
This chapter addresses the question “What is computer science?” We begin by introducing the essence of computational problem solving via some classic examples. Next, computer algorithms, the heart of computational problem solving, are discussed. This is followed by a look at computer hardware (and the related issues of binary representation and operating systems) and computer software (and the related issues of syntax, semantics, and program translation). The chapter finishes by presenting the process of computational problem solving, with an introduction to the Python programming language.

OBJECTIVES

After reading this chapter and completing the exercises, you will be able to:

- ◆ Explain the essence of computational problem solving
- ◆ Explain what a computer algorithm is
- ◆ Explain the fundamental components of digital hardware
- ◆ Explain the role of binary representation in digital computing
- ◆ Explain what an operating systems is
- ◆ Explain the fundamental concepts of computer software
- ◆ Explain the fundamental features of IDLE in Python
- ◆ Modify and execute a simple Python program

CHAPTER CONTENTS

Motivation

Fundamentals

1.1 What Is Computer Science?

1.2 Computer Algorithms

1.3 Computer Hardware

1.4 Computer Software

Computational Problem Solving

1.5 The Process of Computational Problem Solving

1.6 The Python Programming Language

1.7 A First Program—Calculating the Drake Equation

MOTIVATION

Computing technology has changed, and is continuing to change the world. Essentially every aspect of life has been impacted by computing. Just-in-time inventory allows companies to significantly reduce costs. Universal digital medical records promise to save the lives of many of the estimated 100,000 people who die each year from medical errors. Vast information resources, such as Wikipedia, now provide easy, quick access to a breadth of knowledge as never before. Information sharing via Facebook and Twitter has not only brought family and friends together in new ways, but has also helped spur political change around the world. New interdisciplinary fields combining computing and science will lead to breakthroughs previously unimaginable. Computing-related fields in almost all areas of study are emerging (see Figure 1-1).



In the study of computer science, there are fundamental principles of computation to be learned that will never change. In addition to these principles, of course, there is always changing technology. That is what makes the field of computer science so exciting. There is constant change and advancement, but also a foundation of principles to draw from. What can be done with computation is limited only by our imagination. With that said, we begin our journey into the world of computing. I have found it an unending fascination—I hope that you do too. Bon voyage!

Various Computational-Related Fields		
Computational Biology	Computational Medicine	Computational Journalism
Computational Chemistry	Computational Pharmacology	Digital Humanities
Computational Physics	Computational Economics	Computational Creativity
Computational Mathematics	Computational Textiles	Computational Music
Computational Materials Science	Computational Architecture	Computational Photography
Computer-Aided Design	Computational Social Science	Computational Advertising
Computer-Aided Manufacturing	Computational Psychology	Computational Intelligence

FIGURE 1-1 Computing-Related Specialized Fields

FUNDAMENTALS

1.1 What Is Computer Science?

Many people, if asked to define the field of computer science, would likely say that it is about programming computers. Although programming is certainly a primary activity of computer science, programming languages and computers are only *tools*. What computer science is fundamentally

about is **computational problem solving**—that is, solving problems by the use of computation (Figure 1-2).

This description of computer science provides a succinct definition of the field. However, it does not convey its tremendous *breadth and diversity*. There are various areas of study in computer science including software engineering (the design and implementation of large software systems), database management, computer networks, computer graphics, computer simulation, data mining, information security, programming language design, systems programming, computer architecture, human–computer interaction, robotics, and artificial intelligence, among others.

The definition of computer science as computational problem solving begs the question: *What is computation?* One characterization of computation is given by the notion of an *algorithm*. The definition of an algorithm is given in section 1.2. For now, consider an algorithm to be a series of steps that can be systematically followed for producing the answer to a certain type of problem. We look at fundamental issues of computational problem solving next.



FIGURE 1-2 Computational Problem Solving

Computer science is fundamentally about computational problem solving.

1.1.1 The Essence of Computational Problem Solving

In order to solve a problem computationally, two things are needed: a *representation* that captures all the relevant aspects of the problem, and an *algorithm* that solves the problem by use of the representation. Let's consider a problem known as the **Man, Cabbage, Goat, Wolf problem** (Figure 1-3).

A man lives on the east side of a river. He wishes to bring a cabbage, a goat, and a wolf to a village on the west side of the river to sell. However, his boat is only big enough to hold himself, and either the cabbage, goat, or wolf. In addition, the man cannot leave the goat alone with the cabbage because the goat will eat the cabbage, and he cannot leave the wolf alone with the goat because the wolf will eat the goat. How does the man solve his problem?

There is a simple algorithmic approach for solving this problem by simply trying all possible combinations of items that may be rowed back and forth across the river. Trying all possible solutions to a given problem is referred to as a *brute force approach*. What would be an appropriate



FIGURE 1-3 Man, Cabbage, Goat, Wolf Problem

representation for this problem? Since only the relevant aspects of the problem need to be represented, all the irrelevant details can be omitted. A representation that leaves out details of what is being represented is a form of **abstraction**.

The use of abstraction is prevalent in computer science. In this case, is the color of the boat relevant? The width of the river? The name of the man? No, the only relevant information is *where* each item is at each step. The collective location of each item, in this case, refers to the *state* of the problem. Thus, the *start state* of the problem can be represented as follows.

man cabbage goat wolf

[E, E, E, E]

In this representation, the symbol E denotes that each corresponding object is on the east side of the river. If the man were to row the goat across with him, for example, then the representation of the new problem state would be

man cabbage goat wolf

[W, E, W, E]

in which the symbol W indicates that the corresponding object is on the west side of the river—in this case, the man and goat. (The locations of the cabbage and wolf are left unchanged.) A solution to this problem is a sequence of steps that converts the initial state,

[E, E, E, E]

in which all objects are on the east side of the river, to the *goal state*,

[W, W, W, W]

in which all objects are on the west side of the river. Each step corresponds to the man rowing a particular object across the river (or the man rowing alone). As you will see, the Python programming language provides an easy means of representing sequences of values. The remaining task is to develop or find an existing algorithm for computationally solving the problem using this representation. The solution to this problem is left as a chapter exercise.

As another example computational problem, suppose that you needed to write a program that displays a calendar month for any given month and year, as shown in Figure 1-4. The representation of this problem is rather straightforward. Only a few values need to be maintained—the month and year, the number of days in each month, the names of the days of the week, and the day of the week that the first day of the month falls on. Most of these values are either provided by the user (such as the month and year) or easily determined (such as the number of days in a given month).

The less obvious part of this problem is how to determine the day of the week that a given date falls on. You would need an algorithm that can compute this. Thus, no matter how well you may know a given programming language or how good a programmer you may be, without such an algorithm you could not solve this problem.

MAY 2012						
Sun	Mon	Tues	Wed	Thur	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

FIGURE 1-4 Calendar Month

In order to solve a problem computationally, two things are needed: a *representation* that captures all the relevant aspects of the problem, and an *algorithm* that solves the problem by use of the representation.

1.1.2 Limits of Computational Problem Solving

Once an algorithm for solving a given problem is developed or found, an important question is, “Can a solution to the problem be found in a reasonable amount of time?” If not, then the particular algorithm is of limited practical use.

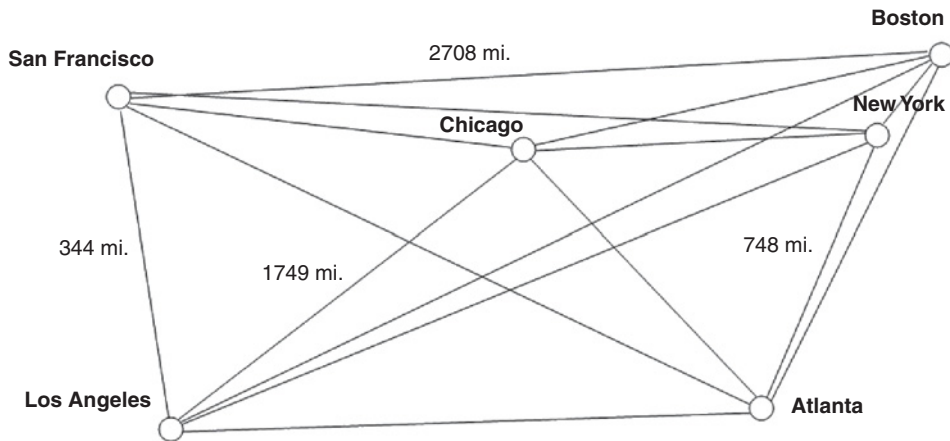


FIGURE 1-5 Traveling Salesman Problem

The **Traveling Salesman problem** (Figure 1-5) is a classic computational problem in computer science. The problem is to find the shortest route of travel for a salesman needing to visit a given set of cities. In a brute force approach, the lengths of all possible routes would be calculated and compared to find the shortest one. For ten cities, the number of possible routes is $10!$ (10 factorial), or over three and a half million (3,628,800). For twenty cities, the number of possible routes is $20!$, or over two and a half quintillion (2,432,902,008,176,640,000). If we assume that a computer could compute the lengths of one million routes per second, it would take over 77,000 years to find the shortest route for twenty cities by this approach. For 50 cities, the number of possible routes is over 10^{64} . In this case, it would take more time to solve than the age of the universe!

A similar problem exists for the game of chess (Figure 1-6). A brute force approach for a chess-playing program would be to “look ahead” to *all* the eventual outcomes of every move that can be made in deciding each next move. There



FIGURE 1-6 Game of Chess

are approximately 10^{120} possible chess games that can be played. This is related to the average number of look-ahead steps needed for deciding each move. How big is this number? There are approximately 10^{80} atoms in the observable universe, and an estimated 3×10^{90} grains of sand to fill the universe solid. Thus, *there are more possible chess games that can be played than grains of sand to fill the universe solid!* For problems such as this and the Traveling Salesman problem in which a brute-force approach is impractical to use, more efficient problem-solving methods must be discovered that find either an exact or an approximate solution to the problem.

Any algorithm that correctly solves a given problem must solve the problem in a reasonable amount of time, otherwise it is of limited practical use.

