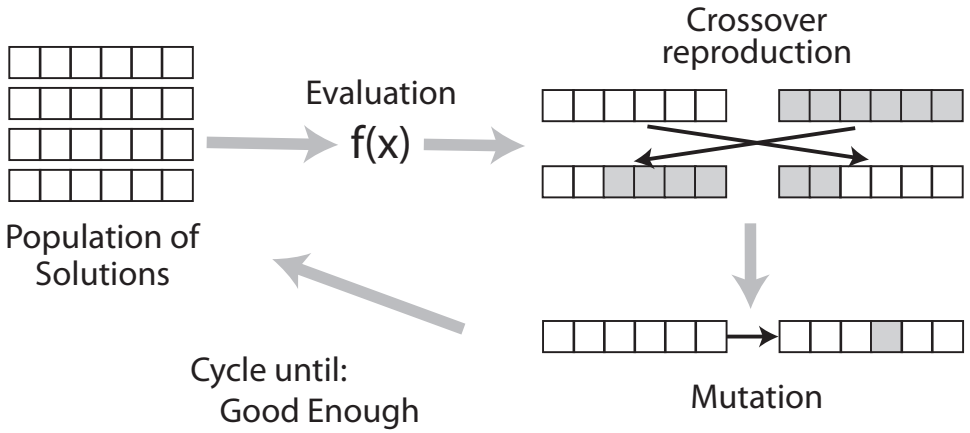


Figure 4: A genetic algorithm.

Evolutionary Computation

The evolution of biological species can be viewed as a computation process. In this view, the inputs of the computational process are the environmental variables the biological entity is subjected to; the computational process is the adaptation of the genetic code and the output is the adaptation results from the genetic code modification.

This point of view has been incorporated into approaches known as *Genetic Algorithms*. These techniques take the concepts of simple genetics, as proposed by Gregor Mendel, and the processes of evolution as described by Charles Darwin, and use them to compute.

The basic parts of a genetic algorithm, shown in Figure 4, are:

- A way to encode a solution in a linear sequence, much like the sequence of information contained in a chromosome. This encoding depends on the problem, but it typically consists of parameters (struts for a bridge, components for a circuit, jobs for a schedule, etc.) that are required to constitute a solution.
- A method to evaluate the "goodness" of a particular solution, called the *evaluation function* and represented as $F(x)$ in the diagram.
- A population of solutions, often initially created by random selection of the solution components, from which new solutions can be created.
- Some genetic modification methods to create new solutions from old solutions. In particular there is *mutation* which is a modification of one aspect of an existing solution and *crossover* which combines aspects of two parent solutions into a new solution.

The process proceeds as follows. Each solution in the population is evaluated to determine how fit it is. Based on the fitness of the existing solutions, new solutions are created using the genetic modification methods. Those solutions that are more fit are given more opportunity to create new solutions while less fit solutions are given less opportunity. This process incorporates a "survival of the fittest" notion. Over time, better and better solutions are evolved that solve the existing problem.

Genetic Algorithms and other similar approaches have been used to solve many complex problems such as scheduling, circuit design, etc.

Figure 5: NACA High-Speed-Flight Station Computer Room.

0.5.2 The Human Computer

The common use of the word "computer" from around the 17^{th} century to about World War II referred to **people**. To compute difficult, laborious values for mathematical constants (such as π or e), fission reactions (for the Manhattan project) and gun trajectory tables, people were used (Figure 5). However, using people had problems.

A classic example of such a problem was created by William Shanks, an English amateur mathematician, who in 1873 published π calculated to 707 decimal places. The calculation by hand took 28 years. Unfortunately, he made a calculation error at the 528^{th} digit which made the last two years of calculation a waste of time. His error wasn't found until seventy years later using a mechanical calculator.

The U.S. Army's Ballistics Research Laboratory was responsible for the creation of gun trajectory tables. Each new artillery piece required a table for the gunner to use to calculate where the round would land. However, it took a lot of human effort to make these tables. Kay Mauchly Antonelli, a Mathematician from the University of Pennsylvania said, "To do just one trajectory, at one particular angle, usually took up between 30 to 40 hours of calculation on this [mechanical] desk calculator." These tables exceeded 1800 entries and required up to four years to produce by hand.[3]

It was obvious that something had to be done. People were neither accurate enough, nor fast enough, to do this kind of calculation. A more modern, faster, and accurate approach was needed.

[3] http://www.comphist.org/pdfs/Video-Giant%20Brains-MachineChangedWorld-Summary.pdf.

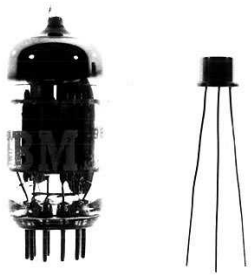


Figure 6: Vacuum tube, single transistor, and chip transistor (the dot) [IBM, 1964].

0.6 The Modern, Electronic Computer

While there may be many notions of a computer, we all know what a modern computer is. We read email on it, text message each other, listen to music, play videos, and play games on them. The design of these computers was first conceived around World War II to solve those tedious calculation problems that humans did so slowly, especially the Army's ballistics tables. How do electronic computers work? What makes them so special?

0.6.1 It's the Switch!

Modern digital computers use, as their base component, nothing more complicated than a simple switch. Very early computers used mechanical switches or relays, while later versions used vacuum tubes and finally, modern computers use transistors (see Figure 6).

A switch's function is pretty obvious. It is either on or off. When turned on, electricity flows through the switch and when off, no electrical flow can occur. Using a switch and its on/off nature, you can construct some simple **logic** circuits. In logic, we have only two states, True or False. In our logic circuit, we translate True and False to the physical process of a switch. The True condition is represented by a current flowing through the circuit and False is represented by lack of current flow. Though we will cover Boolean logic in more detail in Chapter 2, there are two simple combinations of switches we can show, the **and** and **or** circuits.

For the Boolean **and**, a True value results *only* if both of the two input values are True. All other combinations of input have a False output. In the **and** circuit, we connect two switches together in series as shown in Figure 7. Electricity can flow, that is the circuit represents a True value, only if *both* switches are turned on. If either switch is turned off, no electricity flows and the circuit represents a False value.

We can do the same for the Boolean **or**. An **or** is True if *either one or both* of its inputs are True, otherwise it is False. In the **or** circuit of Figure 7, we connect two switches together in parallel. Electricity can flow if *either* switch is turned on representing a True value. Only when both switches are turned off does the circuit represent a False value.

Similar Boolean logic elements, usually called logic **gates**, can be constructed from simple circuits. The amazing thing is that, using only simple logic circuits we can assemble an entire computer. For example an *and* and *or* gate can be combined to make a simple circuit to add two values, called an adder. Once we can add, we can use the adder circuit to build a subtraction circuit. Further, once we have an adder circuit we can do multiplication (by repeated addition), and then division, and so on. Just providing some simple elements like logic gates allows us to build more complicated circuits until we have a complete computer.

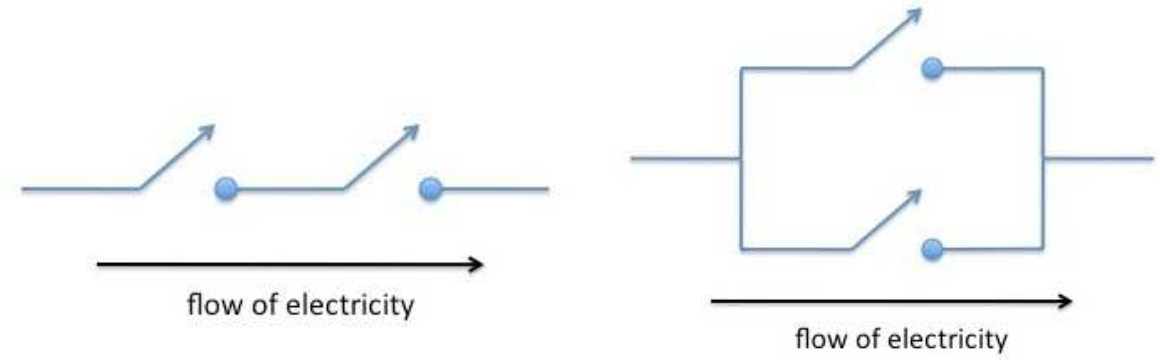


Figure 7: Switches implementing an "and" gate (left) and an "or" gate (right).

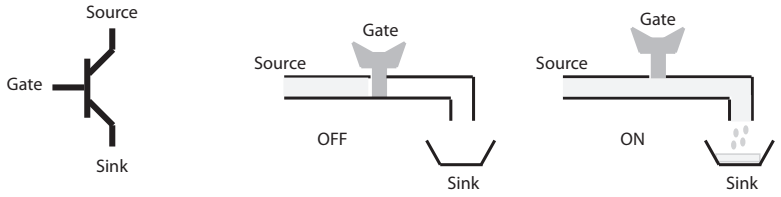


Figure 8: A diagram of a transistor and its equivalent "faucet" view.

0.6.2 The Transistor

While any switch will do, the "switch" that made the electronic computer what it is today is called a transistor. The transistor was an electronic device invented in 1947 by William Shockley, John Bardeen and Walter Brattain at Bell Labs (for which they eventually won the Nobel Prize in Physics in 1956). It utilized a new technology, a material called a semiconductor, that allowed transistors to supersede the use of other components such as vacuum tubes. Though a transistor has a number of uses, the one we care most about for computer use is as a switch. A transistor has three wires with the names Source, Sink and Gate. Electricity flows from the Source to the Sink. If there is a signal, a voltage or current, on the Gate then electricity flows—switch is "on." If there is no signal on the Gate, no electricity flows—switch is "off." See Figure 8.

In the switch examples above, we can use transistors as switches, e.g. as in Figure 7. In that way, transistors are used to construct logic gates, then larger circuits and so on, eventually constructing the higher level components that become the parts of a modern electronic computer.

What makes the transistor so remarkable is how it has evolved in the 60 years since its first creation. It is this remarkable evolution that has made the modern computer what it is today.

Smaller

The size of a transistor has changed dramatically since its inception. The first Shockley transistor was very large, on the order of inches (see Figure 9). By 1954, Texas Instrument was selling the first commercial transistor, and had shrunk the transistor size to a postage stamp.

However, even the small size of individual transistor components was proving to be a limitation. More transistors were needed in a smaller area if better components were to be designed. The solution was the *integrated circuit*, invented by Jack Kilby of Texas Instruments in 1958-1959 (for which he won the 2000 Nobel Prize in Physics), Figure 10. The integrated circuit was a contiguous piece of semiconductor material upon

Figure 9: The Shockley transistor—the first transistor.

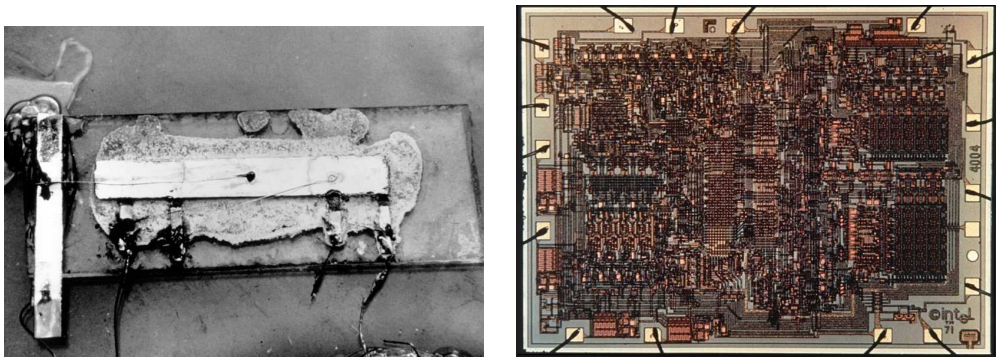


Figure 10: Kilby's first integrated circuit (left) and the Intel 4004 microprocessor (right).

which multiple transistors could be manufactured. The integrated circuit allowed many transistors to be embedded in a single piece of material, allowing a more complex functional circuit on a very small area. By 1971, Intel managed to manufacture the first complete computer processing unit (or CPU) on a single chip, a microprocessor named the Intel 4004 (see Figure 10). It was approximately the size of a human fingernail ($1/8$ by $1/6$ of an inch) with 2300 transistors. By this time the size of the transistor on this microprocessor had shrunk to 10 microns, the width of the a single fiber of cotton or silk.

The shrinkage of the transistor has continued. Today, the size of a transistor has reached amazingly small levels. Figure 11 shows an electron microscope picture of a gate recently developed by IBM. The size of the gate is 50 nanometers, 50×10^{-9} meters, a 1000-times smaller than transistors in the Intel 4004. That is more than 10 times smaller than a single wavelength of visible light. It is approximately the thickness of a cell membrane, and only 10 times larger than a single turn of a DNA helix.

Quantity and Function

As the size of a transistor shrank, the number of transistors that could be put on a single chip increased. This increase has been quite dramatic. In fact, there is a famous statement made by the founder of Intel, Gordon Moore, that predicts this amazing trend. In 1965, Moore predicted that the number of transistors that can be placed inexpensively on a single chip would double about every two years.[4] This trend has proven to be

[4]The original estimate was one year, but was later revised upward.

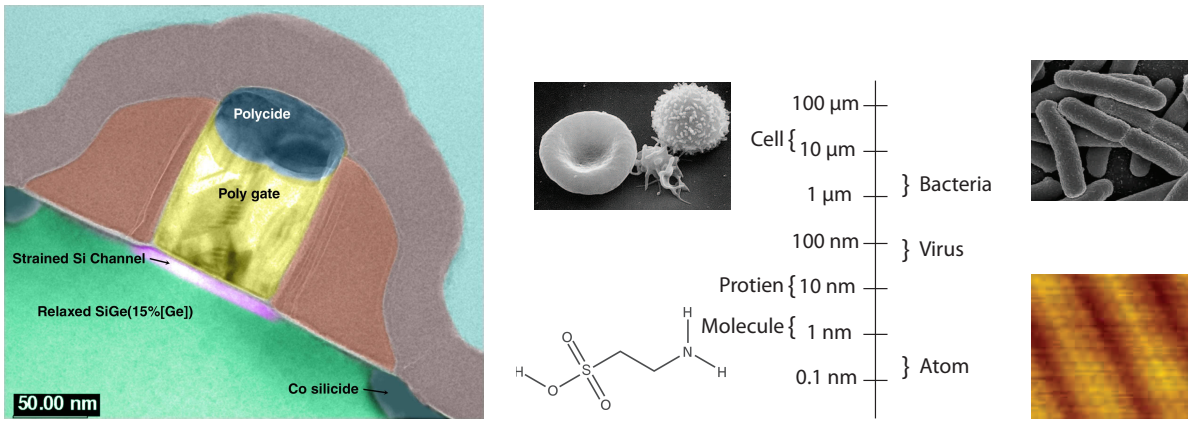


Figure 11: A photomicrograph of a single 50nm IBM transistor (left) and a scale small items(right).

Year	Transistor Count	Model
1971	2,300	4004
1978	29,000	8086
1982	134,000	80286
1986	275,000	80386
1989	1,200,000	80486
1993	3,100,000	Pentium
1999	9,500,000	Pentium III
2001	42,000,000	Pentium 4
2007	582,000,000	Core 2 Quad
2008	2,100,000,000	Itanium Quad

Table 1: Transistor counts in Intel microprocessors, by year.

remarkably accurate and continues to hold to this day. Named in the popular press as "Moore's Law", it's demise has been predicted for many years but continues to hold true. A summary of this trend is shown in the Table 1.

By increasing the number of transistors in a CPU, more functionality can be introduced on each CPU chip. In fact, recently the **number** of CPU's on a single chip has also increased. A common chip in production at the writing of this book is the quad-core processor (see Figure 12) which contains four complete CPUs.

