CHAPTER 3

Inside a Computer

One day ladies will take their computers for walks in the park and tell each other, "My little computer said such a funny thing this morning."

Alan Turing
1951

And now for something completely different!

Monty Python's Flying Circus

There are only 10 types of people in the world: those who understand binary, and those who don't.

Unknown

Alan Turing, the father of Computer Science, optimistically predicted that by now computers would be carrying on meaningful conversations with human beings. While this has not yet come to pass, by executing clever algorithms very fast, computers can sometimes *appear* to exhibit primitive intelligence. In a historic example, in 1997, the IBM Deep Blue computer defeated reigning world champion Garry Kasparov in a six-game match of chess. In 2011, IBM's Watson computer beat two champions in the television game show Jeopardy. Five years later, the AlphaGo program defeated the world Go champion over a five match series. And now we are on the cusp of seeing autonomous vehicles almost entirely replacing human drivers.

The intelligence that we sometimes attribute to computers, however, is actually human intelligence that was originally expressed as an algorithm, and then as a program. Contemporary computers at their cores remain dumb machines. Even the

Copyright Taylor and Francis, 2021

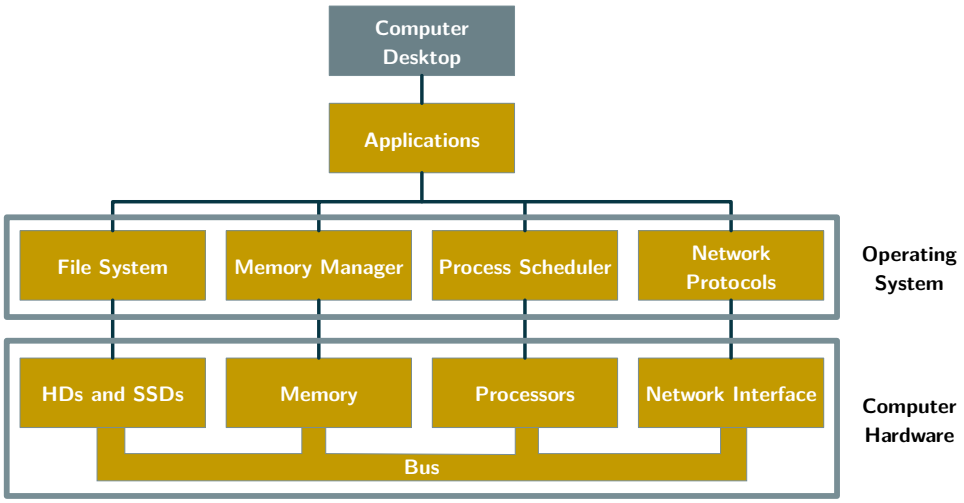


Figure 3.1 A simplified look at layers of abstraction when using a computer.

statements in a high-level programming language are themselves abstract conveniences built upon a much more rudimentary set of instructions that a computer can execute natively.

3.1 COMPUTERS ARE DUMB

When you use a computer, you are utilizing layers of functional abstractions. As illustrated in Figure 3.1, at the highest layer, you are presented with a "desktop" abstraction on which you can store files and use applications (or programs) to do work. That there appear to be many applications active simultaneously on this desktop is also an abstraction. In reality, *some* applications may be working in parallel while others are not, but the computer is alternating among them so quickly that they appear to be working in parallel. Each of these applications is a sequence of ***machine language*** instructions that can be executed by the computer hardware. Some of these instructions rely on various functional abstractions provided by the computer's ***operating system*** to save files, access the computer's memory, retrieve information from the Internet, etc. In other words, the operating system provides functional abstractions that allow us, via the applications we use, to more conveniently and efficiently use the computer's resources.

Since machine language is the only thing that a computer "understands," every statement in a Python program must be translated by the Python interpreter into a sequence of equivalent machine language instructions before it can be executed. An interpreter translates one line of a high-level program into machine language, executes it, then translates the next line and executes it, etc. Other languages (such as C++) use a ***compiler*** instead. A compiler translates a high-level language program all at once into machine language. Then the compiled machine language

Copyright Taylor and Francis, 2021

Tangent 3.1: High performance computing

Although today's computers are extremely fast, there are some problems that are so big that additional power is necessary. These include weather forecasting, molecular modeling and drug design, aerodynamics, and deciphering encrypted data. For these problems, scientists use *supercomputers*. A supercomputer, or *high performance computer*, is made of up to tens of thousands of *nodes*, connected by a very fast data network that shuttles information between nodes. A node, essentially a standalone computer, can contain multiple processors, each with multiple cores. The fastest supercomputers today have millions of cores and a million gigabytes of memory.

To realize the power of these computers, programmers need to supply their cores with a constant stream of data and instructions. The results computed by the individual cores are then aggregated into an overall solution. The algorithm design and programming techniques, known as ***parallel programming***, are beyond the scope of this book, but we provide additional resources at the end of the chapter if you would like to learn more.

program can be executed from start to finish without additional translation. This tends to make compiled programs faster than interpreted ones. However, interpreted languages allow us to more closely interact with a program during its execution.

The types of instructions that constitute a machine language are based on the internal design of a modern computer. As illustrated in Figure 3.1, a computer essentially consists of one or more *processors* connected to a ***memory***. A computer's memory, often called *RAM* (short for *random access memory*), is conceptually organized as a long sequence of ***cells***, each of which can contain one unit of information. Each cell is labeled with a unique ***memory address*** that allows the processor to reference it specifically. So a computer's memory is like a huge sequence of equal-sized post office boxes, each of which can hold exactly one letter. Each P.O. box number is analogous to a memory address and a letter is analogous to one unit of information. The information in each cell can represent either one instruction or one unit of data.[1] So a computer's memory stores both programs and the data on which the programs work. A variable name in Python is essentially a reference to a memory address.

