- You will understand and apply abstraction.
- You will learn to think and communicate more clearly.

1.4 Problem-Solving Strategies

Problem solving happens on three different levels:

- **Strategy:** A high-level idea for finding a solution.
- **Tactics:** Methods or patterns that work in many different settings.
- **Tools:** Specific tricks and techniques that are used in specific situations.

Paul Zeitz [Zei99] provides us with a helpful analogy for illustrating the three different levels of problem solving:

> You are standing at the base of a mountain, hoping to climb to the summit. Your first strategy may be to take several small trips to various easier peaks nearby, so as to observe the target mountain from different angles. After this, you may consider a somewhat more focused strategy, perhaps to try climbing the mountain via a particular ridge. Now the tactical considerations begin: how to actually achieve the chosen strategy. For example, suppose that our strategy suggests climbing the south ridge of the peak, but there are snowfields and rivers in our path. Different tactics are needed to negotiate each of these obstacles. For the snowfield, our tactic may be to travel early in the morning while the snow is hard. For the river, our tactic may be scouting the banks for the safest crossing. Finally, move into the most tightly focused level, that of tools: specific techniques to accomplish specialized tasks. For example, to cross the snowfield we may set up a particular system of ropes for safety and walk with ice axes. The river crossing may require the party to strip from the waist down and hold hands for balance. These are all tools. They are very specific. You would never summarize, "To climb the mountain we had to take our pants off and hold hands," because it was a minor–though essential–component of the entire climb. On the other hand, strategic and sometimes tactical ideas are often described in your summary: "We decided to reach the summit via the south ridge and had to cross a difficult snowfield and a dangerous river to get to the ridge."

As you progress through this book, you will encounter several different problem-solving strategies. In addition, you will see that computer science uses many different problem-solving tactics. In particular you will learn to recognize patterns in the problems you solve that lead to patterns in the programs you write. Finally, as you use the Python programming language you will learn about the tools that Python provides to write your solution as a program.

A simple example will illustrate some of what we are talking about. The question is as follows: "A class has 12 students. At the beginning of class each student shakes hands with each of the other students. How many handshakes take place?"

Your first instinct might be to simply say that since each person must shake hands with 11 other people the answer is $12 \cdot 11 = 132$ handshakes, but you would be wrong. To help you make progress toward the correct answer, you can employ a strategy called **simplification**. Simplification is a strategy that reduces a problem to a trivial size.

Let's assume that instead of 12 people there is only one person in the classroom. When there is only a single person, no handshakes will take place. But what happens when a second person enters the classroom? Upon entering the room, the second person must shake hands with the first (and only other) person in the room for one handshake. Now suppose a third person enters the classroom. The third person must shake hands with the first two, making a total of $2 + 1 + 0 = 3$ handshakes. The fourth person who enters the room must shake hands with the three people already in the room, so our total handshake count is now $3 + 2 + 1 + 0 = 6$. By this time you should see a pattern of **generalization**—a technique that enables you to go from some specific examples to a solution that can be implemented as a program.

In our handshaking problem the pattern is telling us that the Nth person to enter the classroom shakes hands with $N - 1$ other people, and the total number of handshakes is the sum $1 + 2 + 3 + \ldots N - 1$. At this point we might simply write a computer program that adds up the numbers from 1 to $N - 1$ for us. Adding numbers is something that computers are particularly good at. However, we will also point out that there is a general solution for this problem for adding a sequence of numbers:

$$sum = \frac{n \cdot (n + 1)}{2}$$

For our handshake problem, we need to add up the numbers from 1 to 1 less than the number of students. Given that there are 12 students, $n = 11$. Plugging 11 into the formula gives us:

$$\frac{11 \cdot 12}{2} = 66$$

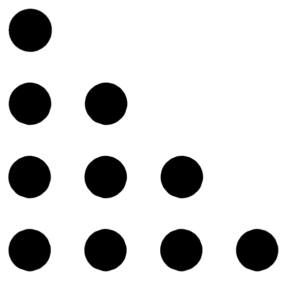


Figure 1.1 Representing the sum of the numbers from 1 to N graphically

You can verify this result yourself by simply adding the numbers from 1 to 11.

In fact we can *prove* that the formula gives us the correct answer by using **representation**, another important strategy that will solve our problem. Proving that $\sum_1^n n = \frac{n \cdot (n+1)}{2}$ is true for all values of n using mathematical induction would be a daunting task for most people. However, let's visualize the problem of adding up the numbers from 1 to N as shown in Figure 1.1.