Self-Test Questions

1. A good definition of computer science is “the science of programming computers.” (TRUE/FALSE)
2. Which of the following areas of study are included within the field of computer science?
 - (a) Software engineering
 - (b) Database management
 - (c) Information security
 - (d) All of the above
3. In order to computationally solve a problem, two things are needed: a representation of the problem, and an _____ that solves it.
4. Leaving out detail in a given representation is a form of _____.
5. A “brute-force” approach for solving a given problem is to:
 - (a) Try all possible algorithms for solving the problem.
 - (b) Try all possible solutions for solving the problem.
 - (c) Try various representations of the problem.
 - (d) All of the above
6. For which of the following problems is a brute-force approach practical to use?
 - (a) Man, Cabbage, Goat, Wolf problem
 - (b) Traveling Salesman problem
 - (c) Chess-playing program
 - (d) All of the above

ANSWERS: 1. False, 2. (d), 3. algorithm, 4. abstraction, 5. (b), 6. (a)

1.2 Computer Algorithms

This section provides a more complete description of an algorithm than given above, as well as an example algorithm for determining the day of the week for a given date.

1.2.1 What Is an Algorithm?

An **algorithm** is a finite number of clearly described, unambiguous “doable” steps that can be systematically followed to produce a desired result for given input in a finite amount of time (that is, it

eventually terminates). Algorithms solve *general* problems (determining whether any given number is a prime number), and not specific ones (determining whether 30753 is a prime number). Algorithms, therefore, are general computational methods used for solving particular problem instances.

The word “algorithm” is derived from the ninth-century Arab mathematician, Al-Khwarizmi (Figure 1-7), who worked on “written processes to achieve some goal.” (The term “algebra” also derives from the term “al-jabr,” which he introduced.)

Computer algorithms are central to computer science. They provide step-by-step methods of computation that a machine can carry out. Having high-speed machines (computers) that can consistently follow and execute a given set of instructions provides a reliable and effective means of realizing computation. However, *the computation that a given computer performs is only as good as the underlying algorithm used.* Understanding what can be effectively programmed and executed by computers, therefore, relies on the understanding of computer algorithms.

Persian_khwarizmi/Wikimedia Commons



FIGURE 1-7 Al-Khwarizmi
(Ninth Century A.D.)

An **algorithm** is a finite number of clearly described, unambiguous “doable” steps that can be systematically followed to produce a desired result for given input in a finite amount of time.

1.2.2 Algorithms and Computers: A Perfect Match

Much of what has been learned about algorithms and computation since the beginnings of modern computing in the 1930s–1940s could have been studied centuries ago, since the study of algorithms does not depend on the existence of computers. The algorithm for performing long division is such an example. However, most algorithms are not as simple or practical to apply manually. Most require the use of computers either because they would require too much time for a person to apply, or involve so much detail as to make human error likely. Because *computers can execute instructions very quickly and reliably without error*, algorithms and computers are a perfect match! Figure 1-8 gives an example algorithm for determining the day of the week for any date between January 1, 1800 and December 31, 2099.

Because computers can execute instructions very quickly and reliably without error, algorithms and computers are a perfect match.

To determine the day of the week for a given **month**, **day**, and **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits** + floor(**year_digits** / 4)
4. If **century_digits** equals 18, then add 2 to **value**, else if **century_digits** equals 20, then add 6 to **value**.
5. If the **month** is equal to January and **year** is not a leap year, then add 1 to **value**, else,
 - if the **month** is equal to February and the **year** is a leap year, then add 3 to **value**; if not a leap year, then add 4 to **value**, else,
 - if the **month** is equal to March or November, then add 4 to **value**, else,
 - if the **month** is equal to May, then add 2 to **value**, else,
 - if the **month** is equal to June, then add 5 to **value**, else,
 - if the **month** is equal to August, then add 3 to **value**, else,
 - if the **month** is equal to October, then add 1 to **value**, else,
 - if the **month** is equal to September or December, then add 6 to **value**,
6. Set **value** equal to (**value** + **day**) mod 7.
7. If **value** is equal to 1, then the day of the week is Sunday; else if **value** is equal to 2, day of the week is Monday; else if **value** is equal to 3, day of the week is Tuesday; else if **value** is equal to 4, day of the week is Wednesday; else if **value** is equal to 5, day of the week is Thursday; else if **value** is equal to 6, day of the week is Friday; else if **value** is equal to 0, day of the week is Saturday

FIGURE 1-8 Day of the Week Algorithm

Note that there is no value to add for the months of April and July.

Self-Test Questions

1. Which of the following are true of an algorithm?
 - (a) Has a finite number of steps
 - (b) Produces a result in a finite amount of time
 - (c) Solves a general problem
 - (d) All of the above
2. Algorithms were first developed in the 1930–1940s when the first computing machines appeared. (TRUE/FALSE)
3. Algorithms and computers are a “perfect match” because: (Select all that apply.)
 - (a) Computers can execute a large number of instructions very quickly.
 - (b) Computers can execute instructions reliably without error.
 - (c) Computers can determine which algorithms are the best to use for a given problem.

4. Given that the year 2016 is a leap year, what day of the week does April 15th of that year fall on? Use the algorithm in Figure 1-8 for this.
5. Which of the following is an example of an algorithm? (Select all that apply.)
 - (a) A means of sorting any list of numbers
 - (b) Directions for getting from your home to a friend's house
 - (c) A means of finding the shortest route from your house to a friend's house.

ANSWERS: 1. (d), 2. False, 3. (a,b) 4. Friday, 5. (a,c)

1.3 Computer Hardware

Computer hardware comprises the physical part of a computer system. It includes the all-important components of the *central processing unit* (CPU) and *main memory*. It also includes *peripheral components* such as a keyboard, monitor, mouse, and printer. In this section, computer hardware and the intrinsic use of binary representation in computers is discussed.

1.3.1 Digital Computing: It's All about Switches

It is essential that computer hardware be reliable and error free. If the hardware gives incorrect results, then any program run on that hardware is unreliable. A rare occurrence of a hardware error was discovered in 1994. The widely used Intel processor was found to give incorrect results only when certain numbers were divided, estimated as likely to occur once every 9 billion divisions. Still, the discovery of the error was very big news, and Intel promised to replace the processor for any one that requested it.

The key to developing reliable systems is to keep the design as simple as possible. In digital computing, all information is represented as a series of digits. We are used to representing numbers using base 10 with digits 0–9. Consider if information were represented within a computer system this way, as shown in Figure 1-9.

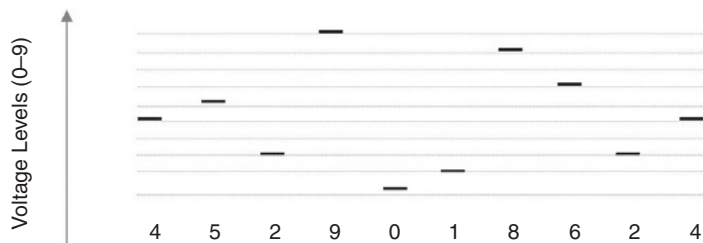


FIGURE 1-9 Decimal Digitalization

In current electronic computing, each digit is represented by a different voltage level. The more voltage levels (digits) that the hardware must utilize and distinguish, the more complex the hardware design becomes. This results in greater chance of hardware design errors. It is a fact of information theory, however, that any information can be represented using only *two* symbols. Because of this, *all information within a computer system is represented by the use of only two digits, 0 and 1, called binary representation*, shown in Figure 1-10.

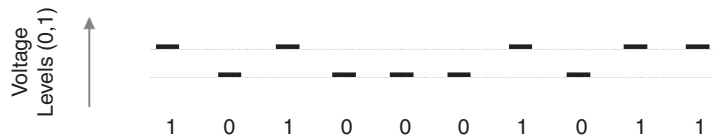


FIGURE 1-10 Binary Digitalization

In this representation, each digit can be one of only two possible values, similar to a light switch that can be either on or off. Computer hardware, therefore, is based on the use of simple electronic “on/off” switches called **transistors** that switch at very high speed. **Integrated circuits** (“chips”), the building blocks of computer hardware, are comprised of millions or even billions of transistors. The development of the transistor and integrated circuits is discussed in Chapter 12. We discuss binary representation next.

All information within a computer system is represented using only two digits, 0 and 1, called **binary representation**.

1.3.2 The Binary Number System

For representing numbers, any base (radix) can be used. For example, in base 10, there are ten possible digits (0, 1, . . . , 9), in which each column value is a power of ten, as shown in Figure 1-11.

10,000,000	1,000,000	100,000	10,000	1,000	100	10	1
10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0
							9 9 = 99

FIGURE 1-11 Base 10 Representation

Other radix systems work in a similar manner. **Base 2** has digits 0 and 1, with place values that are powers of two, as depicted in Figure 1-12.

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	0	0	1	1
0 + 64 + 32 + 0 + 0 + 0 + 2 + 1 = 99							

FIGURE 1-12 Base 2 Representation

As shown in this figure, converting from base 2 to base 10 is simply a matter of adding up the column values that have a 1.

The term **bit** stands for **binary digit**. Therefore, every bit has the value 0 or 1. A **byte** is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits. Although values represented in base 2 are significantly longer than those represented in base 10, binary representation is used in digital computing because of the resulting simplicity of hardware design.

The algorithm for the conversion from base 10 to base 2 is to successively divide a number by two until the remainder becomes 0. The remainder of each division provides the next higher-order (binary) digit, as shown in Figure 1-13.

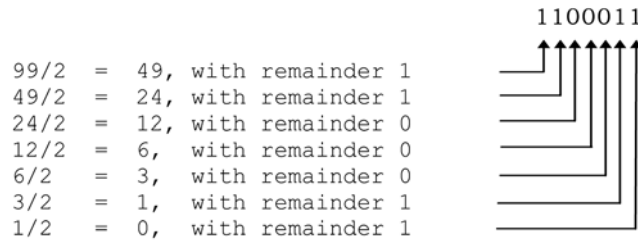


FIGURE 1-13 Converting from Base 10 to Base 2

Thus, we get the binary representation of 99 to be 1100011. This is the same as in Figure 1-12 above, except that we had an extra leading insignificant digit of 0, since we used an eight-bit representation there.

The term **bit** stands for binary digit. A **byte** is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits.

1.3.3 Fundamental Hardware Components

The **central processing unit (CPU)** is the “brain” of a computer system, containing digital logic circuitry able to interpret and execute instructions. **Main memory** is where currently executing programs reside, which the CPU can directly and very quickly access. Main memory is volatile; that is, the contents are lost when the power is turned off. In contrast, **secondary memory** is nonvolatile, and therefore provides long-term storage of programs and data. This kind of storage, for example, can be magnetic (hard drive), optical (CD or DVD), or nonvolatile flash memory (such as in a USB drive). **Input/output devices** include anything that allows for input (such as the mouse and keyboard) or output (such as a monitor or printer). Finally, **buses** transfer data between components within a computer system, such as between the CPU and main memory. The relationship of these devices is depicted in Figure 1-14 below.