A ***processor***, often called a *CPU* (short for *central processing unit*) or *core*, contains both the machinery for executing instructions and a small set of memory locations called *registers* that temporarily hold data values needed by the current instruction. If a computer contains more than one core, as most modern computers do, then it is able to execute more than one instruction at a time. These instructions may be from different programs or from the same program. This means that our previous definition of an algorithm as a *sequence* of instructions is not strictly correct. In fact, an algorithm (or a program) may consist of several semi-independent sequences of steps called *threads* that cooperatively solve a problem.

[1] In reality, each instruction or unit of data usually occupies multiple contiguous cells.

Copyright Taylor and Francis, 2021

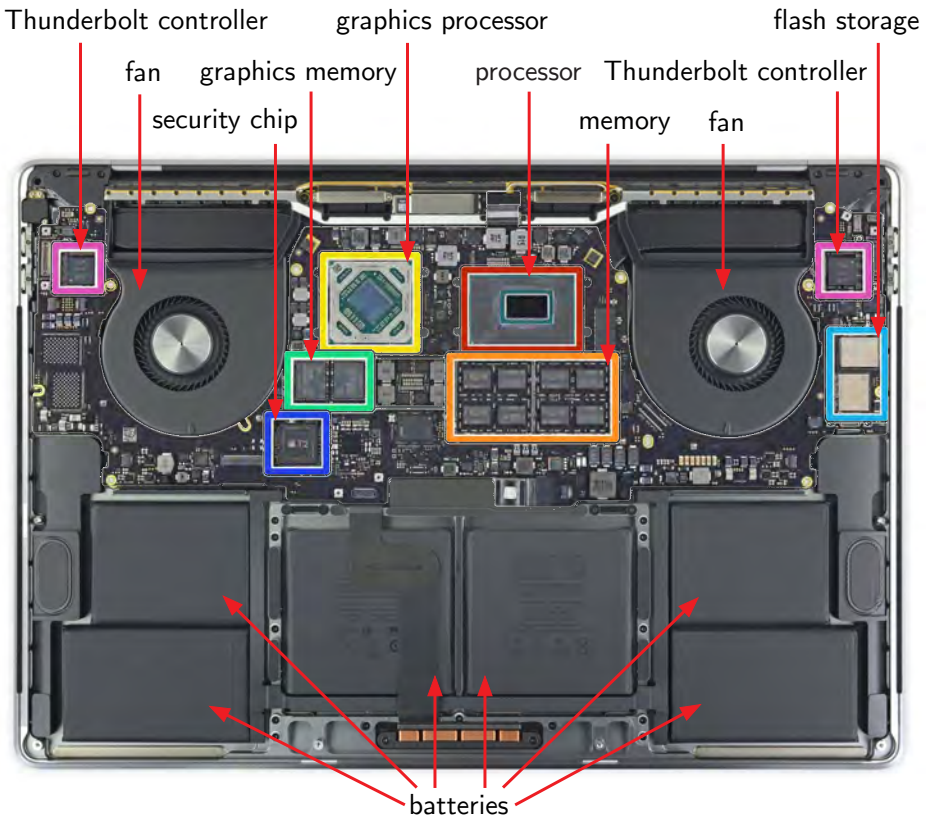


Figure 3.2 Inside an Apple MacBook Pro. Image courtesy of iFixit (`ifixit.com`).

The processors and memory are connected by a communication channel called a ***bus***. When a processor needs a value in memory, it transmits the request over the bus, and then the memory returns the requested value the same way. The bus also connects the processors and memory to several other components that either improve the machine's efficiency or its convenience, like the Internet and ***secondary storage*** devices like hard drives (HD), solid state drives (SSD), and flash memory. As you probably know, the contents of a computer's memory are lost when the computer loses power, so we use secondary storage to preserve data for longer periods of time. We interact with these devices through a "file system" abstraction that makes it appear as if our hard drives are really filing cabinets. When you execute a program or open a file, it is first copied from secondary storage into memory where the processor can access it.

Figure 3.2 shows what these components look like inside a laptop computer. In addition to the processor, memory, and flash storage, which acts as secondary storage, a lot of real estate is occupied by the graphics processor and its dedicated memory, which are responsible for our computers' abilities to display high resolution video and fast-paced video games. The Thunderbolt controllers are responsible for transferring data between the computer and external devices (and the network)

Copyright Taylor and Francis, 2021

connected through Thunderbolt 3 (USB-C) ports. The security chip encrypts data in flash storage, prevents unauthorized software from running, and stores fingerprint data for securely logging in.

Reflection 3.1 *Look up the technical specifications for your computer. On a Mac, select "About This Mac" from the Apple menu. In Windows, search for "System Information" or "msinfo32."*

Machine language

The machine language instructions that a processor can execute are very limited in their abilities.[2] For example, consider something as simple as addition in Python:

```
>>> total = number1 + number2
```

Even something this simple is too complex for one machine language instruction. The machine language equivalent likely, depending on the computer, consists of four instructions that do the following.

1. Load the value in the memory cell referred to by the variable `number1` into a register in the processor.
2. Load the value in the memory cell referred to by the variable `number2` into another register in the processor.
3. Add the values in these two registers and store the result in a third register.
4. Store the value in the third register in the memory cell referred to by the variable `total`.

From the moment a computer is turned on, its processors are operating in a continuous loop called the ***fetch and execute cycle*** (or *machine cycle*). In each cycle, the processor fetches one machine language instruction from memory and executes it. This cycle repeats until the computer is turned off or loses power. This is essentially all a computer does. The rate at which a computer performs the fetch and execute cycle is related to the rate at which its internal clock "ticks" (the processor's *clock rate*). The ticks of this clock keep the machine's operations synchronized. Modern personal computers have clocks that tick a few billion times each second; a 3 gigahertz (GHz) processor ticks 3 billion times per second ("giga" means "billion" and a "hertz" is one tick per second).