In this representation of the problem, we show each of the numbers we want to add as a row of circles, thus representing the addition of $1 + 2 + 3 + 4$. Now we could just count the circles to get our answer, but that is not very interesting and does not prove anything. The interesting part comes in Figure 1.2, where we have taken all four rows of dots, duplicated them, and flipped them diagonally. The dots now form a rectangle that is 4 rows high and 5 columns wide. It is now easy to see that the total number of dots is just $4 \cdot 5 = 20$. But we have twice as many dots as we started with, so the number of dots we started with is $20 \div 2 = 10$.

If you generalize our example a little bit, it is easy to see that this graphical trick works no matter how many rows of dots we use. Therefore we have shown a proof for an interesting mathematical sequence, but because we chose a good representation for the problem we have not had to do anything more complicated than simple multiplication and division.

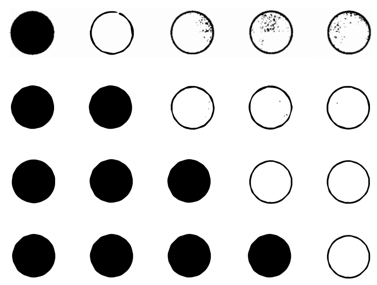


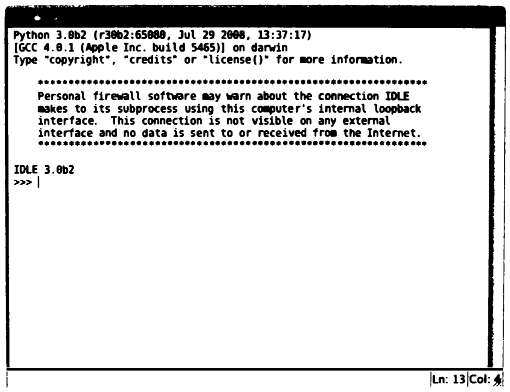
Figure 1.2 The sum of the numbers 1 to N is $\frac{n \cdot (n+1)}{2}$

1.5 Python Overview

In this book the language you will use to write your computer programs is called **Python**. Why did we choose Python instead of a language like C++ or Java? The answer is simple: We want you to focus on learning the problem-solving strategies and techniques that a computer scientist uses. **Programming languages** are tools and Python is a good beginning tool. Languages like Java and C++ are fine tools as well, but they require you to keep track of many more details and they are harder to learn than Python.

The best way to learn Python is to try it out—so let's get started. The first thing we are going to do is start the Python interpreter. The program we are going to use to introduce Python is called IDLE—named after Eric Idle of Monty Python fame. Depending on your operating system, you can select IDLE in your Start menu and click on the IDLE icon in your Applications folder. Or you can simply type `idle` at a command prompt. No matter how you start it, you will know you are successful when you see a window such as the one shown in Figure 1.3. For detailed instructions on installing and starting Python on your system, refer to Appendix A.

As you progress through this chapter, you will see that example programs are in boxes called **listings**, and commands that you can type interactively at the Python shell are in boxes called **sessions**. Whenever you see a session box, we strongly encourage you to try the session for yourself. Also, once you have typed in the example we have shown, feel free to try some variations in order to find out for yourself what works and what does not.



```
Python 3.0b2 (r30b2:65080, Jul 29 2008, 13:37:17)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 3.0b2
>>> |
```

Figure 1.3 The Python shell in IDLE

As we begin to explore Python, we will answer three important questions you should ask about any programming language:

- What are the primitive elements?
- How can we combine the primitive elements?
- How can we create our own abstractions?

1.5.1 Primitive Elements

At the deepest level, the one primitive element in Python is the **object**. In fact, everything in Python is an object, and you will read this refrain often in this book. By now you are probably wondering what we mean by *object*. After all, if you look around you will see many objects: this book, pencils, pens, your chair, a computer. What do these items have to do with Python? Like you, Python thinks of the things in its world as objects. Python even considers numbers to be objects–an idea that may be a bit confusing to you as you probably don't think of numbers as objects. But Python does, and we'll see why this is important shortly.