The **central processing unit (CPU)** is the “brain” of a computer, containing digital logic circuitry able to interpret and execute instructions.

1.3.4 Operating Systems—Bridging Software and Hardware

An **operating system** is software that has the job of managing and interacting with the hardware resources of a computer. Because an operating system is intrinsic to the operation a computer, it is referred to as **system software**.

Adapted from Peter Astbury/Computing Devices/Wikimedia Commons

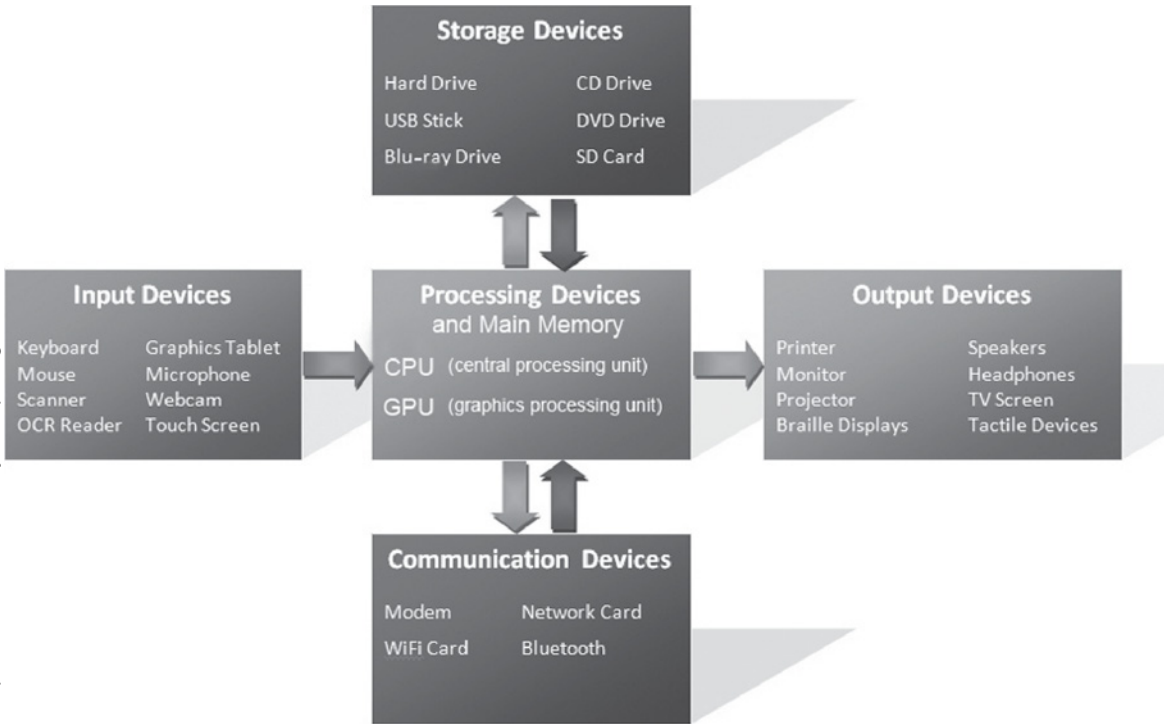


FIGURE 1-14 Fundamental Hardware Components

An operating system acts as the “middle man” between the hardware and executing application programs (see Figure 1-15). For example, it controls the allocation of memory for the various programs that may be executing on a computer. Operating systems also provide a particular user interface. Thus, it is the operating system installed on a given computer that determines the “look and feel” of the user interface and how the user interacts with the system, and not the particular model computer.

An **operating system** is software that has the job of managing the hardware resources of a given computer and providing a particular user interface.

Golftheman/Operating system placement/Wikimedia Commons

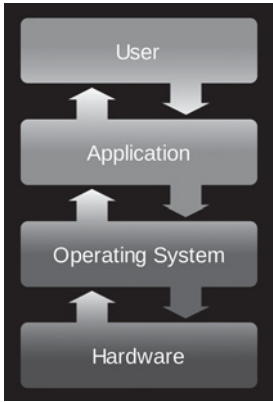


FIGURE 1-15 Operating System

1.3.5 Limits of Integrated Circuits Technology: Moore’s Law

In 1965, Gordon E. Moore (Figure 1-16), one of the pioneers in the development of integrated circuits and cofounder of Intel Corporation, predicted that as a result of continuing engineering

developments, the number of transistors that would be able to be put on a silicon chip would double roughly every two years, allowing the complexity and therefore the capabilities of integrated circuits to grow exponentially. This prediction became known as **Moore's Law**. Amazingly, to this day that prediction has held true. While this doubling of performance cannot go on indefinitely, it has not yet reached its limit.

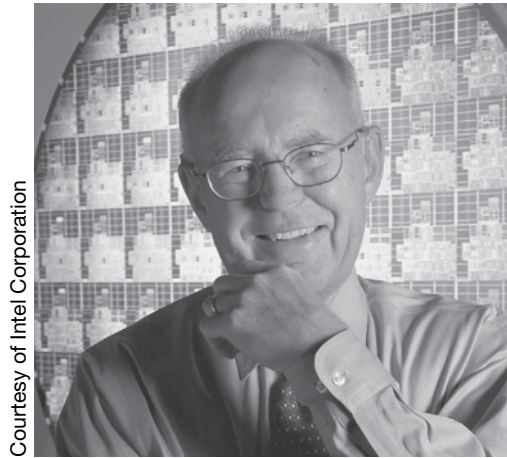


FIGURE 1-16 Gordon E. Moore

Moore's Law states that the number of transistors that can be placed on a single silicon chip doubles roughly every two years.

Self-Test Questions

1. All information in a computer system is in binary representation. (TRUE/FALSE)
2. Computer hardware is based on the use of electronic switches called _____.
3. How many of these electronic switches can be placed on a single integrated circuit, or "chip"?
 - (a) Thousands
 - (b) Millions
 - (c) Billions
4. The term "bit" stands for _____.
5. A bit is generally a group of eight bytes. (TRUE/FALSE)
6. What is the value of the binary representation 0110.
 - (a) 12
 - (b) 3
 - (c) 6
7. The _____ interprets and executes instructions in a computer system.
8. An operating system manages the hardware resources of a computer system, as well as provides a particular user interface. (TRUE/FALSE)
9. Moore's Law predicts that the number of transistors that can fit on a chip doubles about every ten years. (TRUE/FALSE)

1.4 Computer Software

The first computer programs ever written were for a mechanical computer designed by Charles Babbage in the mid-1800s. (Babbage’s Analytical Engine is discussed in Chapter 12). The person who wrote these programs was a woman, Ada Lovelace (Figure 1-17), who was a talented mathematician. Thus, she is referred to as “the first computer programmer.” This section discusses fundamental issues of computer software.

1.4.1 What Is Computer Software?

Computer software is a set of program instructions, including related data and documentation, that can be executed by computer. This can be in the form of instructions on paper, or in digital form. While system software is intrinsic to a computer system, **application software** fulfills users’ needs, such as a photo-editing program. We discuss the important concepts of syntax, semantics, and program translation next.

Computer software is a set of program instructions, including related data and documentation, that can be executed by computer.

1.4.2 Syntax, Semantics, and Program Translation

What Are Syntax and Semantics?

Programming languages (called “artificial languages”) are languages just as “natural languages” such as English and Mandarin (Chinese). *Syntax* and *semantics* are important concepts that apply to all languages.

The **syntax** of a language is a set of characters and the acceptable arrangements (sequences) of those characters. English, for example, includes the letters of the alphabet, punctuation, and properly spelled words and properly punctuated sentences. The following is a syntactically correct sentence in English,

“Hello there, how are you?” The following, however, is not syntactically correct,
 “Hello there, hao are you?”

In this sentence, the sequence of letters “hao” is not a word in the English language. Now consider the following sentence,

“Colorless green ideas sleep furiously.”

This sentence is syntactically correct, but is *semantically* incorrect, and thus has no meaning.

Royal Institution of Great Britain/Photo Researchers, Inc.



FIGURE 1-17 Ada Lovelace “The First Computer Programmer”

The **semantics** of a language is the meaning associated with each syntactically correct sequence of characters. In Mandarin, “Hao” is syntactically correct meaning “good.” (“Hao” is from a system called pinyin, which uses the Roman alphabet rather than Chinese characters for writing Mandarin.) Thus, every language has its own syntax and semantics, as demonstrated in Figure 1-18.

<u>ENGLISH</u>	<u>MANDARIN</u> (pinyin)	<u>MANDARIN</u> (Chinese Characters)
Syntax Hao	Syntax Hao	Syntax 好
Semantics No meaning (syntactically incorrect)	Semantics “Good”	Semantics “Good”

FIGURE 1-18 Syntax and Semantics of Languages

The **syntax** of a language is a set of characters and the acceptable sequences of those characters. The **semantics** of a language is the meaning associated with each syntactically correct sequence of characters.

Program Translation

A central processing unit (CPU) is designed to interpret and execute a specific set of instructions represented in binary form (i.e., 1s and 0s) called **machine code**. Only programs in machine code can be executed by a CPU, depicted in Figure 1-19.

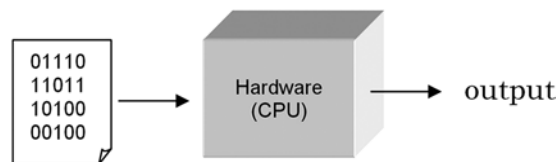


FIGURE 1-19 Execution of Machine Code

Writing programs at this “low level” is tedious and error-prone. Therefore, most programs are written in a “high-level” programming language such as Python. Since the instructions of such programs are not in machine code that a CPU can execute, a translator program must be used. There are two fundamental types of translators. One, called a **compiler**, translates programs directly into machine code to be executed by the CPU, denoted in Figure 1-20.

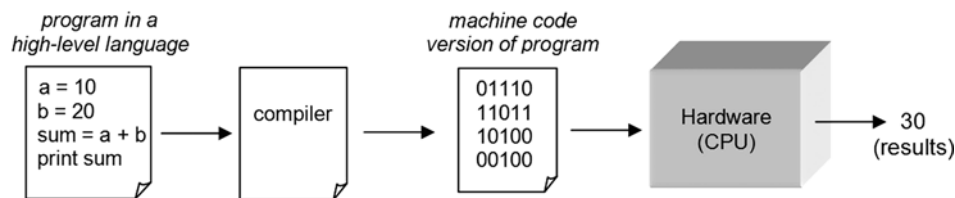


FIGURE 1-20 Program Execution by Use of a Compiler

The other type of translator is called an **interpreter**, which executes program instructions in place of (“running on top of”) the CPU, denoted in Figure 1-21.