Faster

Smaller transistors are also faster transistors so smaller transistors provide a doubling factor: more and faster. But how does one measure speed?

When you buy a computer, the salesman is more than happy to tell you how fast it is, usually in terms of how many "Gigahertz" the computer will run. The value the salesman mentions is really a measure of a special feature of every computer called its *clock*. The clock of a computer is not much like a wall clock however. Rather, it is more like a drummer in a band. The drummer's job in the band is to keep the beat, coordinating the rhythm of all the other members of the band. The faster the drummer beasts, the faster the band plays. The components on a CPU chip are like the band—they need something to help them coordinate their efforts. That is the role of the clock. The clock regularly emits a signal indicating that the next operation is to occur. Upon every "beat" of the clock, another "cycle" occurs on the chip, meaning another set of operations can occur.

Figure 12: Intel Nehalem Quad Core Processor

Therefore, the faster the clock runs, the faster the chip runs and, potentially, the more instructions that can be executed. But how fast is a Gigahertz (GHz)? Literally, 1 Gigahertz means that the clock emits a signal every nanosecond, that is every billionth of a second (10^{-9}). Thus, your 1 GHz computer executes instructions once every nanosecond. That is a very fast clock indeed!

Consider that the "universal speed limit" is the speed of light, roughly 186,282 miles/second (299,792,458 meters/second) in a vacuum. Given that an electrical signal (data) is carried at this speed, how far can an electric signal travel in a nanosecond? If we do the math, it turns out that an electric signal can only travel about 11.8 inches. Only 11.8 inches! At 2 GHz, it can only travel 5.9 inches; at 4GHz, only 2.95 inches. Consider that 5.9 inches is not even the width of a typical computer board! In fact, at the writing of this book a further doubling of speed of present computers is limited by the distance electricity can travel.

Given a measure of the speed of a computer, how does that translate to actual work done by the computer? The process of measuring computing operations, known as "benchmarking", can be a difficult task. Different computing systems are better or worse, depending on how you measure the benchmark. However, manufacturers do list a measure called **Instructions per second** or **IPS**. That is, how many instructions such as an addition can be done every second. In Table 2, we can see how the measure of increased clock rate affects the IPS (MIPS is millions of IPS). One interesting note is that, since about 2000, the clock speed has not increased dramatically for the reasons mentioned above. However, the existence of multiple CPUs on a chip still allows for increases in IPS of a CPU.

0.7 A High Level look at a Modern Computer

Now that we know something about the low level functional elements of a computer, it is useful to step up to a higher level view of the elements of a computer, often termed the computer's **architecture**. The architecture is used to describe the various parts of the computer and how they interact. The standard architecture, named after the stored program model of John von Neumann, looks something like the following:

- **Processor:** As we have mentioned, the processor is the computational heart of a computer. Often called the CPU (Central Processing Unit) it can itself consist of a number of parts including the ALU (Arithmetic and Logic Unit) where logical calculations are done, local fast storage called Cache, connections between components called the Bus, as well as other elements
- **Main Memory:** A processor needs data to process and main memory is where data is stored. Main memory is traditionally volatile, i.e. when power is turned off, data is lost. It is also called RAM

Year	CPU	Instructions/second	Clock Speed
1971	Intel 4004	1 MIPS	740 kHz
1972	IBM System/370	1 MIPS	?
1977	Motorola 68000	1 MIPS	8 MHz
1982	Intel 286	3 MIPS	12 MHz
1984	Motorola 68020	4 MIPS	20 MHz
1992	Intel 486DX	54 MIPS	66 MHz
1994	PowerPC 600s (G2)	35 MIPS	33 MHz
1996	Intel Pentium Pro	541 MIPS	200 MHz
1997	PowerPC G3	525 MIPS	233 MHz
2002	AMD Athlon XP 2400+	5,935 MIPS	2.0 GHz
2003	Pentium 4	9,726 MIPS	3.2 GHz
2005	Xbox360	19,200 MIPS	3.2 GHz
2006	PS3 Cell BE	10,240 MIPS	3.2 GHz
2006	AMD Athlon FX-60	18,938 MIPS	2.6 GHz
2007	Intel Core 2 QX9770	59,455 MIPS	3.2 GHz

Table 2: Speed (in millions of IPS) and clock rates of microprocessors, by year

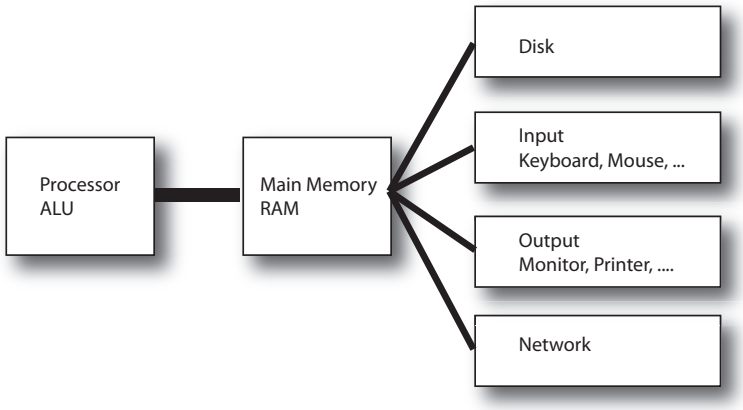


Figure 13: A typical computer architecture.

= Random Access Memory, i.e. retrievable in any order. Use of non-volatile memory is increasing, especially in portable devices.

- **Disk:** The disk is for permanent (non-volatile) storage of data. The disk is also known as the "hard drive." Data must be moved from the disk to main memory before it can be used by the processor. Disks are relatively slow, mechanical devices that are being replaced by non-mechanical memory cards in some portable devices.
- **Input/Output:** These devices convert external data into digital data and vice versa for use and storage in the computer.
- **Network:** A network is needed to communicate with other computers. From the viewpoint of the processor the network is simply another input/output device.

Consider a simple operation $theSum = num1 + num2$. The $theSum$, $num1$, and $num2$ terms are called **variables**, readable names that contain values to be used in a program. The statement adds the two numbers stored in $num1$ and $num2$ to produce a result which is stored in $theSum$. Assume that $num1$ and $num2$ represent numbers that are stored on the disk and that the result $theSum$ will also be stored on the disk.

Assume that the instruction itself $theSum = num1 + num2$ also resides on the disk. Here is how it works:

1 **Fetch Instruction:** When the processor is ready to process an instruction it fetches the instruction from memory. If the instruction is not in memory but on the disk, the memory must first fetch it from the disk. In this way, the instruction $theSum = num1 + num2$ will move from the disk through memory to the processor.

2 **Decode Instruction:** The processor examines the instruction ("decodes" it) and sees that operands $num1$ and $num2$ are needed so it fetches them from memory. If $num1$ and $num2$ are not in memory, but on the disk, the memory must first fetch them from the disk.

3 **Execute Operation:** Once the processor has both the instruction and the operands it can perform the operation, addition in this case.

4 **Store Result:** After calculating the sum the processor will store the resulting sum in memory. At some point before power is turned off, the data in memory will be stored on the disk.

5 **Repeat:** Go back to Fetch Instruction to fetch the next instruction in a program.

The Fetch-Decode-Execute-Store cycle is fundamental for all computers. This simple, sequential process is the basis for all modern computers. These four steps are done in lockstep to the beat of a clock as we described previously.

How complex is each operation? Not very. The ALU of a processor can add, subtract, multiply and divide. It can also compare values and choose which instruction to do next based on that comparison. That's it. In reality, it is slightly more complex than that, but not much.

Also, the processor can only handle two types of operands: integers and floating points. For our purposes, you can think of floating point values as fractional values represented in decimal notation, e.g. 37.842. There will be a separate ALU for integers and a separate one for floating point numbers called the FPU = Floating Point Unit.

The important concept is that everything a computer does boils down to a few simple operations, but at billions of operations per second the result is significant computational power.

0.8 Representing Data

The underlying element of a computer is typically a switch, usually a transistor. Given that, what is the most obvious way to represent data values? The obvious choice is binary. Binary values can only be either 1 or 0, which corresponds to the physical nature of a switch, which is on or off. What is interesting is that we can represent not only numbers in binary, but music, video, images, characters and many other kinds of data also in binary.

0.8.1 Binary Data

By definition a digital computer is binary which is base 2. Our normal number system is decimal, base 10, probably because we have 10 fingers for counting. People haven't always worked in base 10. For example, the ancient Babylonians (2000 BC) used base 60 (sexagesimal) for the most advanced mathematics of that time. As a result, they are the source of modern time keeping and angle measurement: 60 seconds in a minute, 60 minutes in an hour, and 360 degrees in a circle. For decimals we use ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. For sexagesimal the Babylonians used sixty digits: 0, 1, 2, 3, . . . , 58, 59. For binary there are only two digits: 0, 1.

Why binary? Two reasons really. As we have said, the first reason is the hardware being used. Electronic transistors lend themselves very naturally to base two. A transistor is either on or off, which can be directly translated to 1 or 0. However, the second reason is that two digits are easy to store and easy to operate on. Storage devices in binary need a medium that has two states: a state called "one" and another state "zero". Anything with two states can become digital storage. Examples include: high/low voltage, right/left magnetism, charge/no-charge, on/off, etc. For example, main memory has small capacitors which hold a charge (1) or not (0). Disks are magnetized one way (1) or the other (0). CDs and DVDs reflect light one way (1) or the other (0).

Manipulations of the underlying data can also be done simply using electronic gates that we discussed previously, the Boolean logic: and, or, not. Because such logical circuits can be extremely small and fast, they can be implemented to do calculations quickly and efficiently. For example, the adder circuit we discussed previously that adds two binary digits or **bits** (bit = BInary digiT) can be done with logical circuits: sum = (A and (not B)) or ((not A) and B)). From such simple logic all arithmetic operations can be built. For example, subtraction is the addition of a negative value, multiplication is repeated addition, and division can be done using the other three operations. A choice can be made based on the value of a bit: choose one thing or another. That choice bit can be calculated using any arbitrary logical expression using the logical operators: and, or, not. Therefore, the entire ALU (and the rest of the computer) can be built from the logical operators: and, or, not.

0.8.2 Working with Binary

A brief look at the binary representation of numbers and characters provides useful background for understanding binary computation. Since our world is a world of decimal numbers, let's look at representing decimals in binary. We begin with a review of place holding in decimals, by taking you back to elementary school. For example, consider the number 735 in base 10 (written as 735_{10}). Notice in the last line how the exponents start at 2 and work down to 0 as you move left-to-right.

$$\begin{aligned} 735_{10} &= 7\ hundreds + 3\ tens + 5\ ones \\ 735_{10} &= 7 * 100 + 3 * 10 + 5 * 1 \\ 735_{10} &= 7 * 10^2 + 3 * 10^1 + 5 * 10^0 \end{aligned}$$

In binary we only have two digits: 0, 1. Also, our base is 2 rather than 10. Therefore, our rightmost three places are: fours, twos, ones. As with base ten, the exponents will decrease as we move from left to right—in this case: 2, 1, then 0. Using the above notation, but working backwards from the last line to the first line let us determine what 101 in binary (101_2) is in base 10 (decimal).

$$\begin{aligned} 101_2 &= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ 101_2 &= 1 * 4 + 0 * 2 + 1 * 1 \\ 101_2 &= 4 + 0 + 1 \\ 101_2 &= 5_{10} \end{aligned}$$

In a similar way, any decimal integer can be represented in binary. For example,

$$1052_{10} = 10000011100_2$$

Fractions can be represented using integers in the Scientific Notation you learned in science classes.

$$1/8 = 0.125 = 125 * 10^{-3}$$

The mantissa (125) and exponent (-3) are integers that can be expressed and stored in binary. The actual implementation of binary fractions is different because the starting point is binary, but the principle is the same: binary fractions are stored using binary mantissas and binary exponents. How many bits are allocated to mantissa and how many to exponents varies from computer to computer, but the two numbers are stored together.

There are four important concepts that you need to know about representation:

- All numbers in a computer are represented in binary.
- Because of fixed, hardware limits on number storage, there is a limit to how big an integer can be stored in one unit of computer memory (usually 32 or 64 bits of storage).
- Fractions are represented in Scientific Notation and are approximations.
- Everything is converted to binary for manipulation and storage: letters, music, pictures, etc.

0.8.3 Limits

We have covered the representation of numbers in binary. Let's look at limits. Most computers organize their data into **words** that are usually 32-bits in size (though 64-bit words are growing in use). There are an infinite number of integers, but with only 32-bits available in a word, there is a limited number of integers that can fit. If one considers only positive integers, one can represent 2^{32} integers with a 32-bit word or a little over 4 billion integers. To represent positive and negative integers evenly, the represented integers range from negative 2 billion to positive 2 billion. That is a lot of numbers, but there is a limit and it is not hard to exceed it. For example, most US state budgets will not fit in that size number (4 billion). On the other hand, a 64-bit computer could represent 2^{64} integers using a 64-bit word which in base 10 is 1.8×10^{19}: a huge number—over 4 billion times more than can be stored in a 32-bit word.

Fractional values present a different problem. We know from mathematics that between every pair of Real numbers there are an infinite number of Real numbers. To see that, choose any two Real numbers A and B, and (A + B)/2 is a Real number in between. That operation can be repeated an infinite number of times to find more Real numbers between A and B. No matter how we choose to represent Real numbers in binary the representation will always be an approximation. For example, if you enter 19/5.0 into Python, it will be calculated as 3.7999999999999998 instead of the expected 3.8 (try it!). The approximation is a feature of storage in binary rather than a feature of Python.

Bits, Bytes and Words

Computer words (as opposed to English words) are built from **bytes** which contain 8 bits so one 32-bit word is made of 4 bytes. Storage is usually counted in bytes, e.g. 2 GBytes of RAM is approximately 2 billion bytes of memory (actually, it is 2^{31} bytes, which is 2,147,483,648 bytes). Bytes are the unit of measurement for size mainly for historical reasons.

0.8.4 Representing Letters

So far we've dealt with numbers. What about letters (characters): how are characters stored in a computer? Not surprisingly, everything is still stored in binary, which means it is still a number. Early developers created ways to map characters to numbers.

First, what is a character? Characters are what we see on a printed page and are mostly made from letters (a, b, c, ...), digits (0, 1, 2, ...), and punctuation (period, comma, semicolon, etc.). However, there are also characters that are not printable such as 'carriage return' or 'tab' as well as characters that existed at the dawn of computing to control printers such as 'form feed.' The first standardized set of computer characters was the ASCII (American Standard Code for Information Interchange) set shown in Table 3 developed in 1963.

Char	Dec	Char	Dec	Char	Dec	Char	Dec
NUL	0	SP	32	@	64	`	96
SOH	1	!	33	A	65	a	97
STX	2	"	34	B	66	b	98
ETX	3	#	35	C	67	c	99
EOT	4	$	36	D	68	d	100
ENQ	5	%	37	E	69	e	101
ACK	6	&	38	F	70	f	102
BEL	7	'	39	G	71	g	103
BS	8	(	40	H	72	h	104

Table 3: Table of ASCII characters (first few rows–see Appendix E).

The ASCII table is a mapping between numbers and characters. Each character has an associated number, that is "A" is 65 while "a" is 97. Knowing this relationship, we can interpret a set of numbers as characters and then manipulate those characters just as we would numbers.

Note that the ASCII set of characters was developed by English-speakers. As computing has spread across the globe a richer character set was developed to represent the different characters in languages such as Russian, Japanese and Chinese. Thus Unicode was developed to provide more room to represent the many characters in all languages, especially languages such as Chinese which can have more than a thousand characters. Python can handle both ASCII and Unicode, but for most of our purposes ASCII is sufficient and much simpler, so we only discuss ASCII here. We'll talk more about this topic in Chapter 4.