Python classifies the different kinds of objects in its world into **types**. Some of the easiest types to work with are numbers. Python knows about several different types of numbers:

- Integer numbers
- Floating point numbers
- Complex numbers

Integer Numbers

Integers are the whole numbers that you learned about in math class. We will introduce more of Python's primitive types as we progress through this chapter. But before we move on let's look at Python's integers in more detail. We can already do a lot with Python just using integers. For starters, we can use the Python shell we started a few moments ago as a calculator. Let's try a few mathematical expressions. Type in the following examples using the Python shell in IDLE. After you have typed in an expression, press the return key to see the result.

```
>>> 2+2
4
>>> 100-75
25
>>> 7*9
63
>>> 14//2
7
>>> 15//2
7
>>> 15 % 2
1
```

Session 1.1 Simple integer math

The examples in Session 1.1 illustrate some very important Python concepts that you should become familiar with as soon as possible. The first concept is Python's evaluation loop.

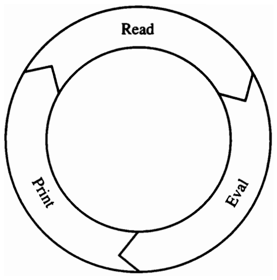


Figure 1.4 The Read–Eval–Print loop in Python

At a high level, the Python interpreter is really very simple. It does three things over and over again: (1) read, (2) evaluate, and (3) print. These are illustrated in Figure 1.4.

First, Python *reads* one line of input. In the first example, Python reads 2 + 2, then it *evaluates* the expression 2 + 2 and determines that the answer is 4. Finally, Python *prints* the resulting value of 4. After displaying the result, Python prints the characters >>> to show you that it is waiting to read another expression. The three characters >>> are called the Python **prompt**.

In general a Python **expression** is a combination of operators and operands. In the examples in our Python session, the operators are familiar mathematical operators '+', '−', '*', and '//.' You may be more used to × and ÷ for multiplication and division, but you will not find those symbols on a standard keyboard, so Python, and almost all other programming languages use "*" and "/."

One thing that may surprise you in the example is the result of the expression `15//2`. Of course, we all know that 15 divided by 2 is really `7.5`. However, because both operands are integer objects and // is the integer division operator, Python produces an integer object as a result. Integer division works like the division you learned when you were young. 15 divided by 2 equals 7, remainder 1. You can find out the remainder part of the result using the modulo operator (%). For example, 15 % 7 evaluates to the remainder value of 1.

Integer division is really useful in some cases, but it can also trip you up. What if you want to divide 7 by 2 and get 7.5 as the answer? In order to get the result as a **floating point** number you must use the floating point division operator '/'.

Exercises

1.1 Find the sum of the numbers 8, 9, and 10. 8 + 9 + 10

1.2 Find the product of the numbers 8, 9, and 10. 8 * 9 * 10

1.3 Compute the number of seconds in a year. 365 * 24 * 60 * 60

1.4 Compute the number of inches in 1 mile. 12

1.5 Compute the number of 2 ft square tiles to cover the floor of a 10 by 12 ft room.
12 * 10 / 2

1.6 Compute the number of handshakes required to shake all the hands of your classmates.

1.7 Find the average age of five people around you using integer division. Double-check your answer.

Floating-Point Numbers

Floating-point numbers are Python's approximation of what you called real numbers in math class. We say that floating-point numbers are an approximation because unlike real numbers, floating-point numbers cannot have an infinite number of digits following the decimal point. In Python you can tell the difference between a floating-point number and an integer because a float has a decimal point. Session 1.2 presents some examples of math using floating-point numbers.

```
>>> 2.0 + 2.0
4.0
>>> 2 + 2.0
4.0
>>> 15 / 2
7.5
>>> 2.0 ** 50
1125899906842624.0
>>> 2.5 ** 25
8881784197.0012531
>>> 2.0 ** 500
3.2733906078961419e+150
>>> 1.33e+5 + 1.0
133001.0
```

Session 1.2 Floating-point math

Notice that we have added something new in this example: the ** symbol, which is called the exponentiation operator. So `2.0 ** 50` is really two to the fiftieth power. You should also notice that when the result of a floating-point operation gets really big, Python uses scientific notation to express the results. The Python number `3.273e+150` really means 3.273 times 10 to the 150th power, or 3273 followed by 147 zeros! A very big number indeed. Notice also that you can use floating-point numbers in scientific notation as part of a Python expression.

Exercises

1.8 Find the average age of five people around you using floating-point division. Double-check your answer.

1.9 Find the volume of a sphere with a radius of 1 using the formula $4/3\pi r^3$.

1.10 Compute 1/3 of 15. Did you get the right answer?

1.11 The Andromeda galaxy is 2.9 million light-years away. There are 5.878×10^{12} miles per light-year. How many miles away is the Andromeda galaxy?