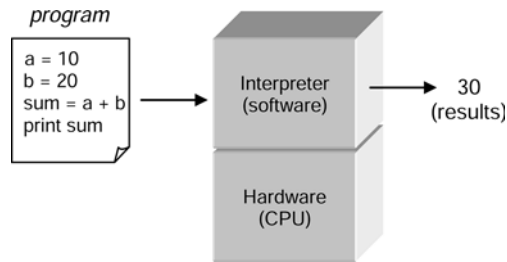


FIGURE 1-21 Program Execution by Use of an Interpreter

Thus, an interpreter can immediately execute instructions as they are entered. This is referred to as **interactive mode**. This is a very useful feature for program development. Python, as we shall see, is executed by an interpreter. On the other hand, compiled programs generally execute faster than interpreted programs. Any program can be executed by either a compiler or an interpreter, as long there exists the corresponding translator program for the programming language that it is written in.

A **compiler** is a translator program that translates programs directly into machine code to be executed by the CPU. An **interpreter** executes program instructions in place of (“running on top of”) the CPU.

Program Debugging: Syntax Errors vs. Semantic Errors

Program debugging is the process of finding and correcting errors (“**bugs**”) in a computer program. Programming errors are inevitable during program development. **Syntax errors** are caused by invalid syntax (for example, entering `prnt` instead of `print`). Since a translator cannot understand instructions containing syntax errors, translators terminate when encountering such errors indicating where in the program the problem occurred.

In contrast, **semantic errors** (generally called **logic errors**) are errors in program logic. Such errors cannot be automatically detected, since translators cannot understand the intent of a given computation. For example, if a program computed the average of three numbers as follows,

$$(\text{num1} + \text{num2} + \text{num3}) / 2.0$$

a translator would have no means of determining that the divisor should be 3 and not 2. *Computers do not understand what a program is meant to do, they only follow the instructions given.* It is up to the programmer to detect such errors. Program debugging is not a trivial task, and constitutes much of the time of program development.

Syntax errors are caused by invalid syntax. **Semantic (logic) errors** are caused by errors in program logic.

1.4.3 Procedural vs. Object-Oriented Programming

Programming languages fall into a number of *programming paradigms*. The two major programming paradigms in use today are *procedural (imperative) programming* and *object-oriented programming*. Each provides a different way of thinking about computation. While most programming languages only support one paradigm, Python supports both procedural and object-oriented programming. We will start with the procedural aspects of Python. We then introduce objects in Chapter 6, and delay complete discussion of object-oriented programming until Chapter 10.

Procedural programming and **object-oriented programming** are two major programming paradigms in use today.

Self-Test Questions

1. Two general types of software are system software and _____ software.
2. The syntax of a given language is,
 - (a) the set of symbols in the language.
 - (b) the acceptable arrangement of symbols.
 - (c) both of the above
3. The semantics of a given language is the meaning associated with any arrangement of symbols in the language. (TRUE/FALSE)
4. CPUs can only execute instructions that are in binary form called _____.
5. The two fundamental types of translation programs for the execution of computer programs are _____ and _____.
6. The process of finding and correcting errors in a computer program is called _____.
7. Which kinds of errors can a translator program detect?
 - (a) Syntax errors
 - (b) Semantic errors
 - (c) Neither of the above
8. Two major programming paradigms in use today are _____ programming and _____ programming.

ANSWERS: 1. application, 2. (c), 3. False, 4. machine code, 5. compilers, interpreters, 6. program debugging, 7. (a), 8. procedural, object-oriented

COMPUTATIONAL PROBLEM SOLVING

1.5 The Process of Computational Problem Solving

Computational problem solving does not simply involve the act of computer programming. It is a *process*, with programming being only one of the steps. Before a program is written, a design for the program must be developed. And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested. These steps are outlined in Figure 1-22.

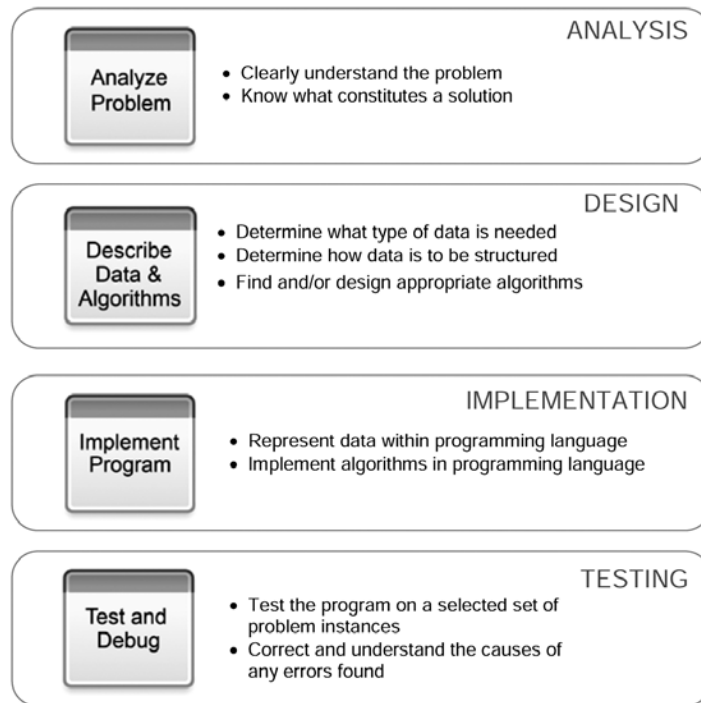


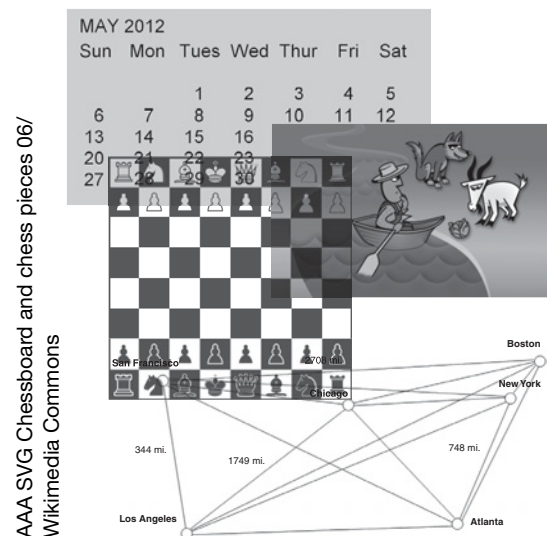
FIGURE 1-22 Process of Computational Problem Solving

1.5.1 Problem Analysis

Understanding the Problem

Once a problem is clearly understood, the fundamental computational issues for solving it can be determined. For each of the problems discussed earlier, the representation is straightforward. For the calendar month problem, there are two algorithmic tasks—determining the first day of a given month, and displaying the calendar month in the proper format. The first day of the month can be obtained by *direct calculation* by use of the algorithm provided in Figure 1-8.

For the Man, Cabbage, Goat, Wolf (MCGW) problem, a brute-force algorithmic approach of trying all possible solutions works very well, since there are a small number of actions that can be taken at each step, and only a relatively small number of steps for reaching a solution. For both the Traveling Salesman problem and the game of chess, the brute-force approach is infeasible. Thus, the computational issue for these problems is to find other, more efficient algorithmic approaches for their



solution. (In fact, methods have been developed for solving Traveling Salesman problems involving tens of thousands of cities. And current chess-playing programs can beat top-ranked chess masters.)

Knowing What Constitutes a Solution

Besides clearly understanding a computational problem, one must know what constitutes a solution. For some problems, there is only one solution. For others, there may be a number (or infinite number) of solutions. Thus, a program may be stated as finding,

- ◆ A solution
- ◆ An *approximate* solution
- ◆ A *best* solution
- ◆ All solutions

For the MCGW problem, there are an infinite number of solutions since the man could pointlessly row back and forth across the river an arbitrary number of times. A *best solution* here is one with the shortest number of steps. (There may be more than one “best” solution for any given problem.) In the Traveling Salesman problem there is only one solution (unless there exists more than one shortest route). Finally, for the game of chess, the goal (solution) is to win the game. Thus, since the number of chess games that can be played is on the order of 10^{120} (with each game ending in a win, a loss, or a stalemate), there are a comparable number of possible solutions to this problem.

1.5.2 Program Design

Describing the Data Needed

For the Man, Cabbage, Goat, Wolf problem, a list can be used to represent the correct location (east and west) of the man, cabbage, goat, and wolf as discussed earlier, reproduced below,

```
man cabbage goat  wolf
[W,      E,      W,      E]
```

For the Calendar Month problem, the data include the month and year (entered by the user), the number of days in each month, and the names of the days of the week. A useful structuring of the data is given below,

```
[month, year]
[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

The month and year are grouped in a single list since they are naturally associated. Similarly, the names of the days of the week and the number of days in each month are grouped. (The advantages of list representations will be made clear in Chapter 4.) Finally, the first day of the month, as determined by the algorithm in Figure 1-8, can be represented by a single integer,

```
0 – Sunday, 1 – Monday, . . . , 6 – Saturday
```

For the Traveling Salesman problem, the distance between each pair of cities must be represented. One possible way of structuring the data is as a table, depicted in Figure 1-23.

For example, the distance from Atlanta to Los Angeles is 2175 miles. There is duplication of information in this representation, however. For each distance from x to y , the distance from y to x is

	Atlanta	Boston	Chicago	Los Angeles	New York City	San Francisco
Atlanta	-	1110	718	2175	888	2473
Boston	1110	-	992	2991	215	3106
Chicago	718	992	-	2015	791	2131
Los Angeles	2175	2991	2015	-	2790	381
New York City	888	215	791	2790	-	2901
San Francisco	2473	3106	2131	381	2901	-

FIGURE 1-23 Table Representation of Data

also represented. If the size of the table is small, as here, this is not much of an issue. However, for a significantly larger table, significantly more memory would be wasted during program execution. Since only half of the table is really needed (for example, the shaded area in the figure), the data could be represented as a list of lists instead,