0.8.5 Representing other Data

You must get the idea by now: what computers can represent is numbers. If you want to represent other data, you must find a way to turn that data into numbers.

Images

How to store an image? If the image is discrete, that is built of many individual parts, we can represent those individual parts as numbers. Take a close look at your monitor or TV screen (with a magnifying glass if you can). The image is made up of thousands of very small dots. These dots, called **pixels** (short for picture elements) are used to create an image. Each pixel has an associated color. If you put a lot of these pixels together in a small area, they start to look like an image (see Figure 14)

Each pixel can be represented as a location (in a two dimensional grid) and as a color. The location is two numbers (which row and which column the pixel occupies), and the color is also represented as a number. While there are a variety of ways to represent color, a common way is to break each color into its contribution from the three basic colors: red, green and blue. The color number represents how much of each basic color is contributed to the final color. An 8-bit color scheme means that each color can contribute 8 bits, or $2^8 = 256$ possible shades of that color. Thus, a 24 bit color system can represent 2^{24} different colors, or 16,777,216.

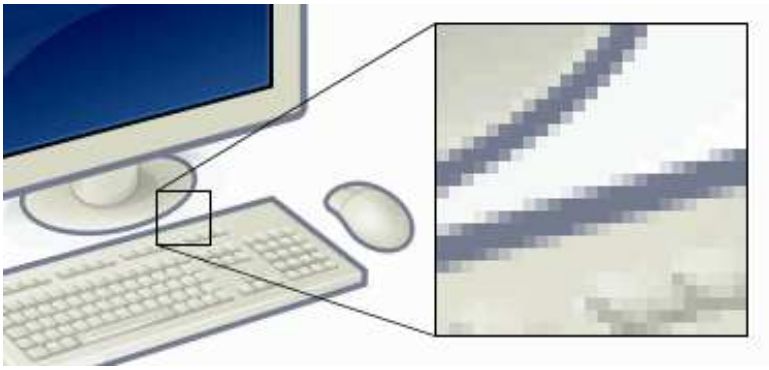


Figure 14: A computer display picture with a close-up of the individual pixels.

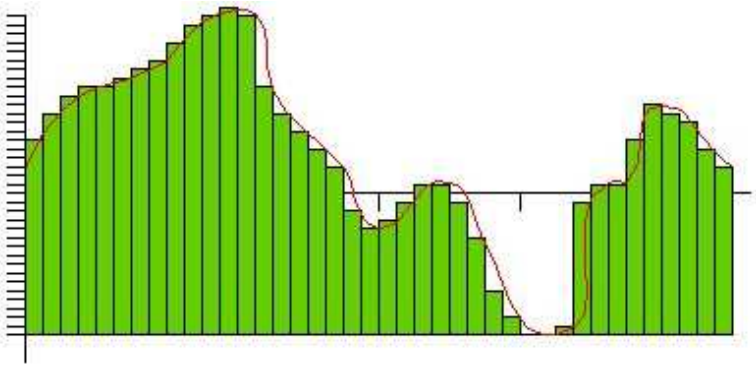


Figure 15: A sound wave and the samples of that sound wave (green bar height) over time

The quantity of pixels is important. Standard analog television (extinct as of the writing of this book) used 525 lines of pixels, with each line containing 480 pixels, a total of 252,000 pixels. High definition television have 1920×1080 pixels for a total of 2,073,600, a much higher resolution and a much better image.

Music

How to represent music as numbers? There are two types of musical sound we might want to capture: recorded sound and generated sound. First, let's look at recording sound. A sound "wave" is a complex wave of air pressure that we detect with our ears. If we look at the shape of this sound wave (say with an oscilloscope) we can see how complex the wave can be. However, if we record the height of that wave at a very high rate, say at a rate of 44,100 times/second (or 44kHz, the sampling rate on most mp3's), then we can record that height as a number for that time in the sound wave. Therefore, we record two numbers: the height of the recorded sound and the time when that height was recorded. When we play sound back, we can reproduce the sound wave's shape by creating a new sound wave that has the same height at the selected times as the recorded sound at each point in time. The more often we "sample" the sound wave, the better our ability to reproduce it accurately. See Figure 15 for an example.

To generate our own new sound, we can write computer programs that generate the same data, wave height at some point in time, and then that data can be played just like recorded sound.

0.8.6 What does a number represent?

We've shown that all the data we record for use by a computer is represented as a number, whether the data be numeric, text, image, audio or any other kind of data. So how can I tell what a particular recorded data value represents? You cannot by simply looking at the bits. Any particular data value can be interpreted as an integer, a character, a sound sample, etc. It depends on the use for which that data value was recorded. That is, it depends on the **type** of the data. We will talk more of type in Chapter 1, but the type of the data indicates for what use the recorded data values are to be used. If the type is characters, then the numbers represent ASCII values. If the type is floating point, then the numbers represent the mantissa and exponent of a value. If the type is an image, then the number would represent the color of a particular pixel. Knowing the **type** of the data lets you know what the data values represent.

0.8.7 How to talk about quantities of data

How much data can a computer hold, and what constitutes "a lot" of data? The answer to that question varies, mostly depending on when you ask it. Very much like Moore's law and processing speed, the amount of data a commercial disk can hold has grown dramatically over the years.

Again, there are some terms in common usage in the commercial world. In the context of data amounts, we talk about values like "Kilobytes" (abbreviated Kb) or "Megabytes" (abbreviate Mb) or "Gigabytes" (abbreviated Gb), but the meaning is a bit odd. "Kilo" typically refers to 10^3 or 1 thousand of something , "Mega" 10^6 or 1 million of something, and "Giga" usually to 10^9 or 1 billion of something, but it's meaning here is a little off. This is because of the fact that 10^3 or 1000 is *pretty close* to 2^{10} or 1024.

So in discussing powers of 2 and powers of 10, most people use the following rule of thumb. Any time you talk about multiplying by 2^{10}, that's pretty close to multiplying 10^3, so we will just use the power of ten prefixes we are used to. Another way to say it is that every 3 powers of 10 is "roughly equivalent" to 10 powers of 2. Thus a kilobyte is not really 1000 (10^3) bytes, but 1024 (2^{10}) bytes. A megabyte is not 1 million bytes (10^6), but 1,048,576 (2^{20}) bytes. A gigabyte is not 1 billion bytes (10^9), but 1,073,741,824 bytes (2^{30}).

0.8.8 How much data is that?

At the writing of this book, the largest standard size commercial disk is 1 Terabyte, 1 trillion bytes (10^{12}). On the horizon is the possibility of a 1 Petabyte, 1 quadrillion (10^{15}) bytes. Again, the growth in disk sizes is dramatic. The first disk was introduced by IBM in 1956 and held only 4 MegaBytes. It took 35 years before a 1 GigaByte disk was made, but only 14 more years until we had 500 GigaByte disks, and only two more years until a 1 TeraByte (1,000 GigaByte) disk was available. A petabyte disk (1,000,000 GigaBytes) is predicted for 2010.

So, how big is a PetaByte? Let's try to put it into terms you can relate to.

- A book holds roughly a megabyte of data. If you read one book a day every day of your life, say 80 years, that will amount to less than 30 gigabytes of data. Remember, a petabyte is 1 million gigabytes so you will still have 999,970 gigabytes left over.
- How many pictures can a person look at in a lifetime? We'll assume an average picture has 4 Megabytes of data. Let's say 100 images a day—about one picture every 10 waking minutes. After 80 years, that collection of snapshots would add up to 30 terabytes. So your petabyte disk will have 970,000 gigabytes left after a lifetime of high quality photos and books.
- What about music? MP3 audio files run a megabyte a minute, more or less (and that is compressed data). At that rate, a lifetime of listening—24 hours a day, 7 days a week for 80 years—would consume

42 terabytes of space. So with all your music and pictures and books for a lifetime you will have 928,000 gigabytes free on your disk. That is, almost 93% of your disk is still empty.

- The one kind of content that might possibly overflow a petabyte disk is video. In the format used on DVDs, the data rate is about two gigabytes per hour. Therefore, the petabyte disk will hold some 500,000 hours worth of video. If you want to record all day and all night without a break, the video will actually fill up your petabyte drive after about 57 years.

Of course, why stop there? More prefixes have been defined.

- exa is 2^{60} = 1,152,921,504,606,846,976
- zeta is 2^{70} = 1,180,591,620,717,411,303,424
- yotta is 2^{80} = 1,208,925,819,614,629,174,706,176

Does anything get measured in exabytes or even zetabytes? By 2006 the total Internet traffic in the US was estimated to be roughly 8.4 exabytes for the year. However, video is increasing Internet traffic dramatically. By the end of 2007 YouTube was estimated to be generating 600 petabytes per year all by itself. If YouTube were in high definition, it would generate 12 exabytes by itself (in 2007). Amateur video is estimated to capture 10 exabytes of video per year, but much of that does not show up on the Internet. NetFlix currently distributes movies by sending DVDs through the US mail. If they distributed through the Internet, that would consume 5.8 exabytes of Internet bandwidth per year. If the movies were high definition, they would consume 100 exabytes per year. Video conferencing and Internet gaming are also increasing dramatically. Taken together, these applications have the potential to create what has been termed the "Exaflood" of the Internet. The most extreme estimates put the demand on the Internet of potentially 1000 exabytes in 2015—that is, a zetabyte!

0.9 Overview of Coming Chapters

This text is divided into five parts. The first gets you started on Python, computing, and problem solving. With that in hand we get down into details in the next part where we develop both the Python language and problem solving skills sufficient to solve interesting problems. The third part provides more tools in the form of Python built-in data structures, algorithm and problem solving development, and functions. Part four shows you how to build classes—your own data structures. The final part includes more on Python.

0.10 Summary

In this chapter we considered ways that data can be represented and manipulated—at the hardware level. In subsequent chapters we will introduce how you, as a programmer, can control the representation and manipulation of data.

25 A day has 86400 secs (24*60*60). Given a number in the range 1 to 86400, output the current time as hours, minutes, and seconds with a 24-hour clock. For example: 70000 sec is 19 hours, 26 minutes, and 40 seconds.

26 A telephone directory has N lines on each page and each page has exactly C columns. An entry in any column has a name with the corresponding telephone number. On which page, column and line is the X^{th} entry (name and number) present? (Assume that page, line, column numbers and X all start from 1.)

27 If the lengths of the two parallel sides of a trapezoid are X meters and Y meters respectively, and the height is H meters, what is the area of the trapezoid? Write Python code to output the area.

28 Simple interest is calculated by the product of the principal, number of years, and interest all divided by 100. Write code to calculate the simple interest on a principal amount of $10000 for a duration of 5 years with the rate of interest equal to 12.5%.

29 The radius and mass of the Earth are $r = 6378 \times 10^3$ meters and $m1 = 5.9742 \times 10^{24}$ kg respectively. Mr. Jones has a mass of X kg. Prompt the user to input X and then calculate the gravitational force (F) and acceleration due to gravity (g) caused by the gravitational force exerted on him by the Earth. Remember, $F = G(m1)(m2)/(r^2)$ and $F = mg$. Let the universal gravitational constant $G = 6.67300 \times 10^{-11}$ (in units of $m^3 kg^{-1} s^{-2}$ assuming the MKS(Meter-Kilogram-Second) system). Check: The resulting value of g should be close to $9.8 m/s^2$.

30 Using modules: Python comes with hundreds of modules. Here is a challenge for you: find a module that you can import that will generate today's date so you can print it. Use your favorite search engine for help in finding which module you need and how to use it. In the end, you task is to do the following:

```
>>> print "Today's date is:", X
Today's date is:  2009-05-22
```

31 Consider a triangle with sides of lengh 3, 7, and 9. The law of cosines states that given three sides of a triangle (a, b, and c) and the angle C between sides a and b: $c^2 = a^2 + b^2 - 2 * a * b * cos(C)$. Write Python code to calculate the three angles in the triangle.

32 In Football there is a statistic for quarterbacks called the Passer Rating. To calculate the passer rating you need five inputs: pass completions, pass attempts, total passing yards, and interceptions. There are five steps in the algorithm. Write a program that asks for the five inputs and then prints the pass rating.

(a) C is the "completions per attempt" times 100 - 30 all divided by 20

(b) Y is the "yards per attempt" - 3 all divided by 4

(c) T is the "touchdowns per attempt" times 20

(d) I is 2.375 minus ("interceptions minus attempts" times 35)

(e) The pass rating is the sum of C, Y, T, and I all divided by 6 and then multiplied by 100

33 Checking the user input for errors is a vital part of programming. The simple program below attempts to take a string input and convert it into an integer. What will happen if the user enters 'Hello World' at the prompt rather than a number? Can you think of a way that the program can be altered to handle this input (hint: think about adjusting how the program handles different types of input)?

```
Raw1 = input ('Please enter a number:  ')
Int1 = int (Raw1)
```

34 BMI. Body Mass Index (BMI) is a number calculated from a person's weight and height. According to the Centers for Disease Control the BMI is a fairly reliable indicator of body fatness for most people. BMI does not measure body fat directly, but research has shown that BMI correlates to direct measures of body fat, such as underwater weighing and dual energy x-ray absorptiometry. The formula for BMI is:

$$weight/height^2$$

where *weight* is in kilograms and *height* is in meters.

(a) Write a program that prompts for metric weight and height and outputs the BMI.

(b) Write a program that prompts for weight in pounds and height in inches, converts the values to metric, and then calculates the BMI.

1.12.1 Programming Projects

1 **The Great Lakes are how big?**
The Great Lakes in the United States contain roughly 22% of the world's fresh surface water (22,810 km^3). It is hard to conceive how much water that is. Write a program to calculate how deep it would be if all the Great Lakes' water was spread evenly across the 48 contiguous U.S. states. You will need to do some Internet research to determine the area of that region.

2 **Where is Voyager 1**?
The Voyager 1 spacecraft, launched September 15, 1977, is the farthest traveling earth-made object. It is presently on the outer edges of our solar system. The NASA update page on November, 7, 2008 reported it at a distance of approximately 10,044,000,000 miles from the Sun, traveling away from the Sun at 38,241 miles/hr.

Your program will prompt the user for an integer number (a number without decimal points) which indicates the number of days after 11/07/08. You will calculate the distance of Voyager from the Sun using the numbers from 11/07/08 (assume velocity is constant) plus the entered number of days and report:

- Distance in miles
- Distance in kilometers (1.609344 kilometers/mile)
- Distance in Astronomical Units (AU, 92,955,887.6 miles/AU)
- Round trip time for radio communication in hours. Radio waves travel at the speed of light, listed at 299,792,458 meters/second.

3 **Oil Conversions and Calculations**.
Your program will prompt the user for a floating point number which stands for gallons of gasoline. You will reprint that value along with other information about gasoline and gasoline usage:

- Number of liters
- Number of barrels of oil required to make this amout of gasoline
- Number of pounds of CO2 produced
- Equivalent energy amount of ethanol gallons
- Price in US dollars

Here are some approximate conversion values:

- 1 barrel of oil produces 19.5 gallons of gasoline.

- 1 gallon of gasoline produces 20 pounds of CO_2 gas when burned.
- 1 gallon of gasoline contains 115,000 BTU (British Thermal Units) of energy.
- 1 gallon of ethanol contains 75,700 BTU of energy.
- 1 gallon of gasoline costs $4.00/gallon.

Look on the Internet for some interesting values for input such as the average number of gallons consumed per person per year, or consumed by the country per day, or consumed per year.

4 **Measurement**
Your program will prompt the user for an floating point value representing miles/hour. You will reprint that value along with the value converted to the following values:

- barleycorns/day
- furlongs/fortnight
- Mach number
- percentage of the speed of light (PSL)

Here are some approximate conversion values:

- 1 meter is 117.647 barleycorns (an old English measure of length).
- 1 furlong is 220 yards.
- 1 fortnight is 2 weeks.
- Mach 1 is the speed of sound in air: 1130 feet/second.
  Mach 1.5 means 1.5 times the speed of sound.
- PSL is the percentage of the speed of light in a vacuum: 299,792,458 meters/second.