So computers, at their most basic level, really are quite dumb; the processor blindly follows the fetch and execute cycle, dutifully executing whatever sequence of simple instructions we give it. The frustrating errors that we yell at computers about are, in fact, human errors. The great thing about computers is not that they are smart, but that they follow our instructions so quickly; they can accomplish an incredible amount of work in a very short amount of time. Whether that work is useful, however, is up to us.

[2]Python programs are actually translated into an intermediate form called byte code first. See Tangent 3.2 for details.

Copyright Taylor and Francis, 2021

Tangent 3.2: Byte code

In Python, the translation process is actually a little more complicated than the more generic discussion in this section. Before they are executed, Python programs are compiled into an intermediate language called ***byte code***. Then the byte code instructions are executed by a *virtual machine* that understands how to translate them into machine language. A virtual machine is a program that emulates a computer.

The Python virtual machine uses a *stack* instead of registers to store references to objects. A stack is conceptually a "pile" of values that can only be accessed from the top. The only operations on a stack are *pushing* a value onto the top of the stack and *popping* the value off the top of the stack. The following sequence of byte code instructions is what is actually produced for `total = number1 + number2` by the Python interpreter.

```
0 LOAD_FAST                0 (number1)
2 LOAD_FAST                1 (number2)
4 BINARY_ADD
6 STORE_FAST               2 (total)
```

The first two instructions push references to the variables `number1` and `number2` onto the stack. The `BINARY_ADD` instruction pops the two values off the stack, adds them together (in binary) and pushes the result onto the top of the stack. The final instruction pops the result off the stack and stores it at the memory location referred to by the variable `total`.

If you would like see byte code for yourself, you can. To see the byte code generated for a function named `myFunction`, do this:

```
import dis
dis.dis(myFunction)
```

3.2 EVERYTHING IS BITS

Our discussion so far has glossed over a very important consideration: in what form does a computer store programs and data? In addition to machine language instructions, we need to store numbers, documents, maps, images, sounds, presentations, spreadsheets, and more. Using a different storage medium for each type of information would be insanely complicated and inefficient. Instead, we need a simple storage format that can accommodate any type of data. The answer is bits. A ***bit*** is the simplest possible unit of information, capable of taking on one of only two values: 0 or 1 (or equivalently, off/on, no/yes, or false/true). Storing data as bits makes perfect sense because bit storage is simple, bits can represent anything, and computing with bits is almost trivial.

Bits are switches

Storing the value of a bit is absurdly simple: a bit is equivalent to an on/off switch, and the value of the bit is equivalent to the state of the switch: off = 0 and on = 1. A computer's memory is essentially composed of billions of microscopic switches,

Copyright Taylor and Francis, 2021

organized into memory cells. Each memory cell contains 8 switches, so each cell can store 8 bits, called a ***byte***. We represent a byte simply as a sequence of 8 bits, such as `01101001`. A computer with 8 gigabytes (GB) of memory contains about 8 billion memory cells, each storing one byte. (Similarly, a kilobyte (KB) is about one thousand bytes, a megabyte (MB) is about one million bytes, and a terabyte (TB) is about one trillion bytes.)

Bits can represent anything

A second advantage of storing information with bits is that, as the simplest possible unit of information, it serves as a "lowest common denominator." All information can be converted to bits in a fairly straightforward manner. Numbers, words, images, music, and machine language instructions are all encoded as sequences of bits before they are stored in memory. Information stored as bits is also said to be in ***binary notation***. For example, consider the following sequence of 16 bits (or two bytes).

`0100010001010101`

This bit sequence can represent each of the following, depending upon how it is interpreted:

(a) the integer value 17,493;

(b) the decimal value 2.166015625;

(c) the two characters "DU";

(d) the Intel machine language instruction `inc x` ("increment" `x`); or

(e) the 4×4 black and white image to the right, called a ***bitmap*** (0 represents a white square and 1 represents a black square).

For now, let's just look more closely at how numbers are stored. Integers are represented in a computer using the *binary number system*, which is understood best by analogy to the decimal number system. In decimal, each position in a number has a value that is a power of ten: from right to left, the positional values are $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, etc. The value of a decimal number comes from adding the digits, each first multiplied by the value of its position. So 1,831 represents the value

$$1 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 1 \times 10^0 = 1000 + 800 + 30 + 1.$$

The binary number system is no different, except that each position represents a power of two, and there are only two digits instead of ten. From right to left, the binary number system positions have values $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, etc. So, for example, the binary number 110011 represents the value

$$1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 32 + 16 + 2 + 1 = 51$$

in decimal. The 16 bit number above, `0100010001010101`, is equivalent to

$$2^{14} + 2^{10} + 2^6 + 2^4 + 2^2 + 2^0 = 16{,}384 + 1{,}024 + 64 + 16 + 4 + 1 = 17{,}493.$$

Copyright Taylor and Francis, 2021

Tangent 3.3: Hexadecimal notation

Hexadecimal is base 16 notation. Just as the positional values in decimal are powers of 10 and the positional values in binary are powers of 2, the positional values in hexadecimal are powers of 16. Because of this, hexadecimal needs 16 digits, which are `0` through `9`, followed by the letters `a` through `f`. The letter `a` has the value 10, `b` has the value 11, ..., `f` has the value 15. For example, the hexadecimal number `51ed` has the decimal value

$$5 \cdot 16^3 + 1 \cdot 16^2 + 14 \cdot 16^1 + 13 \cdot 16^0 = 5 \cdot 4096 + 1 \cdot 256 + 14 \cdot 16 + 13 \cdot 1 = 20{,}973.$$

Hexadecimal is used a convenient shorthand for binary. Because any 4 binary digits can represent the values 0 through 15, they can be conveniently replaced by a single hexadecimal digit. So the hexadecimal number `100522f10` is equivalent to `00010000000001010010001011110001000` in binary, as shown below:

```
  1     0     0     5     2     2     f     1     0
0001  0000  0000  0101  0010  0010  1111  0001  0000
```

Instead of displaying this 36 bit binary number, it is more convenient to display the 9 digit hexadecimal equivalent.