```
[ ['Atlanta', ['Boston', 1110], ['Chicago', 718], ['Los Angeles', 2175], ['New York', 888],  
  ['San Francisco', 2473] ],  
  ['Boston', ['Chicago', 992], ['Los Angeles', 2991], ['New York', 215], ['San Francisco', 3106] ],  
  ['Chicago', ['Los Angeles', 2015], ['New York', 791], ['San Francisco', 2131] ],  
  ['Los Angeles', ['New York', 2790], ['San Francisco', 381] ],  
  ['New York', ['San Francisco', 2901] ] ]
```

Finally, for a chess-playing program, the location and identification of each chess piece needs to be represented (Figure 1-24). An obvious way to do this is shown on the left below, in which each piece is represented by a single letter ('K' for the king, 'Q' for the queen, 'N' for the knight, etc.),

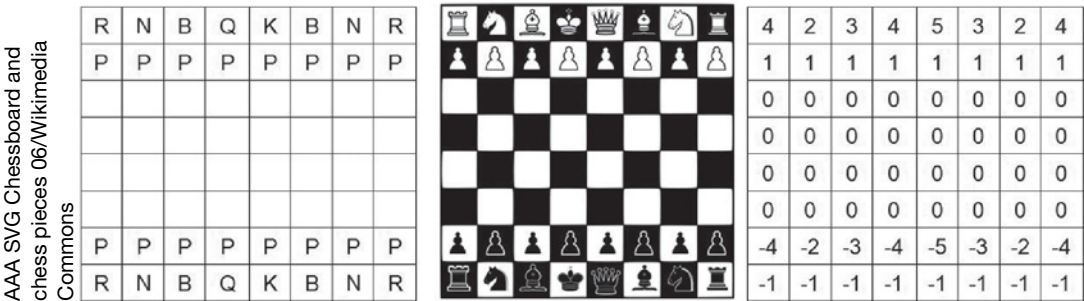


FIGURE 1-24 Representations of Pieces on a Chess Board

There is a problem with this choice of symbols, however—there is no way to distinguish the white pieces from the black ones. The letters could be modified, for example, PB for a black pawn and PW for a white pawn. While that may be an intuitive representation, it is not the best representation for a program. A better way would be to represent pieces using positive and negative integers as shown on the right of the figure: 1 for a white pawn and -1 for a black pawn; 2 for a white bishop

and -2 for a black bishop, and so forth. Various ways of representing chess boards have been developed, each with certain advantages and disadvantages. The appropriate representation of data is a fundamental aspect of computer science.

Describing the Needed Algorithms

When solving a computational problem, either suitable existing algorithms may be found or new algorithms must be developed. For the MCGW problem, there are standard search algorithms that can be used. For the calendar month problem, a day of the week algorithm already exists. For the Traveling Salesman problem, there are various (nontrivial) algorithms that can be utilized, as mentioned, for solving problems with tens of thousands of cities. Finally, for the game of chess, since it is infeasible to look ahead at the final outcomes of every possible move, there are algorithms that make a best guess at which moves to make. Algorithms that work well in general but are not guaranteed to give the correct result for each specific problem are called *heuristic algorithms*.

1.5.3 Program Implementation

Design decisions provide *general* details of the data representation and the algorithmic approaches for solving a problem. The details, however, do not specify which programming language to use, or how to implement the program. That is a decision for the implementation phase. Since we are programming in Python, the implementation needs to be expressed in a syntactically correct and appropriate way, using the instructions and features available in Python.

1.5.4 Program Testing

As humans, we have abilities that far exceed the capabilities of any machine, such as using commonsense reasoning, or reading the expressions of another person. However, one thing that we are not very good at is dealing with detail, which computer programming demands. Therefore, while we are enticed by the existence of increasingly capable computing devices that unfailingly and speedily execute whatever instructions we give them, writing computer programs is difficult and challenging. As a result, *programming errors are pervasive, persistent and inevitable*. However, the sense of accomplishment of developing software that can be of benefit to hundreds, thousands, or even millions of people can be extremely gratifying. If it were easy to do, the satisfaction would not be as great.

Given this fact, software testing is a crucial part of software development. Testing is done incrementally as a program is being developed, when the program is complete, and when the program needs to be updated. In subsequent chapters, program testing and debugging will be discussed and expanded upon. For now, we provide the following general truisms of software development in Figure 1-25.

1. Programming errors are pervasive, persistent, and inevitable.
2. Software testing is an essential part of software development.
3. Any changes made in correcting a programming error should be fully understood as to why the changes correct the detected error.

FIGURE 1-25 Truisms of Software Development

Truism 1 reflects the fact that programming errors are inevitable and that we must accept it. As a result of truism 1, truism 2 states the essential role of software testing. Given the inevitability of programming errors, it is important to test a piece of software in a thorough and systematic manner. Finally, truism 3 states the importance of understanding *why* a given change (or set of changes) in a program fixes a specific error. If you make a change to a program that fixes a given problem but you don't know why it did, then you have lost control of the program logic. As a result, you may have corrected one problem, but inadvertently caused other, potentially more serious ones.

Accountants are committed to reconciling balances to the penny. They do not disregard a discrepancy of one cent, for example, even though the difference between the calculated and expected balances is so small. They understand that a small, seemingly insignificant difference can be the result of two (or more) very big discrepancies. For example, there may be an erroneous credit of \$1,500.01 and an erroneous debit of \$1,500. (The author has experienced such a situation in which the people involved worked all night to find the source of the error.) Determining the source of errors in a program is very much the same. We next look at the Python programming language.

1.6 The Python Programming Language

Now that computational problem solving and computer programming have been discussed, we turn to the Python programming language and associated tools to begin putting this knowledge into practice.

1.6.1 About Python

Guido van Rossum (Figure 1-26) is the creator of the Python programming language, first released in the early 1990s. Its name comes from a 1970s British comedy sketch television show called *Monty Python's Flying Circus*. (Check them out on YouTube!) The development environment IDLE provided with Python (discussed below) comes from the name of a member of the comic group.

Python has a simple syntax. Python programs are clear and easy to read. At the same time, Python provides powerful programming features, and is widely used. Companies and organizations that use Python include YouTube, Google, Yahoo, and NASA. Python is well supported and freely available at www.python.org. (See the Python 3 Programmers' Reference at the end of the text for how to download and install Python.)

1.6.2 The IDLE Python Development Environment

IDLE is an **integrated development environment (IDE)**. An IDE is a bundled set of software tools for program development. This typically includes



Jason E. Kaplan

FIGURE 1-26 Guido van Rossum

an **editor** for creating and modifying programs, a **translator** for executing programs, and a **program debugger**. A debugger provides a means of taking control of the execution of a program to aid in finding program errors.

Python is most commonly translated by use of an interpreter. Thus, Python provides the very useful ability to execute in interactive mode. The window that provides this interaction is referred to as the **Python shell**. Interacting with the shell is much like using a calculator, except that, instead of being limited to the operations built into a calculator (addition, subtraction, etc.), it allows the entry and creation of any Python code. Example use of the Python shell is demonstrated in Figure 1-27.

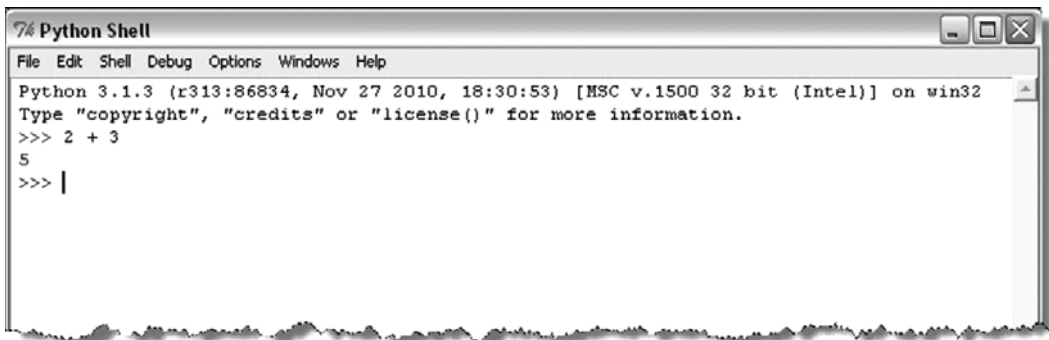


FIGURE 1-27 Python Shell

Here, the expression `2 + 3` is entered at the **shell prompt** (`>>>`), which immediately responds with the result 5.

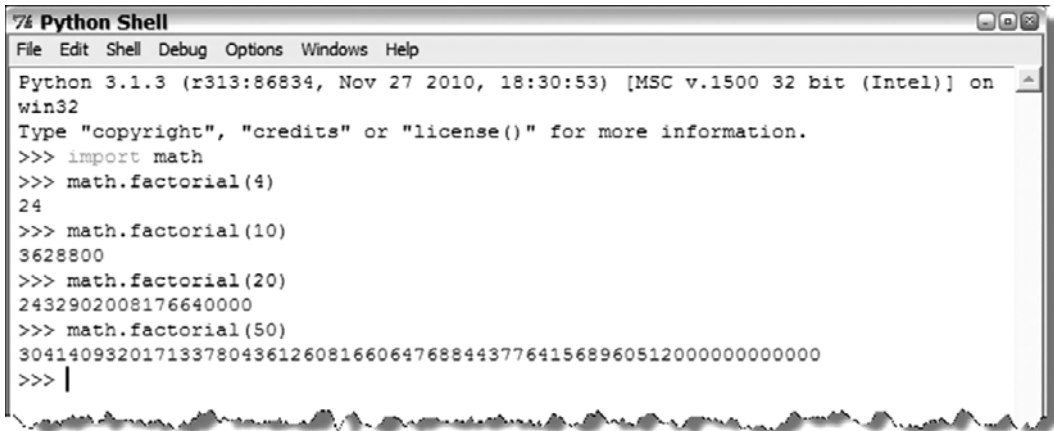
Although working in the Python shell is convenient, the entered code is not saved. Thus, for program development, a means of entering, editing, and saving Python programs is provided by the program editor in IDLE. Details are given below.

An **Integrated Development Environment (IDE)** is a bundled set of software tools for program development.

1.6.3 The Python Standard Library

The **Python Standard Library** is a collection of *built-in modules*, each providing specific functionality beyond what is included in the “core” part of Python. (We discuss the creation of Python modules in Chapter 7.) For example, the `math` module provides additional mathematical functions. The `random` module provides the ability to generate random numbers, useful in programming, as we shall see. (Other Python modules are described in the Python 3 Programmers’ Reference.) In order to utilize the capabilities of a given module in a specific program, an **import** statement is used as shown in Figure 1-28.

The example in the figure shows the use of the `import math` statement to gain access to a particular function in the `math` module, the `factorial` function. The syntax for using the `factorial`