PART 2

Starting to Program

CHAPTER 1

Beginnings

> A good workman is known by his tools.
>
> proverb

OUR FIRST STEPS IN PROGRAMMING ARE TO LEARN THE DETAILS, SYNTAX, AND semantics of the Python programming language. This necessarily involves getting into some of the language details, focusing on level 1 (language) issues as opposed to level 2 (problem-solving) issues. Don't worry: we haven't forgotten that the goal is to do effective problem solving, but we have to worry about both aspects, moving between levels as required. A little proficiency with Python will allow us to write effective problem-solving programs.

Here are our first two **RULES** of programming:

> **Rule 1:** Think before you program!
> **Rule 2:** A program is a human-readable essay on problem solving that also happens to execute on a computer.

1.1 PRACTICE, PRACTICE, PRACTICE

Let's start experimenting with Python. Before we get too far along, we want to emphasize something important. One of the best reasons to start learning programming using Python is that you can easily experiment with Python. That is, Python makes it easy to try something out and see the result. Anytime you have a question, simply try it out.

Learning to experiment with a programming language is a very important skill, and one that seems hard for introductory students to pick up. So let's add a new **RULE**.

> **Rule 3:** The best way to improve your programming and problem skills is to practice!

Problem solving—and problem solving using programming to record your solution—requires practice. The title of this section is the answer to the age-old joke:

> **Student:** How do you get to Carnegie Hall?
> **Teacher:** Practice, practice, practice!

Learning to program for the first time is not all that different from learning to kick a soccer ball or play a musical instrument. It is important to learn about fundamentals by reading and talking about them, but the best way to *really* learn them is to practice. We will encourage you throughout the book to type something in and see what happens. If what you type in the first time doesn't work, who cares? Try again; see if you can fix it. If you don't get it right the first time, you will eventually. *Experimenting* is an important skill in problem solving, and this is one place where you can develop it!

We begin with our first QUICKSTART. A QUICKSTART shows the development of a working program followed by a more detailed explanation of the details of that program. A QUICKSTART gets us started, using the principle of "doing" before "explaining." Note that we ask you to try some things in the Python shell as we go along. Do it! Remember **RULE 3**. (To get Python, see Appendix A.)

1.2 QUICKSTART, THE CIRCUMFERENCE PROGRAM

Let's start with a simple task. We want to calculate the circumference and area of a circle given its radius. The relevant mathematical formulas are:

- circumference = $2 * \pi * radius$
- area = $\pi * radius^2$

To create the program, we need to do a couple of things:

1. We need to prompt the user for a radius.
2. We need to apply the mathematical formulas listed previously using the acquired radius to find the circumference and area.
3. We need to print out our results.

Here is that program. Let's name it `circumference.py`. The ".py" is a file *suffix*.

PROGRAMMING TIP

Most computer systems add a suffix to the end of a file to indicate what "kind" of file it is—what kind of information it might store: music (".mp3"), pictures (".jpg"), text (".txt"), etc. Python does the same and expects a Python file to have a ".py" suffix. IDLE, Python's default editor (see Appendix A), is fairly picky about this. If you save your program without the ".py" suffix, you will know it right away, as all the colors in the editor window disappear. Add the suffix, resave, and the colors come back. Those colors are useful in that each color indicates a type of thing (yellow for strings, blue for keywords) in the program, making it more readable.

Code Listing 1.1

```
1 # Calculate the area and circumference of a circle from its radius.
2 # Step 1: Prompt for a radius.
3 # Step 2: Apply the area formula.
4 # Step 3: Print out the results.
5
6 import math
7
8 radius_str = input("Enter the radius of your circle: ")
9 radius_int = int(radius_str)
10
11 circumference = 2 * math.pi * radius_int
12 area = math.pi * (radius_int ** 2)
13
14 print ("The cirumference is:",circumference,  \
15       ", and the area is:",area)
```

Important: The line numbers shown in the program are *not* part of the program. We list them here only for the reader's convenience.

Before we examine the code, let's illustrate how the program runs with two different input radii to confirm that it works properly.

The easiest way to run a program is to open that program in the IDLE editor, then select Run → Run Module (F5). This *imports* the file into the shell and runs it. Note that every time you run the program, the Python shell prints the `"====== RESTART ======"` line, indicating that the shell is restarting and running your program.

You can choose some "obvious" values to see if you get expected results. For example, a radius value of 1 results in the area having the recognizable value of π = 3.14159.... Although not a complete test of the code, it does allow us to identify any gross errors quickly. The other case has a radius of 2 that can be easily checked with a calculator:

```
IDLE 3.2
>>> ================================ RESTART ================================
>>>
Enter the radius of your circle: 1
The cirumference is: 6.283185307179586 , and the area is: 3.141592653589793
>>> ================================ RESTART ================================
>>>
Enter the radius of your circle: 2
The cirumference is: 12.566370614359172 , and the area is: 12.566370614359172
>>>
```

1.2.1 Examining the Code

Let's examine the circumference.py code line by line. In this first example, there are many topics that we will touch on briefly, but they will be explained in more detail in subsequent chapters. Note that we number only every fifth line in the code listing. Here is a walk-through of this code.

Lines 1–4: Anything that follows a pound sign (#) is a comment for the human reader. The Python interpreter ignores it. However, it does provide us as readers some more information on the intent of the program (more on this later). Remember **RULE 2**. Comments help make your document easier to understand for humans.

Line 6: This line imports special Python code from the `math` *module*. A module is a Python file containing programs to solve particular problems; in this case, the `math` module provides support for solving common math problems. Modules are described in more detail in Section 1.4.1. Python has many such modules to make common tasks easier. In this case we are interested in the value π provided by the `math` module, which we indicate in the program using the code `math.pi`. This is a naming convention we will explore in more detail in Section 1.8, but essentially the code `math.pi` means within the module named `math` there is a value named `pi`, with a "." separating the module and value name.

Line 8: This line really has two parts: the Python code to the right of the = sign and the Python code on the left:

- On the right, `input` is a small Python program called a *function*. Functions (see Chapter 8) are often-used, small program utilities that do a particular task. The `input` function prints the characters in quotes "Enter the radius of your circle:" to the Python shell and waits for the user to type a response. Whatever the user types in the shell before pressing the Enter key at the end is *returned*, that is, provided as input to program.
- On the left side of the = is a *variable*. For now, consider a variable to be a name that is associated with a value. In this case, the value returned from `input` will be associated with the name `radius_str` (programmers traditionally shorten "string" to "str").

  Think of the = as a kind of glue, linking the values on the right side with the variable on the left. A line with = is called an *assignment statement*; we will have more to say about assignment in Section 1.5.1.

Line 9: The user's response returned by `input` is stored as a sequence of characters, referred to in computer science as a *string* (see Chapter 4). Python differentiates a sequence of characters, such as those that constitute this sentence, from numbers on which we can perform operations such as addition, subtraction, and so on. Strings are differentiated from numbers by using quotes, and either single or double quotes are acceptable ("hi mom" or 'monty'). For this program we want to work with numbers, not characters, so we must convert the user's response from a string of characters to numbers. The `int` function takes the value associated with the variable `radius_str` and returns the

integer value of `radius_str`. In other words, it converts the string to an integer. As in the previous line, the value returned by *`int`* is associated with a variable using the assignment statement. For example, if `radius_str` holds the string of characters '27', the *`int`* function will convert those characters to the integer 27 and then associate that value the new variable name `radius_int`.

The difference between the characters "27" and the number 27 will likely seem strange at first. It is our first example of value *types*. Types will be described in more detail in Section 1.6, but for now, let's say that a type indicates the kinds of things we can do to a value and the results we obtain.

Line 11: Here we calculate the circumference using the following formula:

```
circumference = 2 * pi * radius
```

While +, -, and / mean what you expect for math operations (addition, subtraction, and division), we use the * symbol to represent multiplication as opposed to · or ×. This convention avoids any confusion between "x" and × or "." and ·. The integer 2, the value associated with the variable `math.pi`, and the value associated with the `radius_int` are multiplied together, and then the result is associated with the variable named `circumference`. As you will later see, the ordering is important: the mathematical expression on the right-hand side of the equal sign is evaluated first, and then the result is associated with the variable on the left-hand side of the equal sign.

Line 12: Similarly, we calculate the area using the formula listed previously. There isn't a way to type in an exponent (superscript) from a keyboard, but Python has an exponentiation operator `**` by which the value on the left is raised to the power on the right. Thus `radius_int ** 2` is the same as `radius_int` squared or $\texttt{radius\_int}^2$. Note that we use parentheses to group the operation. As in normal math, expressions in parentheses are evaluated first, so the expression `math.pi * (radius_int ** 2)` means square the value of `radius_int`, then take that result and multiply it by *`pi`*.

Lines 14 and 15: We print the results using the Python ***`print`*** statement. Like the `input` statement, ***`print`*** is a function that performs a much-used operation—printing values to the Python shell. The print statement can print strings bracketed by quotes (either single or double quotes) and a value associated with a variable. Printable elements are placed in the parentheses after the ***`print`*** statement. If the element being printed is quoted, it is printed exactly as it appears in the quotes; if the element is a variable, then *the value associated with the variable* is printed. Each object (string, variable, value, etc.) that is to be printed is separated from other objects by commas. In this case, the ***`print`*** statement outputs a string, a variable value, another string, and finally a variable value. The backslash character (\) indicates that the statement continues onto the next line—in this case, the two-line ***`print`*** statement behaves as if it were one long line. Stretching a statement across two lines can enhance readability—particularly on a narrow screen or page. See Section 1.4.3.

1.3 AN INTERACTIVE SESSION

An important feature of Python, particularly when learning the language, is that it is an *interpreted* language. By interpreted we mean that there is a program within Python called the interpreter that takes each line of Python code, one line at a time, and executes that code. This feature means that we can try out lines of code one at a time by typing into the Python shell. The ability to experiment with pieces of code in the Python shell is something that really helps while you're learning the language—you can easily try something out and see what happens. That is what is so great about Python—you can easily explore and learn as you go.

Consider our circumference program. The following code shows a session in a Python shell in which a user types each line of the program we listed previously, to see what happens. We also show a few other features, just to experiment. *Open up a Python shell and follow along by trying it yourself.* The comments (text after #, the pound sign) are there to help walk you through the process. There is no need to type them into the Python shell.

```
>>> import math
>>> radius_str = input("Enter the radius of your circle: ")
Enter the radius of your circle: 20
>>> radius_str     # what is the value associated with radius_str
'20'
>>> radius_int = int(radius_str)  # convert the string to an integer
>>> radius_int    # check the value of the integer
20                  # look, no quotes because it is a number
>>> int(radius_str)  # what does int() return without assignment (=)
20
>>> radius_str     # int() does not modify radius_str!!
'20'
>>> math.pi          # let's see what value is associated with math.pi
3.141592653589793
>>> circumference = 2 * math.pi * radius_int  # try our formula
>>> circumference
125.66370614359172
>>> area = math.pi * radius_int ** 2   # area using exponentiation
>>> area
1256.6370614359173
>>> math.pi * radius_int ** 2   # area calculation without assignment
1256.6370614359173

>>> print("Circumference: ", circumference, ", area: ", area)
Circumererence:  125.66370614359172 , area:  1256.6370614359173
>>>
```

Within the Python shell, you can find out the value associated with any variable name by typing its name followed by the Enter key. For example, when we type `radius_str` in the shell as shown in the example, '20' is output. The quotes indicate that it is a string of characters, '20', rather than the integer 20.

Interestingly, Python treats single quotes '20' the same as double quotes "20". You can choose to designate strings with single or double quotes. It's your choice!

We can see that after the conversion of the string to an integer using the `int` function, simply typing `radius_int` results in the integer 20 being printed (no quotes).

We can also try the expression `int(radius_str)` without assigning the result to `radius_int`. Note that `radius_str` is unchanged; it is provided as a value for the `int` function to use in its calculations. We then check the value of `pi` in the `math` module named `math.pi`. Next we try out the circumference and area formulas. The area calculation is next, using the exponentiation operator (`**`), raising `radius_int` to the second power. Finally, we try the **`print`** statement. Remember **RULE 3.**

An advantage of Python over other languages such as C, C++, and Java is the Python shell. Take advantage of this feature, because it can greatly enhance learning what the Python language can do. If you wonder about how something might or might not work, try it in the Python shell!

1.4 PARTS OF A PROGRAM

RULE 2 describes a program as an essay on problem solving that is also executable. A program consists of a set of *instructions* that are executed sequentially, one after the other in the order in which they were typed. We save the instructions together in a *module* for storage on our file system. Later, the module can be *imported* into the Python interpreter, which runs programs by executing the instructions contained in the module.

1.4.1 Modules

- A *module* contains a set of Python commands.
- A module can be stored as a file and *imported* into the Python shell.
- Usage:

```
import module  # load the module
```

Hundreds of modules come with the standard Python distribution—the `math` module and many more can be found and imported. You can even write your own modules and use them as tools in your own programming work!

1.4.2 Statements and Expressions

Python differentiates code into one of two categories: *expressions* or *statements*. The concept of an expression is consistent with the mathematical definition, so it may be familiar to you.

Expression: a combination of values and operations that creates a new value that we call a *return value*—i.e., the value returned by the operation(s). If you enter an expression into

the Python shell, a value will be returned and displayed; that is, the expression `x + 5` will display 7 if the value of `x` is 2. Note that the value associated with `x` is not changed as a result of this operation!

Statement: does not *return a value*, but does perform some task. Some statements may control the flow of the program, and others might ask for resources; statements perform a wide variety of tasks. As a result of their operation, a statement may have a *side effect*. A side effect is some change that results from executing the statement. Take the example of the *assignment statement* `my_int = 5` (shown in the following example). This statement does not have a *return value*, but it does set the value associated with the variable `my_int` to 5, a *side effect*. When we type such an assignment into the Python shell, no value is returned, as you can see here:

```
>>> my_int = 5  # statement, no return value but my_int now has value 5
>>> my_int
5
>>> my_int + 5  # expression, value associated with my_int added to 5
10
>>> my_int      # no side effect of expression, my_int is unchanged
5
>>>
```

However, after we type the assignment statement, if we type the variable `my_int`, we see that it does indeed now have the value of 5 (see the example session). A statement never returns a value, but some statements (not all) may have a side effect. You will see more of this behavior as we explore the many Python statements.

PROGRAMMING TIP

Knowing that an expression has a value, but a statement does not, is useful. For example, you can print the value generated by an expression: ***print***`(x + 5)` (as long as `x` has a value). However, if you try to print a statement, Python generates an error. If no value is returned from a statement, then what will the ***print*** output? Python avoids this by not allowing a statement to be printed. That is, it is not allowable syntax.

```
>>> print(x + 5)          # printing an expression
7
>>> print(y = x + 5)      # trying to print a statement
SyntaxError: invalid syntax
```

There are a number of instances of entering expressions and statements into the Python shell in the previous examples. Whenever an expression was entered, its return value was printed on the following line of the console. Whenever a statement was entered, nothing was printed: if we wanted to see what a statement changed (side effect), we, the programmers, would have had to inquire about the change.

1.4.3 Whitespace

When we type, we usually separate words with what is typically called *whitespace*. Python counts as whitespace the following characters: space, tab, return, linefeed, formfeed, and vertical tab. Python has the following rules about how whitespace is used in a program:

- Whitespace is *ignored* within both expressions and statements.
  For example, `Y=X+5` has exactly the same meaning as `Y = X + 5`.
- *Leading* whitespace, whitespace at the beginning of a line, defines *indentation*. Indentation plays a special role in Python (see the following section).
- Blank lines are also considered to be whitespace, and the rule for blank lines is trivial: blank lines are allowed anywhere and are ignored.