Reflection 3.2 *To check your understanding, show why the binary number* `1001000` *is equivalent to the decimal number 72.*

This idea can be extended to numbers with a fractional part as well. In decimal, the positions to the right of the decimal point have values that are negative powers of 10: the tenths place has value $10^{-1} = 0.1$, the hundredths place has value $10^{-2} = 0.01$, etc. So the decimal number 18.31 represents the value

$$1 \times 10^1 + 8 \times 10^0 + 3 \times 10^{-1} + 1 \times 10^{-2} = 10 + 8 + 0.3 + 0.01.$$

Similarly, in binary, the positions to the right of the "binary point" have values that are negative powers of 2. For example, the binary number 11.0011 represents the value

$$1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} = 2 + 1 + \frac{1}{8} + \frac{1}{16} = 3\frac{3}{16}$$

in decimal. This is not, however, how we derived (b) above. As we will discuss in Section 3.3, numbers with fractional components are stored in a computer using floating point notation, which converts a number to scientific notation first before storing it.

Reflection 3.3 *To check your understanding, show why the binary number* `1001.101` *is equivalent to the decimal number* 9 5/8.

Computing with bits

A third advantage of binary is that computation on binary values is exceedingly easy. In fact, there are only three fundamental operators, called **and**, **or**, and **not**.

Copyright Taylor and Francis, 2021

These are known as the ***Boolean operators***, after 19th century mathematician George Boole, who is credited with inventing modern mathematical logic, now called *Boolean logic* or *Boolean algebra*.

Let the variables a and b each represent a bit with a value of 0 or 1. Then a **and** b is equal to 1 only if both a and b are equal to 1; otherwise a **and** b is equal to 0.[3] This is conveniently represented by the following ***truth table***:

a	b	a **and** b
0	0	0
0	1	0
1	0	0
1	1	1

Each row of the truth table represents one of the four permutations of the values of a and b. These permutations are shown in the first two columns. The last column of the truth table contains the corresponding values of a **and** b for each row. We see that a **and** b is 1 only when a and b are both 1. If we let 1 represent "true" and 0 represent "false," this conveniently matches our own intuitive meaning of "and." The statement "the barn is red *and* white" is true only if the barn both has red on it and has white on it.

The second Boolean operator, **or**, also takes two operands. The expression a **or** b is equal to 1 if at least one of a or b is equal to 1; otherwise a **or** b is equal to 0. This is represented by the following truth table:

a	b	a **or** b
0	0	0
0	1	1
1	0	1
1	1	1

Notice that a **or** b is 1 even if both a and b are 1. This is different from our normal understanding of "or." If we say that "the barn is red or white," we usually mean it is either red or white, not both. But the Boolean operator can mean both are true. (There is another Boolean operator called "exclusive or" that does mean "either/or," but we won't get into that here.)

Finally, the **not** operator only takes one operand and simply inverts a bit, changing 0 to 1, or 1 to 0. So, **not** a is equal to 1 if a is equal to 0, or 0 if a is equal to 1. The truth table for the **not** operator is simple:

a	**not** a
0	1
1	0

[3] In formal Boolean algebra, a **and** b is usually represented $a \wedge b$, a **or** b is represented $a \vee b$, and **not** a is represented $\neg a$.

Copyright Taylor and Francis, 2021

Notice that this truth table only needs two rows because there are only two possible inputs.

With these basic operators, we can build more complicated expressions. For example, suppose we wanted to find the value of the expression **not** a **and** b. Note that here the **not** operator applies only to the variable a immediately after it, not the entire expression a **and** b. For **not** to apply to the expression, we would need parentheses: **not** (a **and** b). We can evaluate the Boolean expression **not** a **and** b by building a truth table for it. We start by listing all of the combinations of values for a and b, and then create a column for **not** a, since we need that value before we can evaluate the **and** in the expression.

a	b	**not** a
0	0	1
0	1	1
1	0	0
1	1	0

Then, we create a column for **not** a **and** b by **and**ing each value in the **not** a column with its corresponding value in the b column. These individual operations are shown off to the right for each row.

a	b	**not** a	**not** a **and** b	
0	0	1	0	(1 **and** 0 = 0)
0	1	1	1	(1 **and** 1 = 1)
1	0	0	0	(0 **and** 0 = 0)
1	1	0	0	(0 **and** 1 = 0)

So, referring to the first two columns of the second row, we find that **not** a **and** b is 1 only when a is 0 and b is 1. Or, equivalently, **not** a **and** b is *true* only when a is *false* and b is *true*. (Think about that for a moment; it makes sense, doesn't it?)

Python can also work with Boolean values. The Boolean value `0` is represented by `False` in Python and the Boolean value `1` is represented by `True`. (Note the capitalization.) For example, we can validate that the first row of our truth table is correct like this:

```
>>> a = False
>>> b = False
>>> not a and b
False
```

Then, by just changing `b`, we can validate the second row:

```
>>> b = True
>>> not a and b
True
```

Reflection 3.4 *Use Python to validate the third and fourth rows of the truth table as well.*

Copyright Taylor and Francis, 2021

In Chapter 5, we will start using Boolean expressions in conditional statements and loops to control what happens in our programs.

Exercises

3.2.1* Show how to convert the binary number `1101` to decimal.

3.2.2. Show how to convert the binary number `1111101000` to decimal.

3.2.3* Show how to convert the binary number `11.0011` to decimal.