```

Python 3.1.3 (x313:86834, Nov 27 2010, 18:30:53) [MSC v.1500 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>> math.factorial(4)
24
>>> math.factorial(10)
3628800
>>> math.factorial(20)
2432902008176640000
>>> math.factorial(50)
304140932017133780436126081660647688443776415689605120000000000000
>>> |

```

FIGURE 1-28 Using an import statement

function is `math.factorial(n)`, for some positive integer n . We will make use of library modules in Chapter 2. In section 1.7, we see how to enter and execute a complete Python program.

The **Python Standard Library** is a collection of *modules*, each providing specific functionality beyond what is included in the core part of Python.

1.6.4 A Bit of Python

We introduce a bit of Python, just enough to begin writing some simple programs. Since all computer programs input data, process the data, and output results, we look at the notion of a variable, how to perform some simple arithmetic calculations, and how to do simple input and output.

Variables

One of the most fundamental concepts in programming is that of a *variable*. (Variables are discussed in detail in Chapter 2.) A simple description of a variable is “a name that is assigned to a value,” as shown below,

`n = 5` variable `n` is assigned the value 5

Thus, whenever variable `n` appears in a calculation, it is the current value that `n` is assigned to that is used, as in the following,

`n + 20` (`5 + 20`)

If variable `n` is assigned a new value, then the same expression will produce a different result,

`n = 10`
`n + 20` (`10 + 20`)

We next look at some basic arithmetic operators of Python.

A **variable** is “a name that is assigned to a value.”

Some Basic Arithmetic Operators

The common arithmetic operators in Python are + (addition), − (subtraction), * (multiplication), / (division), and ** (exponentiation). Addition, subtraction, and division use the same symbols as standard mathematical notation,

10 + 20 25 − 15 20 / 10

(There is also the symbol // for truncated division, discussed in Chapter 2.) For multiplication and exponentiation, the asterisk (*) is used.

5 * 10 (5 times 10) 2 ** 4 (2 to the 4th power)

Multiplication is never denoted by the use of parentheses as in mathematics, as depicted below,

10 * (20 + 5) CORRECT 10(20 + 5) INCORRECT

Note that parentheses may be used to denote subexpressions. Finally, we see how to input information from the user, and display program results.

The common arithmetic operators in Python are + (addition), − (subtraction), * (multiplication), / (division), and ** (exponentiation).

Basic Input and Output

The programs that we will write request and get information from the user. In Python, the `input` function is used for this purpose,

```
name = input('What is your name?: ')
```

Characters within quotes are called *strings*. This particular use of a string, for requesting input from the user, is called a *prompt*. The `input` function displays the string on the screen to prompt the user for input,

```
What is your name?: Charles
```

The underline is used here to indicate the user’s input.

The `print` function is used to display information on the screen in Python. This may be used to display a message,

```
>>> print('Welcome to My First Program!')
Welcome to My First Program!
```

or used to output the value of a variable,

```
>>> n = 10
>>> print(n)
10
```

or to display a combination of both strings and variables,

```
>>> name = input('What is your name?: ')
What is your name?: Charles
>>> print('Hello', name)
Hello Charles
```

Note that a comma is used to separate the individual items being printed, causing a space to appear between each when displayed. Thus, the output of the print function in this case is Hello Charles, and not HelloCharles. There is more to say about variables, operators, and input/output in Python. This will be covered in the chapters ahead.

In Python, `input` is used to request and get information from the user, and `print` is used to display information on the screen.

1.6.5 Learning How to Use IDLE

In order to become familiar with writing your own Python programs using IDLE, we will create a simple program that asks the user for their name and responds with a greeting. This program utilizes the following concepts:

- ◆ creating and executing Python programs
- ◆ `input` and `print`



First, to create a Python program file, select New Window from the File menu in the Python shell as shown in Figure 1-29:

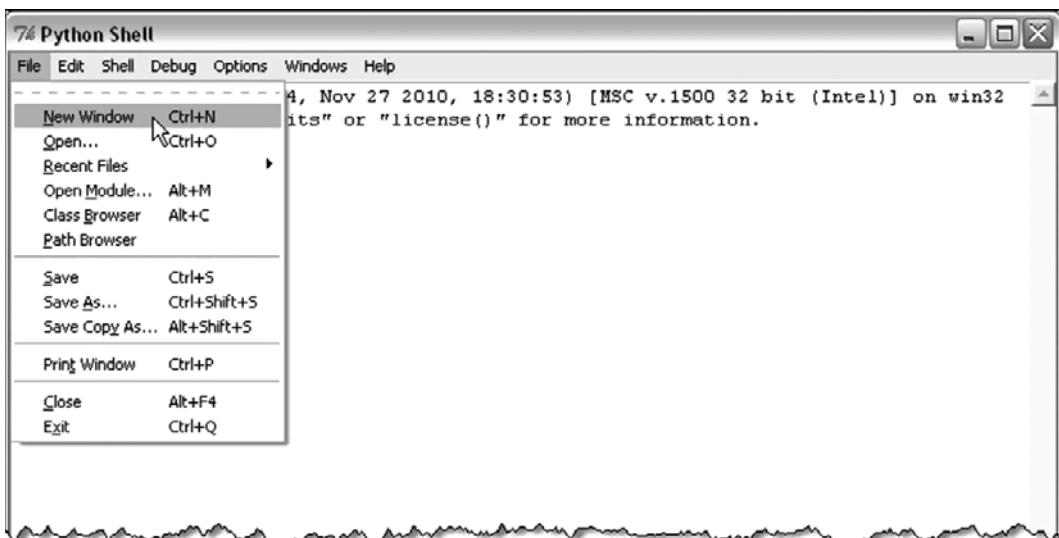
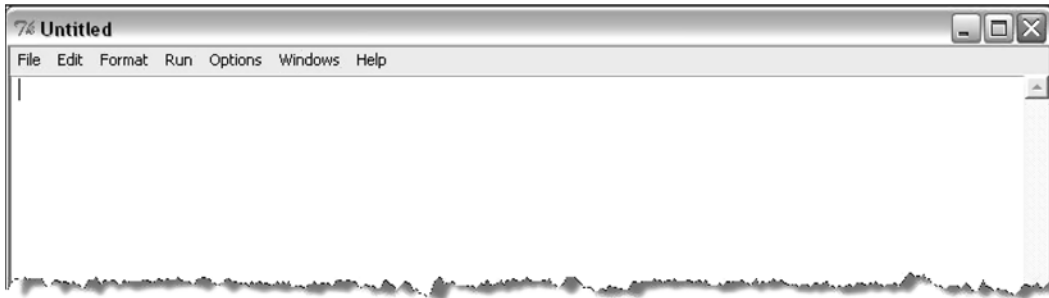


FIGURE 1-29 Creating a Python Program File

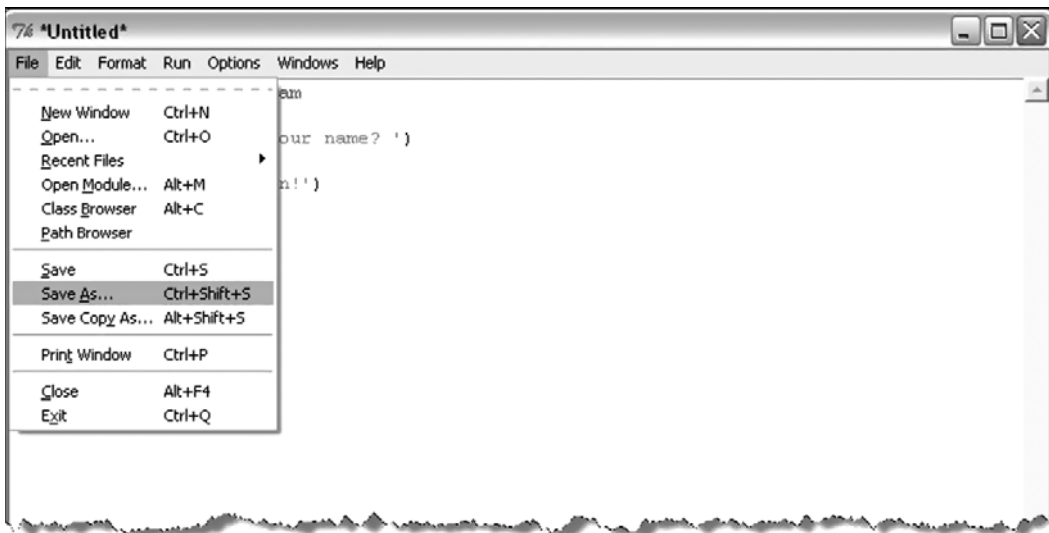
A new, untitled program window will appear:



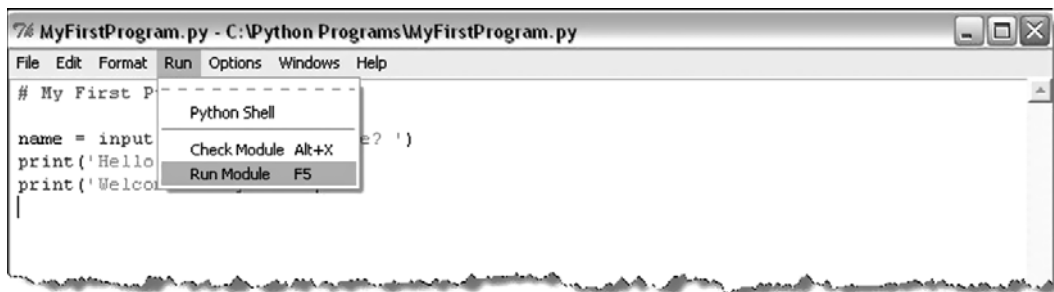
Type the following in the program window exactly as shown.



When finished, save the program file by selecting Save As under the File menu, and save in the appropriate folder with the name `MyFirstProgram.py`.



To run the program, select Run Module from the Run menu (or simply hit function key F5).



If you have entered the program code correctly, the program should execute as shown in Figure 1-30.

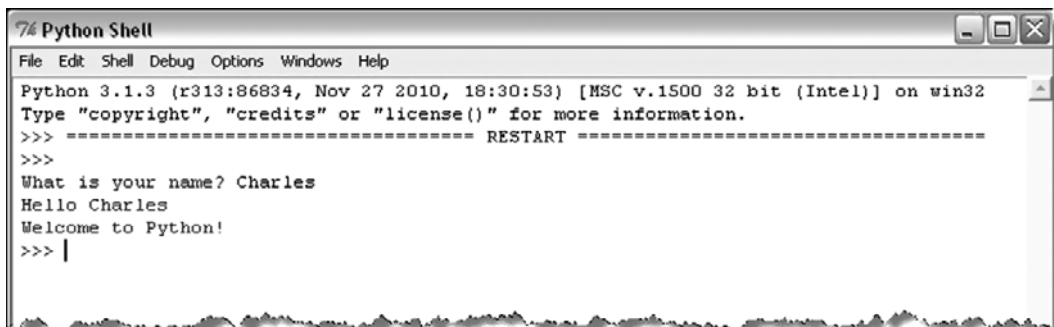


FIGURE 1-30 Sample Output of MyFirstProgram.py

If, however, you have mistyped part of the program resulting in a syntax error (such as mistyping print), you will get an error message similar to that in Figure 1-31.

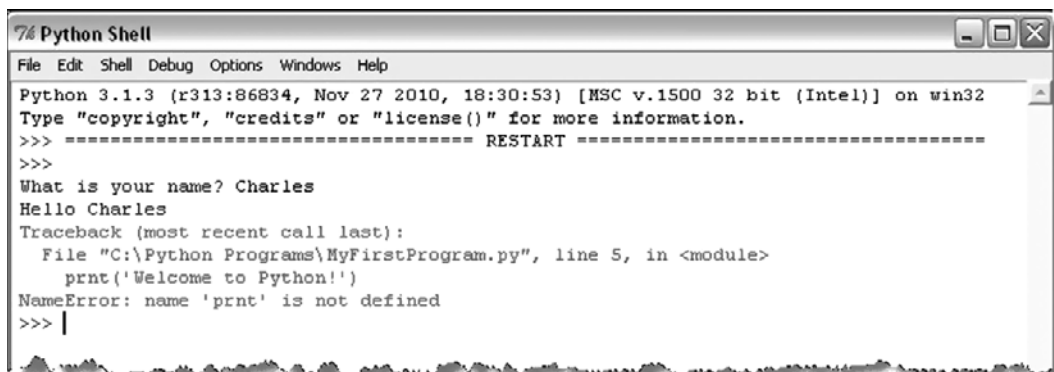


FIGURE 1-31 Output Resulting from a Syntax Error

In that case, go back to the program window and make the needed corrections, then re-save and re-execute the program. You may need to go through this process a number of times until all the syntax errors have been corrected.

1.7 A First Program—Calculating the Drake Equation

Dr. Frank Drake conducted the first search for radio signals from extraterrestrial civilizations in 1960. This established SETI (Search for Extraterrestrial Intelligence), a new area of scientific inquiry. In order to estimate the number of civilizations that may exist in our galaxy that we may be able to communicate with, he developed what is now called the *Drake equation*.

The Drake equation accounts for a number of different factors. The values used for some of these are the result of scientific study, while others are only the result of an “intelligent guess.” The factors consist of *R*, the average rate of star creation per year in our galaxy; *p*, the percentage of those stars that have planets; *n*, the average number of planets that can potentially support life for each star with planets; *f*, the percentage of those planets that actually go on to develop life; *i*, the percentage of those planets that go on to develop intelligent life; *c*, the percentage of those that have the technology communicate with us; and *L*, the expected lifetime of civilizations (the period that they can communicate). The Drake equation is simply the multiplication of all these factors, giving *N*, the estimated number of detectable civilizations there are at any given time,

$$N = R \cdot p \cdot n \cdot f \cdot i \cdot c \cdot L$$

Figure 1-32 shows those parameters in the Drake equation that have some consensus as to their correct value.



Vincdevries/LittleGreenMen/Wikimedia Commons

Drake Equation Factor Values		Estimated Values
Rate of star creation	<i>R</i>	7 [†]
Percentage of stars with planets	<i>p</i>	40%
Average number of planets that can potentially support life for each star with planets	<i>n</i>	(no consensus)
Percentage of those that go on to develop life	<i>f</i>	13%
Percentage of those that go on to intelligent develop life	<i>i</i>	(no consensus)
Percentage of those willing and able to communicate	<i>c</i>	(no consensus)
Expected lifetime of civilizations	<i>L</i>	(no consensus)
[†] Estimate of NASA and the European Space Agency		

FIGURE 1-32 Proposed Values for the Drake Equation

1.7.1 The Problem

The value of 7 for R , the rate of star creation, is the least disputed value in the Drake equation today. Given the uncertainty of the remaining factors, you are to develop a program that allows a user to enter their own estimated values for the remaining six factors (p , n , f , i , c , and L) and displays the calculated result.

1.7.2 Problem Analysis

This problem is very straightforward. We only need to understand the equation provided.

1.7.3 Program Design

The program design for this problem is also straightforward. The data to be represented consist of numerical values, with the Drake equation as the algorithm. The overall steps of the program are depicted in Figure 1-33.

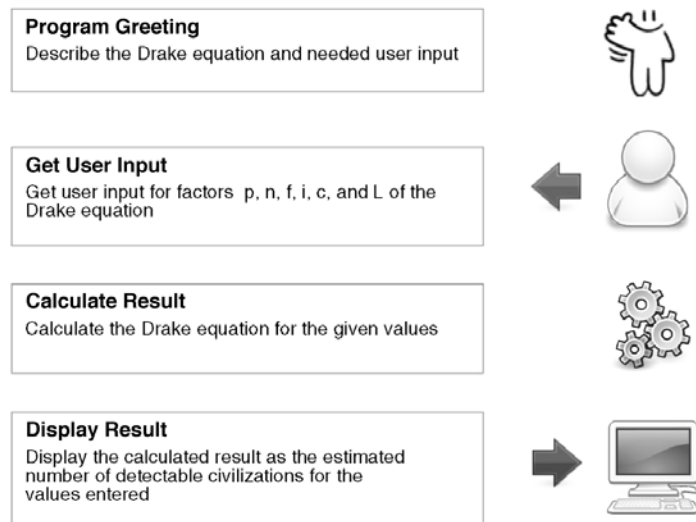


FIGURE 1-33 The Overall Steps of the Drake Equation Program

1.7.4 Program Implementation

The implementation of this program is fairly simple. The only programming elements needed are input, assignment, and print, along with the use of arithmetic operators. An implementation is given in Figure 1-34. Example execution of the program is given in Figure 1-35.

First, note the program lines beginning with the hash sign, `#`. In Python, this symbol is used to denote a *comment statement*. A **comment statement** contains information for persons reading the program. Comment statements are ignored during program execution—they have no effect on the program results. In this program, the initial series of comment statements (**lines 1–23**) explain the Drake equation and provide a brief summary of the purpose of the program.