Indentation

Indentation is used by all programmers to make code more readable. Programmers often indent code to indicate that the indented code is *grouped together*, meaning those statements have some common purpose. However, indentation is treated uniquely in Python. Python *requires* it for grouping. When a set of statements or expressions needs to be grouped together, Python does so by a consistent indentation. Grouping of statements will be important when we get to control statements in Chapter 2.

Python requires consistency in whitespace indentation. If previous statements use an indentation of four spaces to group elements, then that must be done consistently throughout the program.

The benefit of indentation is *readability*. While other programming languages encourage indentation, Python's whitespace indentation *forces* readability. The disadvantage is that maintaining consistency with the number of spaces versus tabs can be frustrating, especially when cutting and pasting. Fortunately, Python-aware editors such as IDLE automatically indent and can repair indentation.

PROGRAMMING TIP

IDLE provides commands under the Format menu to help change indentation. If you are having problems with indentation, or if Python is complaining about irregular indentation, use these commands to repair the problem.

Continuation

Long lines of code, those wider than the width of a reasonably sized editing window, can make reading code difficult. Because readability is very important (remember **RULE 2**), Python provides ways to make long lines more readable by splitting them. Such splitting is called a *continuation*. If a single statement runs long, that statement can be continued onto another line (or lines) to help with readability. That is, a continued line is still a

single line, it just shows up over multiple lines in the editor. You indicate a continuation by placing a backslash character (\) at the end of a line. Multiple-line expressions can be continued similarly. In this way, a long line that may be difficult to read can be split in some meaningful and readable way across multiple lines. The first program in this chapter, `circumference.py`, used such a backslash character in the print statement.

1.4.4 Comments

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
>
> Donald Knuth[1]

We will say many times, in different ways, that a program is more than "just some code that does something." A program is a document that describes the thought process of its writer. Messy code implies messy thinking and is both difficult to work with and understand. Code also happens to be something that can run, but just because it can run does not make it a good program. Good programs can be read, just like any other essay. Comments are one important way to improve *readability*. Comments contribute nothing to the running of the program, because Python ignores them. In Python, anything following a pound character (#) is ignored on that line. However, comments are critical to the readability of the program.

There are no universally agreed-upon rules for the right style and number of comments, but there is near-universal agreement that they can enhance readability. Here are some useful guidelines:

- The *why* philosophy: "Good comments don't repeat the code or explain it. They clarify its intent. Comments should explain, at a higher level of abstraction than the code, what you're trying to do." (*Code Complete* by McConnell)
- The *how* philosophy: If your code contains a novel or noteworthy solution, add comments to explain the methodology.

1.4.5 Special Python Elements: Tokens

As when learning any language, there are some details that are good to know before you can fully understand how they are used. Here we show you the special keywords, symbols, and characters that can be used in a Python program. These language elements are known generically as *tokens*. We don't explain in detail what each one does, but it is important to note that Python has special uses for all of them. You will become more familiar with them as we proceed through the book. More important than what they do is the fact that Python

[1] "Literate Programming," *Computer Journal* 27(2), 1984

reserves them for its own use. You can't redefine them to do something else. Be aware they exist so that you don't accidentally try to use one (as a variable, function, or the like).

Keywords

Keywords are special words in Python that cannot be used by you to name things. They indicate commands to the Python interpreter. The complete list is in Table 1.1. We will introduce commands throughout the text, but for now, know that you cannot use these words as names (of variables, of functions, of classes, etc.) in your programs. Python has already taken them for other uses.

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

TABLE 1.1 Python Keywords

Operators

Operators are special tokens (sequences of characters) that have meaning to the Python interpreter. Using them implies some operation, such as addition, subtraction, or something similar. We will introduce operators throughout the text. The complete list is in Table 1.2. You can probably guess many of them, but some of them will not be familiar.

+	-	*	**	/	//	%
<<	>>	&	|	^	~	
<	>	<=	>=	==	!=	<>
+=	-=	*=	/=	//=	%=	
&=	|=	^=	>>=	<<=	**=	

TABLE 1.2 Python Operators

Punctuators and Delimiters

Punctuators, a.k.a. delimiters, separate different elements in Python statements and expressions. Some you will recognize from mathematics; others you will recognize from English. We will introduce them throughout the text. The complete list is in Table 1.3.

(	)	[	]	{	}
,	:	.	`	=	;
'	"	#	\	@	

TABLE 1.3 Python Punctuators

Literals

In computer science, a *literal* is a notation for representing a fixed value—a value that cannot be changed in the program. Almost all programming languages have notations for atomic values, such as integers, floating-point numbers, strings, and Booleans. For example, 123 is a literal; it has a fixed value and cannot be modified. In contrast to literals, variables are symbols that can be assigned a value, and that value can be modified during the execution of the code.

1.4.6 Naming Objects

> The practitioner of . . . programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.
>
> Donald Knuth[2]

If writing a program is like writing an essay, then you might guess that the names you use in the program, such as the names of your variables, would help greatly in making the program more readable. Therefore, it is important to choose names well. Later we will provide you with some procedures to choose readable names, but here we can talk about the rules that Python imposes on name selection.

1. Every name must begin with a letter or the underscore character (_):
   - A numeral is not allowed as the first character.[3]
   - Multiple-word names can be linked together using the underscore character (_)—e.g. `monty_python`, `holy_grail`. A name *starting* with an underscore is often used by Python and Python programmers to denote a variable with special characteristics. You will see these as we go along, but for now it is best to *not* start variable names with an underscore until you understand what that implies.
2. After the first letter, the name may contain any combination of letters, numbers, and underscores:
   - The name cannot be a *keyword* as listed in Table 1.1.
   - You cannot have any delimiters, punctuation, or operators (as listed in Tables 1.2 and 1.3) in a name.

[2] *ibid*

[3] This is so that Python can easily distinguish variable names from numbers.

3. A name can be of any length.
4. UPPERCASE is different from lowercase:
   - `my_name` is different than `my_Name` or `My_Name` or `My_name`.

1.4.7 Recommendations on Naming

Because naming is such an important part of programming, conventions are often developed that describe how to create names. Such conventions provide programmers with a common methodology to make clear what the program is doing to anyone who reads it. These conventions describe how to name various elements of a program based on their function, in particular when to use various techniques to name elements (capitalize, uppercase, leading underscore, etc.). It is a bit beyond us at this point to fully discuss these standards, as we are not yet familiar with the different kinds of elements in a Python program, but we will describe the rules as we go along. However, for those who are interested, we will be using the standard that Google uses for its Python programmers, the Google Style Guide for Python, which is similar to Python's PEP 8, the Style Guide for Python Code. While we follow those guidelines, it is wise to not follow them to the point where readability is impinged. In that spirit we pull our next rule straight from PEP 8:

> **Rule 4:** A foolish consistency is the hobgoblin of little minds.

That rule is actually a quote from Ralph Waldo Emerson: "A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines. With consistency a great soul has simply nothing to do." As Emerson says, our rules are useful to follow, but sometimes not.

1.5 VARIABLES

A *variable* is a name you create in your program to represent "something" in your program. That "something" can be one of many types of entities: a value, another program to run, a set of data, a file. For starters, we will talk about a variable as something that represents a value: an integer, a string, a floating-point number, and so on. We create variables with meaningful names, because the purpose of a good variable name is to make your code more readable. The variable name *pi* is a name most would recognize, and it is easier to both read and write than 3.1415926536. Once a variable is created, you can store, retrieve, or modify the data associated with that variable. Every time you use your variable in a program, its value is retrieved by Python and used in that variable's place. Thus, a variable is really a way to make it easier to read the program you are writing.

[4] http://google-styleguide.googlecode.com/svn/trunk/pyguide.html.

PROGRAMMING TIP

It is often useful to describe your variable using a multiword phrase. The recommended way to do this, according to the Google Style Guide, is called "lower_with_under". That is, use lowercase to write your variable names and connect the words together using an underline. Again, it is useful to avoid using a leading underline with your variable names for reasons that will become clear later. It is also useful to avoid capital letters, as the style guide will have something to say about when to use those later. Thus "radius_int" or "circumference_str" are good variable names.

How does Python associate the value of the variable with its name? In essence, Python makes a list of names (variables, but other names as well) that are being used right now in the Python interpreter. The interpreter maintains a special structure called a *namespace* to keep this list of names and their associated values (see Figure 1.1). Each name in that list is associated with a value, and the Python interpreter updates both names and values during the course of its operation. The name associated with a value is an *alias* for the value, i.e., another name for it. Whenever a new variable is created, its name is placed in the list along with an association to a value. If a variable name already exists in the table, its association is updated.

1.5.1 Variable Creation and Assignment

How is a name created? In Python, when a name is first used (assigned a value, defined as a function name, etc.) is when the name is created. Python updates the namespace with the new name and its associated value.

Assignment is one way to create a name for a variable. The purpose of assignment is to associate a name with a value. The symbol of assignment for Python is the equal sign (=).

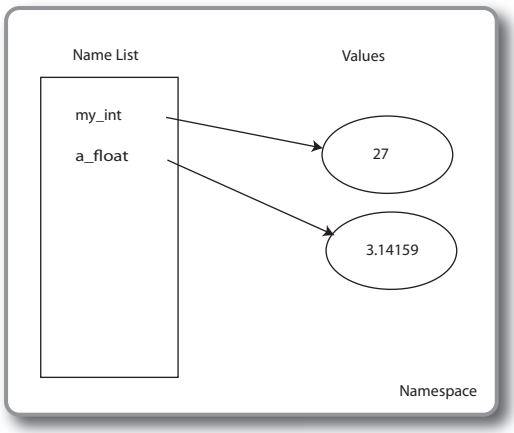


FIGURE 1.1 Namespace containing variable names and associated values.

A simple example follows:

```
my_int = math.pi + 5
```

In this case, 5 is added to the value of variable associated with `math.pi` and the result of that expression is associated with the variable `my_int` in the Python namespace. It is important to note that the value `math.pi` is **not** modified by this operation. In fact, assignment does not change any values on the right-hand side. Only a new association is created by assignment—the association with the name on the left-hand side.

The general form of the assignment statement is the same as in mathematics:

left-hand side = right-hand side
LHS = RHS

Although we use the familiar equal sign (=) from mathematics, its meaning in programming languages is different! In math, the equal (=) indicates equality: what is on the left-hand side of the equal sign (=) has the same value as what is on the right-hand side. In Python, the equal sign (=) represents *assignment*. Assignment is the operation to associate a value with a variable. The left-hand side represents the variable name and the right-hand side represents the value. If the variable does not yet exist, then the variable is created and placed in the namespace otherwise, the variable's value is updated to the value on the right-hand side. This notation can lead to some odd expressions that would not make much sense mathematically but make good sense from Python's point of view, such as:

```
my_var = my_var + 1
```

We interpret the previous statement as follows: get the value referred to by `my_var`, add 1 to it, and then associate the resulting sum with the variable `my_var`. That is, if `my_var`'s value was 4, then after assignment, its value will be updated to be 5. More generally, the process is to evaluate everything in the expression on the right-hand side first, get the value of that expression, and associate that value with the name on the left-hand side.

Evaluation of an assignment statement is a two-step process:

1. Evaluate the expression on the right-hand side.
2. Take the resulting value from the right-hand expression and associate it with the variable on the left-hand side (create the variable if it doesn't exist; otherwise update it).

For example:

```
my_var = 2 + 3 * 5
```

First evaluate the expression on the right-hand side, resulting in the value 17 (remember, multiplication before addition), then associate 17 with `my_var`, that is, update the namespace so that `my_var` is an alias for 17 in the namespace. Notice how Python uses the standard mathematical rules for the order of operations: multiplication and division before addition or subtraction.

In the earlier assignment statement

```
my_var = my_var + 1
```

notice that `my_var` has two roles. On the right-hand side, it represents a value: "get this value." On the left-hand side, it represents a name that we will associate with value: "a name we will associate with the value."

With that in mind, we can examine some assignment statements that do not make sense and are thus not allowed in Python:

- `7 = my_var + 1` is illegal because 7, like any other integer, is a literal, not a legal variable name, and cannot be changed (you wouldn't want the value 7 to somehow become 125).
- `my_var + 7 = 14` is illegal because `my_var + 7` is an expression, not a legal variable name in the namespace.
- Assignment cannot be used in a statement or expression where a value is expected. This is because assignment is a statement; it does not return a value. The statement `print(my_var = 7)` is illegal, because `print` requires a value to print, but the assignment statement does not return a value.

Check Yourself: Variables and Assignment

1. Which of the following are acceptable variable names for Python?
   (a) xyzzy
   (b) 2ndVar
   (c) rich&bill
   (d) long_name
   (e) good2go
2. Which of the following statements best describes a Python namespace?
   (a) A list of acceptable names to use with Python
   (b) A place where objects are stored in Python
   (c) A list of Python names and the values with which they are associated
   (d) All of the above
   (e) None of the above
3. Give the values printed by the following program for each of the labeled lines.

```
int_a = 27
int_b = 5
int_a = 6

print(int_a)      # Line 1
print(int_b + 5)  # Line 2
print(int_b)      # Line 3
```

   (a) What is printed by `Line 1`?
   (b) What is printed by `Line 2`?
   (c) What is printed by `Line 3`?

1.6 OBJECTS AND TYPES

In assignment, we associate a value with a variable. What exactly is that value? What information is important and useful to know about that value?

In Python, every "thing" in the system is considered to be an *object*. In Python, though, the word *object* has a very particular meaning. An object in Python has:

- An *identity*
- Some *attributes*
- Zero or more names

Whenever an object is created by Python, it receives an identification number. If you are ever interested in the number of any object, you can use the `id` function to discover its ID number. In general, the ID number isn't very interesting to us, but that number is used by Python to distinguish one object from another. We will take a brief look at the ID here because it helps explain how Python manages objects.

Notice that in addition to the ID number, an object can also have a name, or even multiple names. This name is not part of the object's ID but is used by us, the programmers, to make the code more readable. Python uses the namespace to associate a name (such as a variable name) with an object. Interestingly, multiple namespaces may associate different names with the same object!

Finally, every object has a set of attributes associated with it. Attributes are essentially information about the object. We will offer more insight into object attributes later, but the one that we are most interested in right now is an object's type.

In Python, and in many other languages, each object is considered an example of a *type*. For example, 1, 27, and 365 are objects, each of the same type, called `int` (integer). Also, 3.1415926, 6.022141×10^{23}, and 6.67428×10^{-11} are all objects that are examples of the type called floating-point numbers (real numbers), which is the type called `float` in Python. Finally, "spam", 'ham', and "'fred"' are objects of the type named strings, called `str` in Python (more in Chapter 4).

Knowing the type of an object informs Python (and us, the programmers) of two things:

- *Attributes* of the object tell us something about its "content." For example, there are no decimal points in an integer object, and there are no letters in either an integer object or a float object.
- *Operations* we can perform on the object and the results they return. For example, we can divide two integer objects or two float objects, but the division operation makes no sense on a string object.