3.2.4. Show how to convert the binary number `11.110001` to decimal.

3.2.5* To convert a decimal number to binary, you can repeatedly divide the number by 2 and keep track of the remainders. The sequence of remainders in reverse order is the binary equivalent. For example, suppose we want to convert 13 to binary. First, divide 13 by 2 and get the quotient and remainder (in Python, `13 // 2` and `13 % 2`), which are 6 and 1. Then do the same with 6, giving quotient 3 and remainder 0. Continue until the quotient is 0. The complete process is summarized in the table below.

	number	number `//` 2	number % 2
Step 1:	13	6	1
Step 2:	6	3	0
Step 3:	3	1	1
Step 4:	1	0	1

The equivalent binary number is the remainder column backwards: `1101`. (Confirm that this is correct by converting this back to decimal.)

Use this process now to convert the decimal number 22 to binary.

3.2.6. Show how to convert the decimal number 222 to binary.

3.2.7* If we want to convert a decimal value less than one to binary, we need a different approach. To perform this conversion, we try to repeatedly subtract decreasing powers of 2, starting with 2^{-1}, from the decimal value. When we can subtract the power of 2, we put a 1 in that place; if we cannot subtract because the power of 2 is too large, we put a 0 in that place instead. For example, let's convert 0.3125 to binary. Since $2^{-1} = 1/2 = 0.5$ is larger than 0.3125, we place a 0 in this place:

	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
	0.5	0.25	0.125	0.0625	...
.	0				

Next, we subtract $2^{-2} = 1/4 = 0.25$ from the remaining 0.3125, leaving 0.0625, and put a 1 in that place:

	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
	0.5	0.25	0.125	0.0625	...
.	0	1			

Copyright Taylor and Francis, 2021

The next place value, $2^{-3} = 1/8 = 0.125$ is larger than our remaining 0.0625, so we put a 0 in that place:

	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
	0.5	0.25	0.125	0.0625	...
.	0	1	0		

The next place value, $2^{-4} = 1/16 = 0.0625$ fits perfectly so after subtracting, we get zero. We put a 1 in that place and we are done.

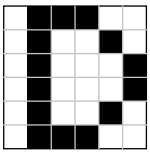
	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
	0.5	0.25	0.125	0.0625	...
.	0	1	0	1	

Our work shows that 0.3125 is decimal is equivalent to 0.0101 in binary.

Use this process to convert the decimal number 3.625 to binary.

3.2.8. Show how to convert the decimal number 25.53125 to binary.

3.2.9. An image stored in a computer is composed of a two-dimensional grid of ***pixels***. In a black and white image, each pixel is either black or white. Consider the 6×6 black and white image to the right. Describe two plausible ways to represent this image as a linear sequence of bits.



3.2.10* Design a truth table for the Boolean expression **not** (*a* **and** *b*).

3.2.11. Design a truth table for the Boolean expression **not** (*a* **or** *b*).

3.2.12* Design a truth table for the Boolean expression **not** *a* **or not** *b*. Compare the result to the truth table for **not** (*a* **and** *b*). What do you notice? The relationship between these two Boolean expressions is one of *De Morgan's laws*.

3.2.13. Design a truth table for the Boolean expression **not** *a* **and not** *b*. Compare the result to the truth table for **not** (*a* **or** *b*). What do you notice? The relationship between these two Boolean expressions is the other of *De Morgan's laws*.

3.3 COMPUTER ARITHMETIC

As we have already seen, computers store numbers in two different ways: as integers and as floats. Understanding this is important because some operators' behaviors depend upon the type they are given. There are also limitations to what is possible with computer arithmetic, meaning that it sometimes behaves differently than real arithmetic depending on which type of numbers you are using.

Limited precision

Binary representation and finite memory often mean that computer arithmetic gives us unexpected results. To illustrate, compute the following very large number:

```
>>> 2.0 ** 100
1.2676506002282294e+30
```

Copyright Taylor and Francis, 2021

This result is a float because `2.0` is a float, and whenever a float is involved in a computation, the result is also a float. Very large floating point numbers like this are printed in scientific notation. The `e` stands for "exponent," and the number following represents a power of ten. So this number represents

$$1.2676506002282294 \times 10^{30} = 1267650600228229400000000000000.$$

(You can also use this notation in your programs, e.g., `18e6` in place of `18000000`.)

Now try computing the same large number with an integer `2` in place of `2.0`:

```
>>> 2 ** 100
1267650600228229401496703205376
```

Reflection 3.5 *Did* `2 ** 100` *and* `2.0 ** 100` *both give the correct answer?*

In normal arithmetic, 2.0^{100} and 2^{100} are, of course, the same number. However, in Python, the results of `2.0 ** 100` and `2 ** 100` are different. The second result is correct because Python integers have ***unlimited precision***, meaning that they can be arbitrarily long, limited only by the computer's memory. The first result is incorrect because the fixed number of bits used to represent floating point numbers leads to ***limited precision***. Although the range of numbers that can be represented in floating point notation is quite large, sometimes a value is too large or too small to even be approximated well. If we line these two numbers up and add commas, we can see that the first result is off by almost 1.5 trillion!

```
1,267,650,600,228,229,401,496,703,205,376
1,267,650,600,228,229,400,000,000,000,000
```

Some numbers cannot be represented at all. For example, try:

```
>>> 10.0 ** 500
OverflowError: (34, 'Result too large')
```

An ***overflow error*** means that the computer did not have enough space to represent the correct value. A similar fatal error will occur if we try to do something illegal, like divide by zero:

```
>>> 10.0 / 0
ZeroDivisionError: division by zero
```

You will also see the effects of limited precision when you perform more common computations. For example, try this:

```
>>> averageWords = 15
>>> averageSyllables = 1.78
>>> readingLevel = 0.39 * averageWords + 11.8 * averageSyllables - 15.59
>>> readingLevel
11.264000000000003
```