```

1 # SETI Program
2 #
3 # The Drake equation, developed by Frank Drake in the 1960s, attempts to
4 # estimate how many extraterrestrial civilizations, N, may exist in our
5 # galaxy at any given time that we might come in contact with,
6 #
7 #     N = R * p * n * f * i * c * L
8 #
9 # where,
10 #
11 #     R ... estimated rate of star creation in our galaxy
12 #     p ... estimated percent of stars that have planets
13 #     n ... estimated average number of planets that can potentially support
14 #         life for each star with planets
15 #     f ... estimated percent of those planets that actually go on to develop life
16 #     i ... estimated percent of those planets go on to develop intelligent life
17 #     c ... estimated percent of those that are willing and able to communicate
18 #     L ... estimated expected lifetime of such civilizations
19 #
20 # Given that the value for R, 7 per year, is the least disputed of the values,
21 # the user will be prompted to enter estimated values for the remaining six
22 # factors. The estimated number of civilizations that may be detected in our
23 # galaxy will then be displayed.
24
25 # display program welcome
26 print('Welcome to the SETI program')
27 print('This program will allow you to enter specific values related to')
28 print('the likelihood of finding intelligent life in our galaxy. All')
29 print('percentages should be entered as integer values, e.g., 40 and not .40')
30 print()
31
32 # get user input
33 p = int(input('What percentage of stars do you think have planets?: '))
34 n = int(input('How many planets per star do you think can support life?: '))
35 f = int(input('What percentage do you think actually develop life?: '))
36 i = int(input('What percentage of those do you think have intelligent life?: '))
37 c = int(input('What percentage of those do you think can communicate with us?: '))
38 L = int(input('Number of years you think civilizations last?: '))
39
40 # calculate result
41 num_detectable_civilizations = 7 * (p/100) * n * (f/100) * (i/100) * (c/100) * L
42
43 # display result
44 print()
45 print('Based on the values entered ...')
46 print('there are an estimated', round(num_detectable_civilizations),
47       'potentially detectable civilizations in our galaxy')

```

FIGURE 1-34 Drake Equation Program

Comment statements are also used in the program to denote the beginning of each program section (**lines 25, 32, 40, and 43**). These *section headers* provide a broad outline of the program following the program design depicted in Figure 1-33.

The program welcome section (**lines 25–30**) contains a series of print instructions displaying output to the screen. Each begins with the word `print` followed by a matching pair of parentheses. Within the parentheses are the items to be displayed. In this case, each contains a particular string of characters. The final “empty” print, `print()` on **line 30** (and **line 44**), does not display anything. It simply causes the screen cursor to move down to the next line, therefore creating a skipped line in the screen output. (Later we will see another way of creating the same result.)

```

Program Execution ...

Welcome to the SETI program
This program will allow you to enter specific values related to
the likelihood of finding intelligent life in our galaxy. All
percentages should be entered as integer values, e.g., 40 and not .40

What percentage of stars do you think have planets?: 40
How many planets per star do you think can support life?: 2
What percentage do you think actually develop life?: 5
What percentage of those do you think have intelligent life?: 3
What percentage of those do you think can communicate with us?: 5
Number of years you think civilizations last?: 10000

Based on the values entered...
there are an estimated 4.2 potentially detectable civilizations in
our galaxy
>>>

```

FIGURE 1-35 Execution of the Drake Equation Program

The following section (**lines 32–38**) contains the instructions for requesting the input from the user. Previously, we saw the `input` function used for inputting a name entered by the user. In that case, the instruction was of the form,

```
name = input('What is your name?:')
```

In this program, there is added syntax,

```
p = int(input('What percentage of stars do you think have planets?:'))
```

The `input` function always returns what the user enters as a string of characters. This is appropriate when a person's name is being entered. However, in this program, numbers are being entered, not names. Thus, in, the following,