If you are unsure of the type of an object in Python, you can ask Python to tell you its type. The function `type` returns the type of any object. The following session shows some interaction with Python and its objects:

```
>>> a_int = 7
>>> id(a_int)
16790848
>>> type(a_int)      # because 7 is an integer, a_nt is of type int
```

```
<class 'int'>
>>> b_float = 2.5
>>> id(b_float)      # note that b_float's ID is different than a_int's ID
17397652
>>> type(b_float)    # because 2.5 is real, b_float is of type float
<class 'float'>
>>> a_int = b_float # associate a_int with value of b_float
>>> type(a_int)
<class 'float'>      # a_int is now of type float
>>> id(a_int)       # a_int's now has the same ID as b_float
17397652
```

Notice a few interesting things in the example session:

- When we ask for the ID (using `id`), we get a system-dependent ID number. That number will be different from session to session or machine to machine. To us, the ID number is hard to remember (though not for Python). That is the point! We create variable names so that we can remember an object we have created so we don't *have* to remember weird ID numbers. We show the ID here to help explain Figure 1.2.
- The information returned by a call to the `type` function indicates the type of the value. An integer returns `<class 'int'>`, a floating-point object returns `<class 'float'>`, etc.
- Most important, the type of an object has *nothing* to do with the variable name; instead it specifies only the *object* with which it is associated. In the previous Python session, an `int` object (`a_int = 7`) and a floating-point object (`b_float = 2.5`) associated (respectively) with `a_int` and `b_float`. We then assign the floating-point object to the variable named `a_int` using `a_int = b_float`. The type of object associated with `a_int` is now float. The name of the variable has nothing to do with the type of object with which it is associated. However, it can be useful if the name *does* say something about the associated object for those who must read the code. In any case, Python does not enforce this naming convention. We, as good programmers, must enforce it. Note that the ID of the object associated with `a_int` is now the same as the ID of the variable `b_float`. Both names are now associated with the same object.

Figure 1.2 shows the namespace with two assignments on the left side and a third assignment on the right.

Python (and therefore we) must pay particular attention to an object's type, because that type determines what is "reasonable" or "permissible" to do to that object, and if the operation can be done, what results. As you progress in Python, you will learn more about predefined types in Python, as well as how we can define our own types (or, said in a different way, our own *class*—see Chapter 11).

The following sections describe a few of the basic types in Python.

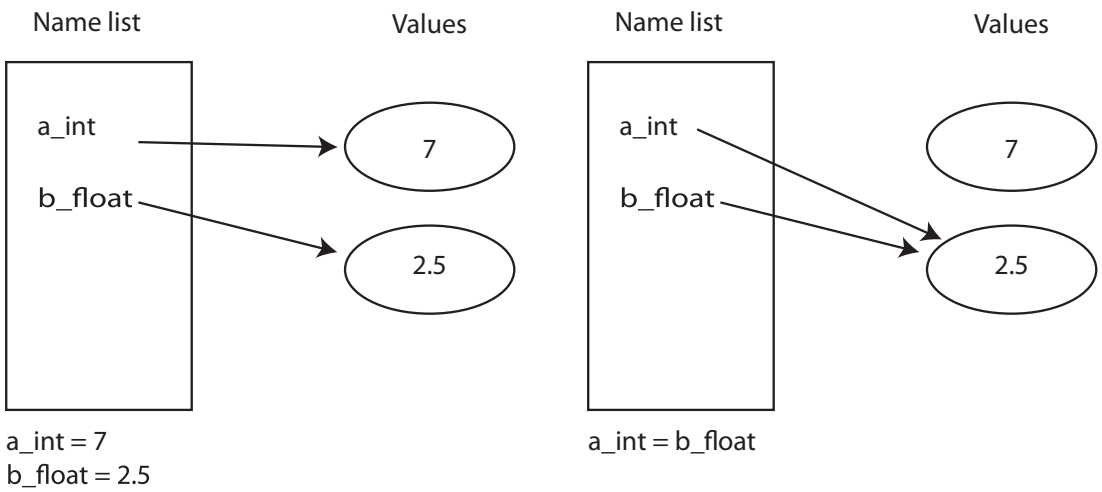


FIGURE 1.2 Namespace before and after the final assignment.

1.6.1 Numbers

Python provides several numeric types. We will work a lot with these types during these early chapters, because they relate to numeric concepts that we are familiar with. You will add more types as we move through the book.

Integers

The integer type is designated in Python as type `int`. The integer type corresponds to our mathematical notion of integer. The operations we can perform are those that we would expect: + (addition), - (subtraction), * (multiplication), and / (division, though there are some complications about division—see Section 1.7), as well as a few others. In Python, integers can grow to be as large as needed to store a value, and that value will be exact, but really big integers can be slower to operate on.

How big an integer can you make? Give it a try.

Integers can be written in normal base 10 form or in other base formats, in particular base 8 (called *octal*) and base 16 (called *hexadecimal*). We note this because of an oddity you might run into with integers: leading zeros are not allowed. Python assumes that, if you precede a number with a 0 (zero), you mean to encode it in a base other than 10. A letter following the 0 indicates the base. If it is "o," base 8 (octal) is specified. An "x" specifies base 16 (hexadecimal) and a "b" specifies base 2 (binary). We illustrate this in a session. The value printed in the shell is the decimal equivalent.

```
>>> 012      # leading zero without letter is invalid
SyntaxError: invalid token
```

```
>>> 0o12    # "o" indicates octal, base 8
10
>>> 0x12    # "x" indicates hexadecimal, base 16
18
>>> 0b101   # "b" indicates binary, base 2
5
```

Floating-Point Numbers

Floating-point or real numbers are designated in Python as type `float`. The floating-point type refers to noninteger numbers—numbers with decimal points. Floats are created either by typing the value, such as 25.678, or by using exponential notation, with the exponent represented by an "e." Thus, 2.99×10^8 is written as `2.99e8` and 9.109×10^{-31} can be written as `9.109e-31`. The operators are, like integers, `+`, `−`, `*`, and / (see Section 1.7 for more detail). Floats represent Real numbers, but only approximately. For example, what is the exact decimal representation of the operation 2.0/3.0? As you know, there *is* no exact decimal equivalent as the result is an infinite series: 2.0/3.0 = 0.666 . . .. Because a computer has a finite amount of memory, real numbers must be represented with approximations of their actual value. Look at the following session.

```
>>> 2.0 / 3.0
0.6666666666666666
>>> 1.1 + 2.2
3.3000000000000003
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
>>>
```

If you were to do the calculations yourself on paper, you would find that 1.1+2.2 is equal to 3.3. However, the session shows it is 3.3000000000000003. Same for the last addition. The result should be zero but Python returns a *very* small value instead. Approximations like this, if carried through multiple evaluations, can lead to significant differences than what is expected. Python does provide a module called the `decimal` module that provides more predictable, and controllable, results for floating-point numbers.

It is important to remember that floating-point values are approximate values, not exact, and that operations using floating-point values yield approximate values. Integers are exact, and operations on integers yield exact values.

Finally, unlike integers in Python, a leading 0 on a floating-point number carries no significance: 012. (notice the decimal point) is equivalent to 12.0.

Fractions

Python also provides the type `Fraction` for rational numbers. A fraction consists of the obvious two parts: the numerator and the denominator. Fractions do not suffer from the

conversion of a rational to a real number, as discussed previously, and can be operated on without loss of precision using addition, subtraction, multiplication, and division. See the `fractions` module for more information.

1.6.2 Other Built-In Types

Python has more types that we will introduce in the coming chapters. We mention them briefly here as a preview.

Boolean

A Boolean value has a Python type *bool*. The Boolean type refers to the values `True` or `False` (note the capitalization). If an object is of type Boolean, it can be only one of those two values. In fact, the two Boolean objects are represented as integers: 0 is `False` and 1 is `True`. There are a number of Boolean operators, which we will examine in Chapter 2.

String

A string in Python has the type *str*. A string is our first *collection* type. A collection type contains multiple objects organized as a single object type. The string type is a *sequence*. It consists of a collection of characters in a sequence (order matters), delimited by single quotes (‘ ’) or double quotes (“ ”). For example, “This is a string!” or ‘here is another string’ or even a very short string as “x”. Some languages, such as C and its derivatives, consider single characters and strings as different types, but Python only has strings. Operations on strings are described in Chapter 4.

List

A list in Python is of type *list*. A list is also a sequence type, like a string, though it can have elements other than characters in the sequence. Because sequences are collections, a list is also a collection. Lists are indicated with square brackets ([and]), and their contents are separated by commas. Lists will be covered in Chapter 7. Here is a list:

[4, 3.57, ‘abc’]

Dictionary

A dictionary in Python is of type *dict*. A dictionary is a *map* type, a collection though not a sequence. A map type consists of a set of element pairs. The first element in the pair is the *key* and the second is the *value*. The key can be used to search for a value, much like a dictionary or phone book. If you want to look up the phone number (the value) or a person (the key), you can efficiently search for the name and find the number. Curly braces, ({ and }) indicate dictionaries; a colon separates the key and value pair. Dictionaries will be covered in Chapter 9. Here is a dictionary:

```
{'Jones':3471124, 'Larson':3472289, 'Smith':3471288}
```

Set

A set in Python is of type *set*. A set is a collection of unique elements—similar to a set in mathematics. Sets, like dictionaries, use curly braces, but unlike dictionaries there are no colons. A set supports mathematical set operations such as *union* and *intersection*. Sets will be covered in Chapter 9. Here is a set:

$$\{1,3,5\}$$

1.6.3 Object Types: Not Variable Types

As noted earlier, every object has a type in Python. Variables can freely change their object association during a program. A Python variable can refer to any object, and that object, and potentially its type, can change over time. For example, a variable could be associated with a string one moment and be reassigned to an integer later on. Consider this Python session:

```
>>> my_var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'my_var' is not defined
>>> my_var = 7              # create my_var as type int (integer)
>>> my_var
7
>>> my_var = 7.77           # now associate my_var with a float (floating point)
>>> my_var
7.77
>>> my_var = True           # now type is bool (Boolean)
>>> my_var
True
>>> my_var = "7.77"         # now type is str (string)
>>> my_var
'7.77'
>>> my_var = float(my_var)  # convert string my_var to float (floating point)
>>> my_var
7.77
>>>
```

Initially, `my_var` is undefined, as it has never had a value assigned to it (has no association with a value yet), so the Python interpreter complains when you ask for its associated object. Remember that to create a variable name in the namespace, we must assign (or otherwise define) the variable. Subsequently in the example, we assign an integer value (type `int`) to `my_var`, the integer 7. Next, we assign a floating point value, 7.77, to `my_var`, so it is now associated with an object of type `float`. Next we assign a Boolean

value to my_var, so it becomes associated with a type bool (Boolean) value. Finally, we assign a string to my_var (type *str*). For illustration purposes, the characters we chose for the string are the same ones used for the floating-point example. However, note the quotes! Because we put the characters in quotes, the Python interpreter considers the characters to be a string (a sequence of printable characters) rather than a number. To further illustrate that point, we then used the *float* function to convert the string value in my_var to a floating-point object and then assigned that floating-point object value back to my_var. Python's *float* could create the new floating-point object from the string object, because the string consisted of only numbers and a decimal point. If we had tried to convert a string of letters, *float* would have generated an error: *float*("*fred*") is an error.

Because the computer has separate hardware for integers and floating-point arithmetic, the Python interpreter keeps careful track of types for use in expressions. Also, operators may perform different operations depending on the type of the operands. For example, with the plus (+) operator, if the operands are of type *int* or *float*, addition will be performed, but if the operands are of type *str* (that is, a string), concatenation will be performed. Here is an example showing the two interpretations of plus (+):

```
>>> my_var = 2
>>> my_var + my_var
4
>>> my_var = "Ni"
>>> my_var + my_var
'NiNi'
```

PROGRAMMING TIP

Because the type information is not part of the variable, it is often helpful to help keep track of types by affixing the type name to their variables, such as my_int,b_float, phone_book_dict. This style of naming is sometimes called "Hungarian notation." Invented by Charles Simonyi, a programmer who worked at Xerox PARC, Hungarian notation puts the type of the value associated with the variable in front of the name (much as Hungarian puts the family name before the surname). We use a variation that places the type as a suffix.

1.6.4 Constructing New Values

We noted earlier that every object has a type and that the type determines both what is stored in the object and the kinds of operations that can be performed on that object.

There are also some operations associated with the type itself. One of the most useful of those operations is the *constructor*. A constructor is a special operation that is used to *make* a particular object of that type. For example, the operation *int* will create a new object that is of type integer.

Each type has an associated constructor: the constructor's name is the name of the type (`int`, `float`, `str`, ...). If no value is provided within the parentheses, then a default value of that type is provided (for `int` it is 0; for `float` it is 0.0, for `str` it is ''). If a value *is* provided in the parentheses, then the constructor will *convert* that value to a new object of the specified type if it can. Thus, constructors can be used to convert one object to a new object of a different type. You have already seen conversion in examples:

- `int(my_var)` returns an integer representation of the object associated with `my_var`.
  - Note `my_var` itself is unaffected. The object associated with `my_var` is provided to the function `int`, a new object is created, the `my_var` object is converted to an `int`, and that new, converted object is returned.
  - If the object associated with `my_var` cannot be converted to an integer (say, the value " "), then an error will occur.
- `float(my_var)` returns a floating-point representation of the object associated with `my_var`. As with `int`, if the object cannot be converted to a floating-point number, then an error will occur.
- `str(my_var)` returns a string representation of the object associated with `my_var`.

It is again important to note that 1 and "1" are very different.

- 1 is the integer 1.
- "1" is the character digit that gets typed on a keyboard.

Next is an example of a Python session that illustrates using constructors for conversion. The comments explain what is happening:

```
>>> int("1")            # convert string to integer
1
>>> int(1.7)            # convert floating point to integer; note the truncation
1
>>> int(True)           # convert Boolean to integer
1
>>> int('1.1')          # can't go from a string to a float, to an int. Too far!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.1'
>>> float("3.33")       # convert string to floating point
3.33
>>> float(42)           # convert integer to floating point; note the .0
42.0
>>> str(9.99)           # convert floating point to string; note the quotes
'9.99'
>>> my_var = "2"        # create a string
>>> my_var
'2'
>>> int(my_var)         # show that conversion does not change my_var
```

```
2
>>> my_var
'2'
>>> my_var = int(my_var)  # need assignment to change my_var
>>> my_var
2
>>>
```

Note what happens when you convert a floating-point value like 1.7 into an integer. No error occurs: the decimal portion is simply removed and the integer portion is used as the value. When an integer is converted to a float, a decimal portion is added (that is, .0 is appended).

1.7 OPERATORS

As we noted in Section 1.6, for every type there is a set of operations that can be performed on that type. Given the limited number of symbols available, some symbols are used for different purposes for different types. This is called *operator overloading*, a term that indicates that a symbol might have multiple meanings depending on the types of the values. For example, we saw earlier that the plus sign (+) performs different operations depending on whether the operands are integers or strings.

1.7.1 Integer Operators

Most of the operators for integers work exactly as you learned in elementary arithmetic class. The plus sign (+) is used for addition, the minus (−) is for subtraction, and the asterisk (*) is for multiplication.

Division works as you would expect as well, but it is a little different from the other operators. In particular, the integers are not closed with respect to division. By closure, we mean that if you add two integers, the sum is an integer. The same is true for subtraction and multiplication. However, if you divide two integers, the result is not necessarily an integer. It could be an integer (say, 4/2) or it could be a float (4/3). Mathematically it is a rational number (a.k.a. fraction), but Python represents it as a float. For consistency's sake, then, *whenever* you do division, regardless of the types of the operands, the type yielded is a `float`. The next session shows this behavior.

```
>>> a_int=4
>>> b_int=2
>>> a_int + b_int
6
>>> a_int * b_int
8
>>> a_int - b_int
```

```
2
>>> a_int / b_int              # note, 2.0 not 2. Result is always float.
2.0
>>> result = a_int / b_int
>>> type(result)               # checking the type, yes it's a float
<class 'float'>
>>> a_int / 3                  # important that it be a float here
1.3333333333333333
>>>
```

However, perhaps you indeed only want an integer result. Python provides an operator that, when used with integers, provides only the integer part of a division, the quotient (what type is an interesting question, but keep reading). That operator is the // operator. Let us see how it works.