The correct answer here is 11.264, but Python didn't make an arithmetic mistake. To execute these statements, our decimal numbers are converted to binary floating point notation, the computations are performed in this format, and then the result is converted back to decimal. At each of these steps, errors may be introduced due to

Copyright Taylor and Francis, 2021

Tangent 3.4: Floating point notation

Python floating point numbers are usually stored in 64 bits of memory, in a format known as IEEE 754 double-precision floating point. One bit is allocated to the sign of the number, 11 bits are allocated to the exponent, and 52 bits are allocated to the ***mantissa***, the fractional part to the right of the point. After a number is converted to binary, the binary point and exponent are adjusted so that there is a single 1 to the left of the binary point. Then the exponent is stored in its 11 bit slot and as much of the mantissa that can fit is stored in the 52 bit slot. If the mantissa is longer than 52 bits, it is simply truncated and information is lost. This is exactly what happens when we compute `2.0 ** 100` on page 118. Since space is limited in computers' memory, tradeoffs between accuracy and space are common, and it is important to understand these limitations to avoid errors. In Chapter 4, we will see a more common situation in which this becomes quite important.

the limited precision of the floating point representation. So when we get the result, it contains a very small error. These kinds of errors are just part of the reality of computer arithmetic. In most of what we do, they won't matter. But in many real scientific computations where very small numbers are the norm, these kinds of errors can have dramatic effects that must be mitigated as a normal part of the process. If you are interested, you can learn more about floating point notation in Tangent 3.4.

Error propagation

Slight errors can become magnified in an iterative computation. To illustrate the problem, suppose we are simulating a process that is occurring continuously over time.[4] Since we cannot actually simulate a continuous process on a computer, we instead repeatedly simulate very small slices of the simulation in a loop. We will use a variable `dt` to represent the length of each slice of time in our simulation. The following loop shows how time would be advanced in this scenario.

```
>>> dt = 0.0001
>>> endTime = 1000000
>>> time = 0
>>> for step in range(1, endTime + 1):
>>>     time = time + dt
>>>     # one slice of the actual simulation would be here
```

Reflection 3.6 *What should the value of* `time` *be at the end of this loop?*

The loop accumulates the value 0.0001 one million times, so `time` should be $0.0001 \cdot 1{,}000{,}000 = 100$. However, it is actually a small fraction over.

```
>>> time
100.00000000219612
```

Since `dt` was very small, there was a slight error every time `dt` was added to `time`.

[4] We will actually develop such simulations in Section 4.4.

Copyright Taylor and Francis, 2021

These errors *propagated* through the loop, making the value of `time` increasingly inaccurate. In some applications, even errors this small may be significant. And it can get even worse with more iterations. Scientific computations can often run for days or weeks, and the number of iterations involved can blow up errors significantly.

Reflection 3.7 *Run the code again but with 10 million iterations. What do you notice about the error?*

To avoid this kind of error propagation, we could have assigned `time` to be the product of `dt` and the current step number:

```
>>> for step in range(1, endTime + 1):
>>>     time = step * dt
```

In this way, the value of `time` is computed from only one arithmetic operation instead of many, reducing the potential error.

Division

When we computed `2 ** 100`, the result was an integer because both operands were integers. This is true everywhere but with division. Even when the result should be an integer, normal division, also called "true division," will give you a float.

```
>>> 100 / 2
50.0
```

Python provides another kind of division, called "floor division," that always gives an integer result. The floor division operator, represented by two slashes (`//`), rounds the quotient down to the nearest integer. (Rounding down to the nearest integer is called "taking the floor" of a number in mathematics, hence the name of the operator.)

```
>>> 14 / 3
4.6666666666666667
>>> 14 // 3
4
```

When both integers are positive, you can think of floor division as the "long division" you learned in elementary school, as it gives the whole quotient without the remainder. In this example, dividing 14 by 3 gives a quotient of 4 and a remainder of 2 because 14 is equal to $4 \cdot 3 + 2$. The operator to get the remainder is called the "modulo" operator. In mathematics, this operator is denoted mod, e.g., $14 \bmod 3 = 2$; in Python it is denoted `%`.

```
>>> 14 % 3
2
```

To see why the `//` and `%` operators might be useful, think about how you would determine whether an integer is odd or even. An integer is even if it is evenly divisible by 2; i.e., when you divide it by 2, the remainder is 0. So, to decide whether an integer is even, we can "mod" the number by 2 and check the answer.

Copyright Taylor and Francis, 2021

```
>>> 14 % 2
0
>>> 15 % 2
1
```

Modular arithmetic is also useful when we need a sequence of numbers to "wrap around," like minutes ticking on a clock. To simulate a clock, instead of just incrementing the minutes with `minutes = minutes + 1`, we want to increment and then "mod" by 60:

```
>>> minutes = 0
>>> minutes = (minutes + 1) % 60
>>> minutes
1
>>> minutes = (minutes + 1) % 60
>>> minutes
2
    ⋮
>>> minutes = (minutes + 1) % 60
>>> minutes
59
>>> minutes = (minutes + 1) % 60
>>> minutes
0
>>> minutes = (minutes + 1) % 60
>>> minutes
1
```

When `minutes` is between 0 and 58, `(minutes + 1) % 60` gives the same result as `minutes + 1` because `minutes + 1` is less than 60. But when `minutes` is 59, `(minutes + 1) % 60` equals `60 % 60`, which is 0.