```
What percentage of stars do you think have planets?: 40
```

40 is to be read as single number and not as the characters '4' and '0'. The addition of the `int(...)` syntax around the `input` instruction accomplishes this. This will be discussed more fully when numeric and string (character) data are discussed in Chapter 2.

On **line 41** the Drake equation is calculated and stored in variable `num_detectable_civilizations`. Note that since some of the input values were meant to be percentages (`p`, `f`, `i`, and `c`), those values in the equation are each divided by 100. Finally, **lines 44–47** display the results.

1.7.5 Program Testing

To test the program, we can calculate the Drake equation for various other values using a calculator, providing a set of *test cases*. A **test case** is a set of input values and expected output of a given program. A **test plan** consists of a number of test cases to verify that a program meets all requirements. A good strategy is to include “average,” as well as “extreme” or “special” cases in a test plan. Such a test plan is given in Figure 1-36.

Based on these results, we can be fairly confident that the program will give the correct results for all input values.

Input Values							Expected Results	Actual Results	Evaluation
p	n	f	i	c	L				
Extreme Cases (no chance of contacting intelligent life)									
Zero Planets per Star	0	2	100%	1%	1%	10,000	0	0	passed
Zero percent of Planets Support Life	50%	0	100%	1%	1%	10,000	0	0	passed
Average Cases									
Average Case 1	30%	3	75%	1%	5%	5,000	12	12	passed
Average Case 2	50%	6	80%	5%	10%	10,000	840	840	passed
Extreme Cases (great chance of contacting intelligent life)									
Extreme Case 1	100%	10	100%	100%	100%	10,000	700,000	700,000	passed
Extreme Case 2	100%	12	100%	100%	100%	100,000	8,400,000	8,400,000	passed

FIGURE 1-36 Test Plan Results for Drake's Equation Program

CHAPTER SUMMARY

General Topics

Computational Problem Solving/Representation/
 Abstraction
 Algorithms/Brute Force Approach
 Computer Hardware/Transistors/Integrated Circuits/
 Moore's Law
 Binary Representation/Bit/Byte/
 Binary-Decimal Conversion
 Central Processing Unit (CPU)/Main and
 Secondary Memory
 Input/Output Devices/Buses
 Computer Software/System Software/
 Application Software
 Operating Systems
 Syntax and Semantics/Program Translation/
 Compiler vs. Interpreter

Program Debugging/Syntax Errors vs. Logic
 (Semantic) Errors
 Procedural Programming vs. Object-Oriented
 Programming
 The Process of Computational Problem Solving/
 Program Testing
 Integrated Development Environments (IDE)/
 Program Editors/Debuggers
 Comment Statements as Program Documentation
 Test Cases/Test Plans

Python-Specific Programming Topics

The Python Programming Language/
 Guido van Rossum (creator of Python)
 Comment Statements in Python

Introduction to Variables, Arithmetic Operators,
 and `print` in Python
 Introduction to Strings with the `input` Function
 in Python
 Introduction to the Python Standard Library and
 the `import` Statement

The Python Shell/The IDLE Integrated
 Development Environment
 The Standard Python Library

CHAPTER EXERCISES

Section 1.1

1. Search online for two computing-related fields named “computational X” other than the ones listed in Figure 1-1.
2. Search online for two areas of computer science other than those given in the chapter.
3. For the Man, Cabbage, Goat, Wolf problem:
 - (a) List all the invalid states for this problem, that is, in which the goat is left alone with the cabbage, or the wolf is left alone with the goat.
 - (b) Give the shortest sequence of steps that solves the MCGW problem.
 - (c) Give the sequence of state representations that correspond to your solution starting with (E,E,E,E) and ending with (W,W,W,W).
 - (d) There is an alternate means of representing states. Rather than a sequence representation, a set representation can be used. In this representation, if an item is on the east side of the river, its symbol is in the set, and if on the west side, the symbol is not in the set as shown below,

`{M,C,G,W}`—all items on east side of river (start state)

`{C,W}`—cabbage and wolf on east side of river, man and goat on west side

`{ }`—all items on the west side of the river (goal state)

Give the sequence of states for your solution to the problem using this new state representation.

- (e) How many shortest solutions are there for this problem?
4. For a simple game that starts with five stones, in which each player can pick up either one or two stones, the person picking up the last stone being the loser,
 - (a) Give a state representation appropriate for this problem.
 - (b) Give the start state and goal state for this problem.
 - (c) Give a sequence of states in which the first player wins the game.

Section 1.2

5. Using the algorithm in Figure 1-8, *show all steps* for determining the day of the week for January 24, 2018. (Note that 2018 is not a leap year.)
6. Using the algorithm in Figure 1-8, determine the day of the week that you were born on.
7. Suppose that an algorithm was needed for determining the day of the week for dates that only occur within the years 2000–2099. Simplify the day of the week algorithm in Figure 1-8 as much as possible by making the appropriate changes.
8. As precisely as possible, give a series of steps (an algorithm) for doing long addition.

Section 1.3

9. What is the number of bits in 8 bytes, assuming the usual number of bits in a byte?
10. Convert the following values in binary representation to base 10. *Show all steps.*
 - (a) 1010 (b) 1011 (c) 10000 (d) 1111

11. Convert the following values into binary (base 2) representation. *Show all steps.*
- (a) 5 (b) 7 (c) 16 (d) 15 (e) 32
 (f) 33 (g) 64 (h) 63 (i) 128 (j) 127
12. What is in common within each of the following groups of binary numbers?
- (a) values that end with a “0” digit (e.g., 1100)
 (b) values that end with a “1” digit (e.g., 1101)
 (c) values with a leftmost digit of “1,” followed by all “0s” (e.g., 1000)
 (d) values consisting only of all “1” digits (e.g., 1111)
13. Assuming that Moore’s Law continues to hold true, where n is the number of transistors that can currently be placed on an integrated circuit (chip), and $k \cdot n$ is the number that can be placed on a chip in eight years, what is the value of k ?

Section 1.4

14. Give two specific examples of an application program besides those mentioned in the chapter.
15. For each of the following statements in English, indicate whether the statement contains a syntax error, a logic (semantic) error, or is a valid statement.
- (a) Witch way did he go?
 (b) I think he went over their.
 (c) I didn’t see him go nowhere.
16. For each of the following arithmetic expressions for adding up the integers 1 to 5, indicate whether the expression contains a syntax error, a semantic error, or is a valid expression.
- (a) $1 + 2 ++ 3 + 4 + 5$
 (b) $1 + 2 + 4 + 5$
 (c) $1 + 2 + 3 + 4 + 5$
 (d) $5 + 4 + 3 + 2 + 1$
17. Give one benefit of the use of a compiler, and one benefit of the use of an interpreter.

Section 1.5

18. Use the Python Interactive Shell to calculate the number of routes that can be taken for the Traveling Salesman problem for:
- (a) 6 cities (b) 12 cities (c) 18 cities (d) 36 cities
19. Enter the following statement into the interactive shell:

```
print('What is your favorite color?')
```

Record the output. Now enter the following statement exactly as given,

```
printt('What is your favorite color?')
```

Record the output. Is this a syntax error or a logic error?

20. For the Traveling Salesman problem,
- (a) Update the list representation of the distances between cities in the table in Figure 1-23 to add the city of Seattle. The distances between Seattle and each of the other cities is given below.
- Atlanta to Seattle, 2641 miles, Boston to Seattle, 3032 miles, Chicago to Seattle, 2043 miles, LA to Seattle, 1208 miles, NYC to Seattle, 2832 miles, San Francisco to Seattle, 808 miles
- (b) Determine a reasonably short route of travel for visiting each city once and only once, starting in Atlanta and ending in San Francisco.

Section 1.6

21. Which of the following capabilities does an integrated development environment (IDE) provide?
 - (a) Creating and modifying programs
 - (b) Executing programs
 - (c) Debugging programs
 - (d) All of the above
22. The Python shell is a window in which Python instructions are immediately executed. (TRUE/FALSE)
23. Suppose that the `math` module of the Python Standard Library were imported. What would be the proper syntax for calling a function in the `math` module named `sqrt` to calculate the square root of four?
24. What is the value of variable `n` after the following instructions are executed?

```
j = 5
k = 10
n = j * k
```

25. Which of the following is a proper arithmetic expression in Python?
 - (a) `10(15 + 6)`
 - (b) `(10 * 2)(4 + 8)`
 - (c) `5 * (6 - 2)`
26. Exactly what is output by the following if the user enters 24 in response to the input prompt.

```
age = input('How old are you?: ')
print('You are', age, 'years old')
```

PYTHON PROGRAMMING EXERCISES

- P1. Write a simple Python program that displays the following powers of 2, one per line: 2^1 , 2^2 , 2^3 , 2^4 , 2^5 , 2^6 , 2^7 , 2^8 .
- P2. Write a Python program that allows the user to enter any integer value, and displays the value of 2 raised to that power. Your program should function as shown below.

```
What power of two? 10
Two to the power of 10 is 1024
```

- P3. Write a Python program that allows the user to enter any integer base and integer exponent, and displays the value of the base raised to that exponent. Your program should function as shown below.

```
What base? 10
What power of 10? 4
10 to the power of 4 is 10000
```

- P4. Write a Python program that allows the user to enter a four-digit binary number and displays its value in base 10. *Each binary digit should be entered one per line, starting with the leftmost digit*, as shown below.

```
Enter leftmost digit: 1
Enter the next digit: 0
Enter the next digit: 0
Enter the next digit: 1
The value is 9
```

- P5.** Write a simple Python program that prompts the user for a certain number of cities for the Traveling Salesman problem, and displays the total number of possible routes that can be taken. Your program should function as shown below.

```
How many cities? 10
For 10 cities, there are 3628800 possible routes
```

PROGRAM MODIFICATION PROBLEMS

- M1.** Modify the sample “hello” Python program in section 1.6.5 to first request the user’s first name, and then request their last name. The program should then display,

```
Hello firstname lastname
Welcome to Python!
```

for the *firstname* and *lastname* entered.

- M2.** Modify the Drake’s Equation Program in section 1.7 so that it calculates results for a best case scenario, that is, so that factors *p* (percentage of stars that have planets), *f* (percentage of those planets that develop life), *i* (percentage of those planets that develop intelligent life), and *c* (percentage of those planets that can communicate with us) are all hard-coded as 100%. The value of *R* should remain as 7. Design the program so that the only values that the user is prompted for are how many planets per star can support life, *n*, and the estimated number of years civilizations last, *L*. Develop a set of test cases for your program with the included test results.

PROGRAM DEVELOPMENT PROBLEMS

- D1.** Develop and test a program that allows the user to enter an integer value indicating the number of cities to solve for the Traveling Salesman problem. The program should then output the number of years it would take to solve using a brute force-approach. Make use of the factorial function of the math module as shown in Figure 1-28. Estimate the total amount of time it takes by using the assumptions given in section 1.1.2.
- D2.** Based on the information provided about the game of chess in section 1.1.2, develop and test a program that determines how many years it would take for all possible chess games to be played if everyone in the world (regardless of age) played one (unique) chess game a day. Assume the current world population to be 7 billion.