Consider how you did division in elementary school. For example, consider 5 divided by 3, as shown in Figure 1.3.

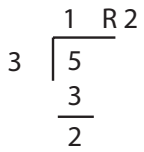


FIGURE 1.3 Long division example.

The *quotient* is 1 and the *remainder* is 2. Observe that both the quotient and remainder are integers, so the integers are closed under the quotient and remainder operations. Therefore, in Python (and many other programming languages), there are separate operators for quotient, //, and remainder, %.[5] We can see these two operators in action by showing the same 5 divided by 3 equation from Figure 1.3 expressed with the two expressions for quotient and remainder:

```
>>> 5 // 3          # integer quotient
1
>>> 5 % 3          # integer remainder
2
>>> 5.0 // 3.0     # integer quotient, but as a float
1.0
>>  5.0 % 3.0      # integer remainder, but as a float
2.0
>> 5.0 / 3.0       # regular division
1.6666666666666667
>> 5 // 3.0        # when types are mixed, result is a float
1.0
```

[5] The remainder operation is known in mathematics as the *modulo* operation.

It is interesting to note the type of the results, though not the value, depends on the types of the operands. If the operation has integer operands, e.g., `5 //3`, the quotient is 1, an *`int`*. If the operation has floating-point operands, e.g., `5.0 //3.0`, the result is 1.0. This is the correct quotient, but its type is a *`float`*. When the types are mixed, e.g., `5 // 3.0`, the type is also a *`float`*. See Section 1.7.3 for more details as to why.

In that session it appears that the `//` operation works the same for both integers and floats, but there is a subtle difference. If the floating-point values have no fractional part, e.g., 2.0, the operation is the same. However, fractional parts indicate that the floating-point operation is actually floating-point division followed by truncation.

Therefore, the full set of operators for integers is:

+	addition
-	subtraction
*	multiplication
/	division
//	quotient
%	remainder
**	exponentiation

1.7.2 Floating-Point Operators

Floating-point operators work as expected. Note that you may use both the quotient and remainder operators as well on floating-point numbers.

+	addition
-	subtraction
*	multiplication
/	division
//	quotient
%	remainder
**	exponentiation

As with integers, addition, subtraction, multiplication, and division perform as expected. As shown in the previous session, quotient and remainder work as well, giving the proper value but as a type *`float`* space then.

1.7.3 Mixed Operations

What is the difference between the numbers 42 and 42.0?[6] The answer is that they are different types. Same value, but different types! Therefore, when you perform operations with them, you might get different answers.

[6] The answer to Life, the Universe, and Everything is 42, according to Douglas Adams's *The Hitchhiker's Guide to the Galaxy.*

What happens when you mix types? For example, how does the computer handle dividing an integer by a floating-point number? The answer is, "it depends." In general, operations provided by each type dictate the rules of what can and cannot be mixed. For numbers, a computer has separate hardware for integer and floating-point arithmetic. Because of that hardware, the operation is best done in one type or the other, so the numbers must be of the same type. Those that are not of the same type must be converted. Given that, what is the correct conversion to perform: integer-to-float or float-to-integer?

Clearly no information is lost when converting an integer to a float, but conversion of a float to an integer would lose the fractional information in the float. Therefore, most programming languages, including Python, when presented with mixed types will "promote" an integer to be a floating point so that both operands are floats and the operation can be performed as floats:

```
>>> var_int = 42
>>> var_float = 42.0
>>> var_int * 5      # multiplication, int times int yields int
210
>>> var_int * 5.0    # multiplication, int times float yields float
210.0
>>> var_float + 5    # addition, float plus int yields float
47.0
>>> var_int / 7      # division, int divide int yields float
6.0                  # division always yields a float!
>>>
```

PROGRAMMING TIP

Normal division in Python always yields a float, even when working with only integer operands.

1.7.4 Order of Operations and Parentheses

The order of arithmetic operations in Python and most common programming languages is the same as the one you learned in arithmetic: multiplication and division before addition or subtraction. The term used to describe the ordering is *precedence*. In this case, we say that multiplication and division have greater precedence than addition or subtraction, so they are done first. Further, exponents have greater precedence than multiplication and division, also as in arithmetic. If operations have the same precedence—that is, multiplication and division—they are done left to right. Finally, as in arithmetic, parentheses can be used to override the precedence order and force some operations to be done before others, regardless of precedence. The precedence of arithmetic operations is shown in Table 1.4; the order in the table is from highest precedence (performed first) to lowest.

Operator	Description
`()`	parentheses (grouping)
`**`	exponentiation
`+x, -x`	positive, negative
`*,/,%,//`	multiplication, division, remainder, quotient
`+,-`	addition, subtraction

TABLE 1.4 Precedence (order) of arithmetic operations: highest to lowest.

Here is a session to illustrate precedence and parentheses:

```
>>> 2 + 3 - 4       # same precedence: left to right
1
>>> 4 / 2 * 5       # same precedence: left to right
10
>>> 2 + 3 * 5       # multiplication before addition
17
>>> (2 + 3) * 5     # parentheses force addition before multiplication
25
>>> 2 + 3 * 5**2    # exponents before multiplication before addition
77
>>> 2 + 3 * 5**2 - 1
76
>>> -4 + 2          # negation before addition and subtraction
-2
```

In Chapter 2 you will learn about Boolean operators and their precedence with respect to the arithmetic operators. The full table of all Python operator precedence is in Appendix E.

1.7.5 Augmented Assignment Operators: A Shortcut!

Operations—especially groups of operations—that are used repeatedly are often provided with a *shortcut*, a simpler way of typing. Python, like other languages, has such shortcuts. They are not required, but they do make typing easier and, at some point, make the code shorter and easier to read.

Our first shortcut is the combination of an integer operation and the assignment sign (=). Though you might not have noticed, they were listed in Table 1.2. They are a combination of one of the arithmetic operators, such as `+, -, *, /`, with the assignment sign `=`. Some examples would be `+=, -=, /=, *=`. Note that the operation comes before the assignment sign, not after!

What does `+=` mean? Let's look at an example: `my_int += 2`. The augmented operator, in general, means "Perform the augmented operation (plus here) using the two operands, the value 2 (value on the right side) and `my_int` (variable on the left side), and reassign the result to `my_int` (variable on the left side)". That is, the following two expressions are exactly equivalent: `my_int += 2` and `my_int = my_int + 2`. The first is a shortcut,

Shortcut	Equivalence
`my_int += 2`	`my_int = my_int + 2`
`my_int -= 2`	`my_int = my_int - 2`
`my_int /= 2`	`my_int = my_int / 2`
`my_int *= 2`	`my_int = my_int * 2`

TABLE 1.5 Augmented Assignment

the second the "long" way. Table 1.5 lists the most commonly used shortcuts. However, incrementing `my_int += 1` is by far the most commonly used shortcut.

Check Yourself: Types and Operators

1. Give the values printed by the following program for each of the labeled lines, and answer the associated questions.

```
a_float = 2.5
a_int = 7
b_int = 6

print(a_int / b_int)      # Line 1
print(a_int // a_float)   # Line 2
print(a_int %  b_int)     # Line 3
print(int(a_float))       # Line 4
print(float(a_int))       # Line 5
```

   (a) `Line 1`: What is printed? What is its type?
   (b) `Line 2`: What is printed? What is its type?
   (c) `Line 3`: What is printed? What is its type?
   (d) `Line 4`: What is printed? What is its type?
   (e) `Line 5`: What is printed? What is its type?

2. Give the values printed by the following program for each of the labeled lines.

```
a_int = 10
b_int = 3
c_int = 2

print(a_int + b_int * c_int)      # Line 1
print( (a_int + b_int) * c_int )  # Line 2
print(b_int ** c_int)             # Line 3
print(0o10 + c_int)               # Line 4
```

   (a) What is printed by `Line 1`?
   (b) What is printed by `Line 2`?
   (c) What is printed by `Line 3`?
   (d) What is printed by `Line 4`?

PROGRAMMING TIP

If you are going to use an augmented assignment, it is important to note that the variable be defined already, meaning that at some point you have assigned it a value. That is, you cannot add 2 to `my_int` if `my_int` doesn't already have a value. Otherwise, you will get an error that `my_int` is undefined.

1.8 YOUR FIRST MODULE, MATH

Python has many strengths, but one of its best is the availability of the many *modules* for various tasks. If you take a look at the Python Package Index (`http://pypi.python.org/pypi`) you will see that there are hundreds[7] of packages provided as modules for you to use in Python—all for free! The Python community, part of the open source community, created those modules for everyone to use. As we described earlier, a module is a collection of instructions saved together as a whole. Those instructions can be loaded into our program using the ***import*** command. We did just that—we used the ***import*** command to import the math module in the program to determine the circumference of a circle in Section 1.2.1. Let's take a deeper look at the math module.

When we import that math module, all the code contained in the math module is made available to our program. The contents of the math module can be found in a couple of ways. The best way is to look online in the Python documentation. Go to the Python docs and take a look: `http://docs.python.org/library/math.html`. You could also import the math module in IDLE, and do the following:

- Type ***import*** `math`
- Type `math.<TAB>`

When you type the tab character after "math.", a list will be created of all the code that was brought in during the import. Alternatively, you could type `help(math)`.

What is typically brought in is either a function (a piece of code we can call to perform some operation) or a variable. Either way, the *name* of the imported code is always preceded with `math.a_name`, meaning "in the math module, the item a_name." Thus, in Section 1.2.1, we used the variable `math.pi`. The `math.` indicates the module name and the name after the "." is one of its elements, in this case the variable `pi`. more simply as "pi in the math module." All the code imported from the math module is referenced in this way. Although there are other ways to import a module, this is the preferred way.

Functions will be properly introduced in Chapter 8. A function has two parts: the *name* of the function, which hopefully indicates what operation it performs, and an *argument* list, which is a parenthetical list of values to be sent into the function as part of its operation.

[7] There are 16,000 for Python 2.7, so we can expect many more.

When the function completes its operation, a *return value* is returned from the function. A typical function is the `math.sin(a_float)` function. The name of the function is `math.sin` and the argument list follows: a parenthetical list of one value, which must be a floating-point number. When `math.sin(a_float)` completes, the sine of `a_float` (which would also be a float) is returned. The returned value can be captured with an assignment, printed, or otherwise used elsewhere in the program.

Some useful functions in the math module include:

`math.sin` Takes one numeric argument and returns the sine of that argument as a float.

`math.cos` Takes one numeric argument and returns the cosine of that argument as a float.

`math.pow` Takes two numeric arguments, x and y, and returns x^y (x raised to the power y) as a float.

`math.sqrt` Takes one numeric argument and returns the square root of that argument as a float.

`math.hypot` Takes two numeric arguments, x and y, and returns $\sqrt{x^2 + y^2}$, the Euclidean distance.

`math.fabs` Takes one numeric argument and returns the absolute value of the argument.

At any time, you can type "help(object)" and Python will provide help on that object. For example:

```
>>> import math
>>> help(math.pow)

Help on built-in function pow in module math:

pow(...)
    pow(x,y)

    Return x**y (x to the power of y).
(END)
```

1.9 DEVELOPING AN ALGORITHM

You know a lot more than when we started this chapter about how to write programs, especially with numbers. Let's take a look at solving a simple problem to see how well you can do. Here's a question:

> How many gallons of water fall on an acre of land given the number of inches of rain that fell? For example, how many gallons for 1 inch, 2 inches, etc.?

This problem really has to do with the weather and how it is reported. When the weather person on television tells you that "1 inch of rain" has fallen, what does that really mean? In case you didn't know, "1 inch of rain" means pretty much what it says. Rain has accumulated to a depth of 1 inch over a certain area.

How do we attack this problem? Our goal is to develop an *algorithm* from which we can derive our program. What is an algorithm? It is one of those common words that can be hard to define precisely. One definition that works for what we need to do in this book is:

> **algorithm:** A method—a sequence of steps—that describes how to solve a problem or class of problems.

When we describe an algorithm, we are describing what we—as human problem solvers—need to do to solve a problem. We can provide a description in a variety of ways: English language, diagrams, flow charts, whatever works for us. Having worked out a solution to a problem in our own best way, we can translate that algorithm into Python code that realizes that algorithm.

For our rainfall problem, how can we develop our algorithm? First, we observe that the problem statement is a combination of linear measurement (inches) and square measurement (acres), but the desired result is in volume (gallons). We must find some intermediate unit for the conversion process. Representing volume in cubic feet is one way; metric units would work as well.

Our algorithm starts with these steps:

1. Prompt the user for the number of inches that have fallen.
2. Find the volume (in cubic feet) of water (where volume = depth * area).
3. Convert the volume (in cubic feet) to gallons.

The Internet can provide the conversion formulas:

```
1 acre = 43,560 square feet
1 cubic foot = 7.48051945 gallons
```

With this information, we can start on our algorithm. Let's begin parts 2 and 3 assuming just 1 inch of rain on an acre:

1. Find the volume in cubic feet of water of 1 inch over 1 acre.
   1 inch is equivalent to 1/12 foot
   volume = depth * area = (1/12)*43,560 cubic feet
2. Convert the volume in cubic feet to gallons.
   gallons = volume * 7.48051945

Now let's try this in Python. We can begin in the Python shell and try out the volume formula:

```
>>> volume = (1/12) * 43560
>>> volume
3630.0
```

Note that the type of `volume` is floating point even though the arguments are integers. This his how division works in Python; it always yields a *`float`*.

Now let's include the conversion to gallons:

```
>>> volume = (1/12) * 43560
>>> volume
3630.0
>>> gallons = volume * 7.48051945
>>> gallons
27154.2856035
```

That looks reasonable. Now we should include part one and prompt the user for the number of inches that have fallen. To do this we need the `input` function, so let's add that to the previous program. We prompt the user for a value `inches` and divide that by 12 to obtain our volume. Let's take a look:

```
>>> inches = input("How many inches of rain have fallen:")
How many inches of rain have fallen:1
>>> volume = (inches/12) * 43560
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>>
```

Hmm, an error. What is the problem? If we look at the error, it gives us a hint: "unsupported operand type(s) for /: 'str' and 'int'. How did a *`str`* get in there?

Yes, that's right. If we do the `input` in isolation, we can see the problem:

```
>>> inches = input("How many inches of rain have fallen:")
How many inches of rain have fallen:1
>>> inches
'1'
>>>
```

The `input` function returns what the user provides as a string (note the quotes around the 1). We need to convert it to a number to get the result we want. Let's take the value from the `input` function and convert it using the *`int`* function. Let's be smarter about naming our `inches` variable as well, so we can tell one type from another:

```
>>> inches_str = input("How many inches of rain have fallen:")
How many inches of rain have fallen:1
>>> inches_int = int(inches_str)
>>> volume = (inches_int/12) * 43560
>>> volume
3630.0
>>> gallons = volume * 7.48051945
>>> gallons
27154.2856035
```

That looks reasonable. Now let's put it together in a program (in IDLE) and call it `rain.py`. We ran the program by selecting "Run ⇒ Run Module" (or F5). See Code Listing 1.2.

Code Listing 1.2

```
# Calculate rainfall in gallons for some number of inches on 1 acre.
inches_str = input("How many inches of rain have fallen: ")
inches_int = int(inches_str)
volume = (inches_int/12)*43560
gallons = volume * 7.48051945
print(inches_int," in. rain on 1 acre is", gallons, "gallons")
```

```
>>> ================================ RESTART ================================
>>>
How many inches of rain have fallen: 1
1  in. rain on 1 acre is 27154.2856035 gallons
>>> ================================ RESTART ================================
>>>
How many inches of rain have fallen: 2
2  in. rain on 1 acre is 54308.571207 gallons
>>> ================================ RESTART ================================
>>>
How many inches of rain have fallen: 0.5
Traceback (most recent call last):
  File "/Users/enbody/Documents/book/tpocup/ch01/programs/program1-3.py",
line 3, in <module>
    inches_int = int(inches_str)
ValueError: invalid literal for int() with base 10: '0.5'
```

For 1 and 2 inches it worked great, but for a value of 1/2 inch, 0.5, we got an error. Why? Again, the error message describes the problem: "invalid literal for int() with base 10: '0.5'. We entered a floating-point value in response to the inches prompt, and Python cannot convert the string `'0.5'` to an integer.