Complex numbers

Although we will not use them in this book, it is worth pointing out that Python can also handle complex numbers. A complex number has both a real part and an imaginary part involving the imaginary unit i, which has the property that $i^2 = -1$. In Python, a complex number like $3.2 + 2i$ is represented as `3.2 + 2j`. (The letter j is used instead of i because in some fields, such as electrical engineering, i has another well-established meaning that could lead to ambiguities.) Most of the normal arithmetic operators work on complex numbers as well. For example,

```
>>> (5 + 4j) + (-4 + -3.1j)
(1+0.8999999999999999j)
>>> (23 + 6j) / (1 + 2j)
(7-8j)
>>> (1 + 2j) ** 2
(-3+4j)
>>> 1j ** 2
(-1+0j)
```

The last example illustrates the definition of i: $i^2 = -1$.

Copyright Taylor and Francis, 2021

Exercises

3.3.1* The earth is estimated to be 4.54 billion years old. The oldest known fossils of anatomically modern humans are about 200,000 years old. What fraction of the earth's existence have humans been around? Use Python's scientific notation to compute this.

3.3.2. In 2012, the birth rate in the United States was 13.42 per 1,000 people and the total population was estimated to be 313.9 million. How many babies were born in 2012? Use Python's scientific notation to compute this.

3.3.3. The earth is about 4.54 billion years old. How many days are in the age of the earth, taking into account a leap year every four years? Use Python's scientific notation to compute this.

3.3.4* If you counted at an average pace of one number per second, how many years would it take you to count to 4.54 billion? Use Python's scientific notation to compute this.

3.3.5. Suppose the internal clock in a modern computer can "count" about 2.8 billion ticks per second. How long would it take such a computer to tick 4.54 billion times?

3.3.6. The floor division and modulo operators also work with negative numbers. Try some examples, and try to infer what is happening. What are the rules governing the results?

3.3.7. What is the value of each of the following Python expressions? Make sure you understand why in each case.

(a) `15 * 3 - 2`

(b) `15 - 3 * 2`

(c) `15 * 3 // 2`

(d) `15 * 3 / 2`

(e) `15 * 3 % 2`

(f) `15 * 3 / 2e0`

3.3.8* Every cell in the human body contains about 6 billion base pairs of DNA (3 billion in each set of 23 chromosomes). The distance between each base pair is about 3.4 angstroms (3.4×10^{-10} meters). Uncoiled and stretched, how long is the DNA in a single human cell? There are about 50 trillion cells in the human body. If you stretched out all of the DNA in the human body end to end, how long would it be? How many round trips to the sun is this? The distance to the sun is about 149,598,000 kilometers. Write Python statements to compute the answer to each of these three questions. Assign variables to hold each of the values so that you can reuse them to answer subsequent questions.

3.3.9* Suppose the variable `number` refers to an integer. Design a single arithmetic expression that assigns `number` to be `number - 1` if `number` is odd, or leaves `number` the same if it is even.

Copyright Taylor and Francis, 2021

3.3.10. Given a variable `number` that refers to an integer value, show how to extract the individual digits in the ones, tens and hundreds places, and assign those values to three variables named `ones`, `tens`, and `hundreds`. For example, if `number = 123`, `ones`, `tens`, and `hundreds` should be assigned to 1, 2, and 3, respectively. (Use the `//` and `%` operators.)

3.3.11. Compute the very small values `2.0 ** -1074` and `2.0 ** -1075`. Explain the results.

3.3.12. Compute the very large values `2.0 ** 1023` and `2.0 ** 1024`. Explain the results.

3.3.13. Contrast and explain the results of computing `2.0 ** 1024` and `2 ** 1024`.

3.3.14* See how closely you can represent the decimal number 0.1 in binary using six places to the right of the binary point. What is the actual value of your approximation?

3.3.15. Approximate the decimal value 1/3 in binary using six places to the right of the binary point. What is the actual value of your approximation?

3.3.16. Try executing the following statements:

```
number = 1.0
for count in range(10):
    number = number - 0.1
```

What is the final value of `number`? What should it be and why is it incorrect?

*3.4 BINARY ARITHMETIC

This section is available on the book website.

3.5 THE UNIVERSAL MACHINE

The advantages of computing in binary would be useless if we could not actually use binary to compute everything we want to compute. In other words, we need binary computation to be *universal*. But is such a thing really possible? And what do we mean by *any computable problem*? Can we really perform any computation—a web browser, a chess-playing program, Mars rover software—just by converting it to binary and using the **and**, **or**, and **not** operators to get the answer?

Perhaps surprisingly, the answer is **yes**, when we combine the Boolean operators with a sufficiently large memory and a simple controller called a ***finite state machine*** (FSM) to route the correct bits through the correct operators at the correct times. A finite state machine consists of a finite set of *states*, along with *transitions* between states. A state represents the current value of some object or the degree of progress made toward some goal. For example, a simple three-level elevator can be represented by the finite state machine below.

Copyright Taylor and Francis, 2021

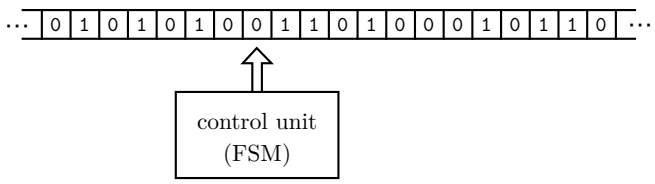
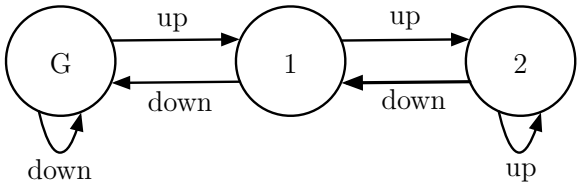


Figure 3.3 A schematic representation of a Turing machine.



The states, representing floors, are circles and the transitions, representing movement between floors, are arrows between circles. In this elevator, there are only up and down buttons (no ability to choose your destination floor when you enter). The label on each transition represents the button press event that causes that transition to occur. For example, when we are on the ground floor and the down button is pressed, we stay on the ground floor. But when the up button is pressed, we transition to the first floor. Many other simple systems, such as vending machines, subway doors, traffic lights and toll booths, can also be represented by finite state machines. A computer's finite state machine coordinates the fetch and execute cycle, as well as various intermediate steps involved in executing machine language instructions.