1.12 How many years would it take to travel to the Andromeda galaxy at 65 miles per hour?

Although 3.273e + 150 is a good approximation, we know that there are not really 147 zeros in the result. One of the disadvantages of using scientific notation is that you lose some *precision* in your result. If you want to get very exact results, integers allow us to do calculations to unlimited precision. Session 1.3 shows the real value of 2 ** 500 using integers.

```
>>> 2 ** 500
3273390607896141870013189696827599152216642046043064789483291368096133796404674554883270092325904157150886684127560071009217256545885393053328527589376
>>>
```

Session 1.3 The use of integers to obtain very precise answers for large numbers

Exercises

1.13 Compute the factorial of 13.

1.14 Compute 2 to the 120th power. 2** 120

1.15 If the universe is 15 billion years old, how many seconds old is it?

1.16 How many handshakes would it take for each person in Chicago to shake hands with every other person?

Complex Numbers

The final primitive numeric type in Python is the **complex number**. As you may remember, complex numbers have two parts to them, a real part and an imaginary part. In Python a complex number is displayed as *real* + *imaginary*j. For example, 5.0 + 3*j* has a real part of 5.0 and an imaginary part of 3. Although we mention complex numbers here to give you a complete list of the numeric primitives, we will not go into any additional details at this point.

Summary of Numeric Types

What happens when we mix integers and floating point numbers? Let's look at the examples shown in Session 1.4 to find out.

```
>>> 100 * 3.4
340.0
>>> 100000000000000000000000000 * 3.4
3.4000000000000003e+26
>>> 10000000 * 1000000
10000000000000
>>> 1000000000 / 1000000000
1
>>> 1000 // 10.0
100.0
>>> 1000 / 10.2
98.039215686274517
>>> 1000 // 10.2
98.0
>>> 5 + 4+3j
(9+3j)
```

Session 1.4 Mixing integers, long integers, floats, and complex numbers

When mixing different types of numbers, you can figure out what the result will be converted to by applying the following rules:

1. If either argument is a complex number, the other is converted to complex.
2. If either argument is a floating-point number, the other is converted to floating-point.
3. For all other arguments, both must be plain integers and no conversion is needed.

Notice that when using floating-point numbers with the integer division operator the result is a floating-point number with the fractional part truncated. You can also tell Python to explicitly convert a number to either an integer or floating-point number by using the `int` or `float` functions. For example, `float(5)` will convert the integer 5 to the floating-point number 5.0. When converting floating-point numbers to integers, Python always truncates the fractional part of the number. For example, `int(3.99999)` will convert the floating-point number 3.99999 to the integer 3.

In summary, we have seen that Python supports several different types of primitive objects in the number family: integers for ordinary simple math; or, when precision is required or when dealing with very large numbers; floating-point numbers, for working with scientific applications or accounting applications where we need to keep track of dollars and cents. We have seen that Python can be used to make simple numerical calculations. However, at this point Python is nothing more than a calculator. In the next section we will add some additional Python primitives that will give us a lot more power.

1.5.2 Naming Objects

Very often we have an object that we would like to remember. Python allows us to **name** objects so that we can refer to them later. For example, we might want to use the name *pi* rather than the value 3.14159 in a mathematical expression. We might also want to give a name to a value that we are going to use over and over again rather than recalculating it each time.

In Python we can name objects using an **assignment statement**. A statement is like an expression except that it does not produce a value for the read–eval–print loop to print. An assignment statement has three parts: (1) the left-hand side, (2) the right-hand side, and (3) the assignment operator (=). The left side contains the name we are assigning to a variable, and the right side can be any Python expression.

```
variableName = python expression
```

When the Python interpreter evaluates an assignment statement, it first evaluates the expression that it finds on the right-hand side of the equals sign. Once the right-hand side expression has been evaluated, the resulting object is referred to using the name found on the left side of the equals sign. In computer science, we call these names **variables**. More formally, we define a variable to be a named reference to a data object. In other words, a variable is simply a name that allows us to locate a Python object.

Suppose we want to calculate the volume of a cylinder where the radius of the base is 8 cm and the height is 16 cm. We will use the formula *volume = area of base * height*. Rather than calculate everything in one big expression, we will divide the work into several assignment statements. First, we will name the numeric objects "pi," "radius," and "height." Next,