How can we fix this? If we are to allow floating-point input, and it seems reasonable to do so, then we should convert the user-provided value to a `float`, not an `int`, to avoid this problem.

Therefore, we change the conversion to be the `float` function, and to help readability we change the name of `inches_int` to `inches_float`. Then we test it again:

Code Listing 1.3

```
# Calculate rainfall in gallons for some number of inches on 1 acre.
inches_str = input("How many inches of rain have fallen: ")
inches_float = float(inches_str)
volume = (inches_float/12)*43560
gallons = volume * 7.48051945
print(inches_float," in. rain on 1 acre is", gallons, "gallons")
```

```
>>> ================================ RESTART ================================
>>>
How many inches of rain have fallen: 1
1.0  in. rain on 1 acre is 27154.2856035 gallons
>>> ================================ RESTART ================================
>>>
How many inches of rain have fallen: 2
2.0  in. rain on 1 acre is 54308.571207 gallons
>>> ================================ RESTART ================================
>>>
How many inches of rain have fallen: 0.5
0.5  in. rain on 1 acre is 13577.14280175 gallons
```

The result is fine, but the output isn't very pretty. Take a look ahead at Section 4.4 for ways to make the output prettier.

1.9.1 New Rule, Testing

One of the things you should learn from the development of the previous algorithm is the importance of testing. You need to test your approach at every point, in as many ways as you can imagine, to make sure your program does what it is supposed to do. This is such an important point that we are going to add a new programming rule:

> **Rule 5:** Test your code, often and thoroughly!

Testing is so important that a number of recent programming paradigms have appeared that emphasize the development of *tests first*, even before the code is written. Look up the

concepts of "test-driven development" and "extreme programming" on the Internet to get a feel for these approaches.

Remember, only in testing your code—all of it—can you be assured that it does what you intended it to do.

1.10 VISUAL VIGNETTE: TURTLE GRAPHICS

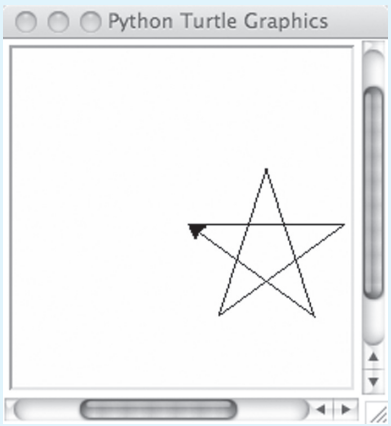
Python version 2.6 introduced a simple drawing tool known as Turtle Graphics. Appendix B provides an introduction and describes a few of the many commands that are available. The concept is that there is a turtle that you command to move forward, right, and left combined with the ability to have the turtle's pen move up or down.

Let's draw a simple five-pointed star. We begin by importing the turtle module. By default, the turtle begins in the middle of the window, pointing right with the pen down. We then have the turtle repeatedly turn and draw a line.

Code Listing 1.4

```
# Draw a 5-pointed star.
import turtle

turtle.forward(100)
turtle.right(144)
turtle.forward(100)
turtle.right(144)
turtle.forward(100)
turtle.right(144)
turtle.forward(100)
turtle.right(144)
turtle.forward(100)
```



As with any problem, the hard part is the thinking that goes into figuring out the details. In this case, what are the angles that we need on each turn? Notice that the center of the star is a regular pentagon. A quick Internet check reveals that each internal angle is 108°. One can view the star as five (isosceles) triangles attached to the pentagon. Because a side of the pentagon extends into a side of a triangle, supplementary angles are formed so the base angle of each triangle is 72° (supplementary angles: 180° − 108°). The two base angles of an isosceles triangle are equal and combine to be 144°, so the third angle must be 36° (a triangle has 180°: 180° − 144°). To make the sharp turn, at each point of the star we need to make a 144° turn (180° − 36°). That is why we have `turtle.right(144)` for each point.

Summary

In this chapter, we introduced a simple but complete program followed by a description of expressions vs. assignments, whitespace and indentation, and finally operators. Most important, we showed how to use the Python shell to practice programming.

Elements

- Keywords: Table 1.1 on page 47
- Operators: Table 1.2 on page 47
- Names
  - begin with a letter; otherwise letters, digits, and underscore
  - beginning with an underscore has special meaning left for later
- Namespace
  Association between a name and an object
- Expression
  Expression is similar to a mathematical expression: it returns a value
- Statement
  Statement performs a task (side effect); does not return a value
- Assignment LHS = RHS
  Steps
  - evaluate expression on RHS; return value
  - associate value from RHS with the name on the LHS
- Modules
  Code that can be imported

Built-In Types

- `int`
  - integers, of any length
  - operations: +, -, *, /, //, %
    note: // is quotient; % is remainder (mod)
- `float`
  - floating-point, a.k.a. decimals
  - operations: +, -, *, /, //, %
- Others
  - Booleans: Chapter 2
  - Strings: Chapter 4
  - Lists: Chapter 7
  - Dictionaries and sets: Chapter 9

Rules

- **Rule 1:** Think before you program!
- **Rule 2:** A program is a human-readable essay on problem solving that also happens to execute on a computer.
- **Rule 3:** The best way to improve your programming and problem skills is to practice!
- **Rule 4:** A foolish consistency is the hobgoblin of little minds.
- **Rule 5:** Test your code, often and thoroughly!

Exercises

1. What is a program?
2. Python is an interpreted language. What does *interpreted* mean in this context?
3. What is a Python *comment*? How do you indicate a comment? What purpose does it serve?
4. What is a *namespace* in Python?
5. Whitespace:
   (a) What is whitespace in Python?
   (b) When does whitespace matter?
   (c) When does whitespace not matter?

6. Explain the difference between a statement and an expression. Give an example of both, and explain what is meant by a statement having a *side effect*.

7. Mixed operations:
   (a) What type results when you divide an integer by a float? A float by an integer?
   (b) Explain why that resulting type makes sense (as opposed to some other type).

8. Consider integer values of a, b, and c and the expression (a + b) * c. In mathematics, we can substitute square brackets, [ ], or curly braces, { }, for parentheses, ( ). Is that same substitution valid in Python? Try it.

9. Write a Python program that prompts for a number. Take that number, add 2, multiply by 3, subtract 6, and divide by 3. You should get the number you started with.

10. A nursery rhyme: *As I was going to St. Ives, I met a man with seven wives. Every wife had seven sacks, and every sack had seven cats, and every cat had seven kittens. Kittens, cats, sacks, and wives, how many were going to St. Ives?* There are interesting aspects to this puzzle, such as who is actually going to St. Ives. For our purposes, assume that everyone and everything is headed to St. Ives. Write a program to calculate that total.

11. Assignment:
```
my_int = 5
my_int = my_int + 3
print(my_int)
```
   (a) If you execute the three lines of code, what will be printed? Explain your answer using the rules of assignment.
   (b) Rewrite `my_int = my_int + 3` using the `+=` symbol.

12. Assignment:
```
my_var1 = 7.0
my_var2 = 5
print(my_var1 % my_var2)
```
   If you execute these three lines of code, what will be printed?

13. Try to predict what will be printed by the following:
```
x = 4
y = 5
print(x//y)
```

14. Given the expression 30 - 3 ** 2 + 8 // 3 ** 2 * 10,
   (a) What is the output of the expression? (You can check your answer in the Python shell.)
   (b) Based on precedence and associativity of the operators in Python, correctly parenthesize the expression such that you get the same output as above.

15. (Order of operations) One example expression was 2 + 3 * 5, the value of which was 17. Using the same expression, include parentheses to arrive at a different value for the expression.

16. Predict the output (check your answer in the Python shell):

(a) 2**2**3
(b) 2**(2**3)
(c) (2**2)**3
Why do two of the expressions have the same output?
Rewrite expression (c) with one exponention operator (**) and one multiplication operator (*).

17. Prompt for input and then print the input as a string, an integer, and a float-point value. What values can you input and print without errors being generated?

18. (Illegal expressions) In a Python shell, try some of the illegal expressions mentioned in the chapter and observe the error messages. For example, try assigning a value to a keyword, such as `and = 4`. Each error message begins with the name of the error, and that name is useful in checking for errors. Generate at least five different error messages.

19. Table 1.2 lists Python operators, some of which may have been unfamiliar to you. Open up the Python shell and experiment with unfamiliar operators and see how many you can define.

20. We know from mathematics that parentheses can change the order of operations. For example, consider $a + b * c$, $(a + b) * c$, and $a + (b * c)$. In general, two of those expressions will be the same and one will be different. Through trial and error, find one set of integer values for a, b, and c so that all three expressions have the same value and a != b != c.

21. Consider the expression $(a + b) * c$, but with string values for a, b, and c. Enter that into the Python shell. What happens? Why?

---

22. (Integer operators) One way to determine whether an integer is even is to divide the number by 2 and check the remainder. Write a three-line program that prompts for a number, converts the input to an integer, and prints a 0 if the number is even and a 1 if the number is odd.

23. Write a program to calculate the volume of water in liters when 1 centimeter of water falls on 1 hectare.

24. Using Turtle Graphics, draw a *six*-pointed star.

25. A day has 86,400 secs (24*60*60). Given a number in the range 1 to 86,400, output the current time as hours, minutes, and seconds with a 24-hour clock. For example: 70,000 sec is 19 hours, 26 minutes, and 40 seconds.

26. A telephone directory has N lines on each page and each page has exactly C columns. An entry in any column has a name with the corresponding telephone number. On which page, column, and line is the Xth entry (name and number) present? (Assume that page, line, column numbers, and X all start from 1.)

27. If the lengths of the two parallel sides of a trapezoid are X meters and Y meters, respectively, and the height is H meters, what is the area of the trapezoid? Write Python code to output the area.

28. Simple interest is calculated by the product of the principal, number of years, and interest, all divided by 100. Write code to calculate the simple interest on a principal amount of $10,000 for a duration of 5 years with the rate of interest equal to 12.5%.

29. Consider a triangle with sides of length 3, 7, and 9. The law of cosines states that given three sides of a triangle (a, b, and c) and the angle C between sides a and b: $c^2 = a^2 + b^2 - 2*a*b*cos(C)$. Write Python code to calculate the three angles in the triangle.

30. Checking the user input for errors is a vital part of programming. The simple program below attempts to take a string input and convert it into an integer. What will happen if the user enters "Hello World" at the prompt rather than a number? Can you think of a way that the program can be altered to handle this input? (Hint: Think about adjusting how the program handles different types of input.)

```
Raw1 = input ('Please enter a number:')
Int1 = int (Raw1)
```

31. The radius and mass of the Earth are $r = 6378 \times 10^3$ meters and $m1 = 5.9742 \times 10^{24}$ kg, respectively. Mr. Jones has a mass of X kg. Prompt the user to input X and then calculate the gravitational force (F) and acceleration due to gravity (g) caused by the gravitational force exerted on him by the Earth. Remember, $F = G(m1)(m2)/(r^2)$ and $F = mg$. Let the universal gravitational constant $G = 6.67300 \times 10^{-11}$ (in units of $m^3 kg^{-1} s^{-2}$ assuming the MKS [meter-kilogram-second] system). Check that the resulting value of g is close to 9.8 m/s^2.

32. (Using modules) Python comes with hundreds of modules. Here is a challenge for you: find a module that you can import that will generate today's date so you can print it. Use your favorite search engine for help in finding which module you need and how to use it. In the end, your task is to do the following:

```
>>> print("Today's date is:", X )
Today's date is: 2009-05-22
```

33. In football, there is a statistic for quarterbacks called the *passer rating*. To calculate the passer rating, you need five inputs: pass completions, pass attempts, total passing yards,

touchdowns, and interceptions. There are five steps in the algorithm. Write a program that asks for the five inputs and then prints the pass rating:

(a) C is the "completions per attempt" times 100 minus 30, all divided by 20.
(b) Y is the "yards per attempt" minus 3, all divided by 4.
(c) T is the "touchdowns per attempt" times 20.
(d) I is 2.375 minus ("interceptions per attempts" times 35).
(e) The pass rating is the sum of C, Y, T, and I, all divided by 6 and then multiplied by 100.

34. Body mass index (BMI) is a number calculated from a person's weight and height. According to the Centers for Disease Control and Prevention, the BMI is a fairly reliable indicator of body fatness for most people. BMI does not measure body fat directly, but research has shown that BMI correlates to direct measures of body fat, such as underwater weighing and dual-energy X-ray absorptiometry. The formula for BMI is

$$weight/height^2$$

where *weight* is in kilograms and *height* is in meters.

(a) Write a program that prompts for metric weight and height and outputs the BMI.
(b) Write a program that prompts for weight in pounds and height in inches, converts the values to metric, and then calculates the BMI.

Programming Projects

1. The Great Lakes are how big?

The Great Lakes in the United States contain roughly 22% of the world's fresh surface water (22,810 km^3). It is hard to conceive how much water that is. Write a program to calculate how deep it would be if all the water in the Great Lakes were spread evenly across the 48 contiguous U.S. states. You will need to do some Internet research to determine the area of that region.

2. Where is *Voyager* 1?

The *Voyager* 1 spacecraft, launched September 15, 1977, is the farthest-traveling Earth-made object. It is presently on the outer edges of our solar system. The NASA update page on September 25, 2009, reported it as being a distance of approximately 16,637,000,000 miles from the sun, traveling away from the sun at 38,241 miles/hour.

Write a program that will prompt the user for an integer number that indicates the number of days after 9/25/09. You will calculate the distance of *Voyager* from the sun using the numbers from 9/25/09 (assume that velocity is constant) plus the entered number of days, and report:

- Distance in miles
- Distance in kilometers (1.609344 kilometers/mile)

- Distance in astronomical units (AU, 92,955,887.6 miles/AU)
- Round-trip time for radio communication in hours. Radio waves travel at the speed of light, listed at 299,792,458 meters/second.

3. Oil conversions and calculations

Write a program that will prompt the user for a floating-point number that stands for gallons of gasoline. You will reprint that value along with other information about gasoline and gasoline usage:

- Number of liters
- Number of barrels of oil required to make this amount of gasoline
- Number of pounds of CO_2 produced
- Equivalent energy amount of ethanol gallons
- Price in U.S. dollars

Here are some approximate conversion values:

- 1 barrel of oil produces 19.5 gallons of gasoline.
- 1 gallon of gasoline produces 20 pounds of CO_2 gas when burned.
- 1 gallon of gasoline contains 115,000 BTU (British thermal units) of energy.
- 1 gallon of ethanol contains 75,700 BTU of energy.
- 1 gallon of gasoline costs $3.00/gallon.

Look on the Internet for some interesting values for input, such as the average number of gallons consumed per person per year, consumed by the country per day, or consumed per year.

4. Population estimation

The U.S. Census provides information on its web page (`http://www.census.gov`) about the current U.S. population as well as approximate rates of change.

Three rates of change are provided:

- There is a birth every 7 seconds.
- There is a death every 13 seconds.
- There is a new immigrant every 35 seconds.

These are obviously approximations of birth, death, and immigration rates, but they can assist in providing population estimates in the near term.

Write a program that takes years as input (as an integer) and prints out an estimated population (as an integer). Assume that the current population is 307,357,870, and assume that there are exactly 365 days in a year.

Hint: Note that the rate units are in seconds.