The question of whether a computational system is universal has its roots in the very origins of computer science itself. In 1936, Alan Turing proposed an abstract computational model, now called a ***Turing machine***, that he proved could compute any problem considered to be mathematically computable. As illustrated in Figure 3.3, a Turing machine consists of a control unit containing a finite state machine that can read from and write to an infinitely long tape. The tape contains a sequence of "cells," each of which can contain a single character. The tape is initially inscribed with some sequence of input characters, and a pointer attached to the control unit is positioned at the beginning of the input. In each step, the Turing machine reads the character in the current cell. Then, based on what it reads and the current state, the finite state machine decides whether to write a character in the cell and whether to move its pointer one cell to the left or right. Not unlike the fetch and execute cycle in a modern computer, the Turing machine repeats this simple process as long as needed, until a designated final state is reached. The output is the final sequence of characters on the tape.

The Turing machine still stands as our modern definition of computability. The ***Church-Turing thesis*** states that a problem is computable if and only if it can be computed by a Turing machine. Any mechanism that can be shown to be equivalent

Copyright Taylor and Francis, 2021

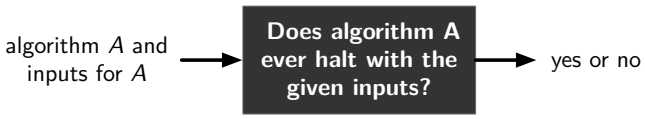


Figure 3.4 The halting problem.

in computational power to a Turing machine is considered to be computationally universal, or ***Turing complete***. A modern computer, based on Boolean logic and a finite state machine, falls into this category.

Almost every problem that we would want to compute is, thankfully, computable, even if it takes an extraordinarily long time to compute. There are, however, problems that are not computable. The most famous is called the ***halting problem***, illustrated in Figure 3.4. An algorithm for the halting problem, if it were to exist, would need to take another algorithm and an input for that algorithm as inputs and then decide whether that algorithm ever halts with the correct answer. But it can be proven that there are algorithms for which the halting problem cannot give an answer. Since an algorithm must be correct for every possible input, this means that there cannot exist a correct algorithm for the halting problem. This is not just an academic problem. In practice, this means that it is impossible to ever create program that can verify whether other programs are correct! So errors in programs are here to stay, unfortunately.

Exercises

3.5.1* Design a finite state machine that represents a highway toll booth controlling a single gate. First, think about what the states should be. Then design the transitions between states.

3.5.2. Design a finite state machine that represents a vending machine. Assume that the machine only takes quarters and vends only one kind of drink, for 75 cents. First, think about what the states should be. Then design the transitions between states.

3.6 SUMMARY AND FURTHER DISCOVERY

In this chapter, we took a peek under the hood to glimpse what a computer is really doing when it executes our programs. We saw that a computer system is itself a complex layering of abstractions. We are isolated from a lot of these details, thankfully, by the interpreter, which transparently translates our programs into machine language. A machine language program may yet need to request that the operating system do things on its behalf (e.g., saving a file or allocating more memory). Below that, each instruction in the resulting computation will be implemented using Boolean logic, controlled by a finite state machine. Knowing this should make you quite thankful that all of these layers of abstraction exist, and that we are able to

Copyright Taylor and Francis, 2021

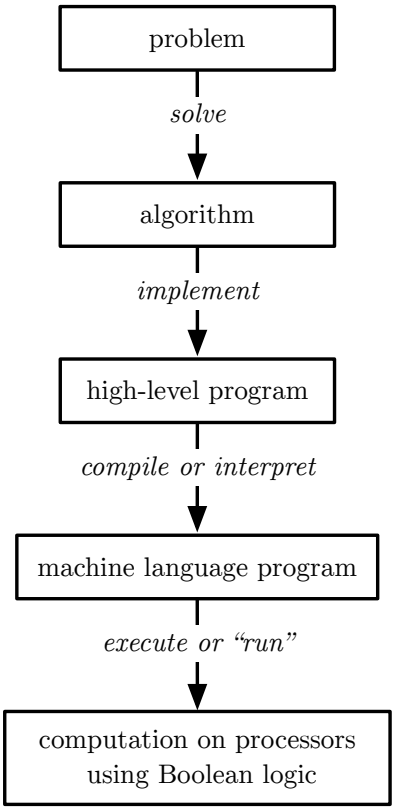


Figure 3.5 Top-to-bottom view of how an algorithm becomes a computation.

solve problems at a much higher layer! This top-to-bottom view of the process of turning an algorithm into a computation is summarized in Figure 3.5.

Notes for further discovery

There are several excellent books available that give overviews of computer science and how computers work. In particular, we recommend *The Pattern on the Stone* by Danny Hillis [23], *Algorithmics* by David Harel [22], *Digitized* by Peter Bentley [7], and *CODE: The Hidden Language of Computer Hardware and Software* by Charles Petzold [49].

A list of the world's fastest supercomputers is maintained at `http://top500.org`. You can learn more about IBM's Deep Blue supercomputer at

`http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/`

and IBM's Watson at

`https://www.ibm.com/ibm/history/ibm100/us/en/icons/watson/`.

To learn more about high performance computing in general, we recommend looking

Copyright Taylor and Francis, 2021

at the website of the National Center for Supercomputing Applications (NCSA) at the University of Illinois, one of the first supercomputer centers funded by the National Science Foundation, at `http://www.ncsa.illinois.edu/about/faq`.

There are also several good books available on the life of Alan Turing, the father of computer science. The definitive biography, *Alan Turing: The Enigma*, was written by Andrew Hodges [25].

Copyright Taylor and Francis, 2021