

This is particularly important when programs are written by more than one person. It may be obvious to *you* that *cv* stands for can volume and not current velocity, but will it be obvious to the person who needs to update your code years later? For that matter, will you remember yourself what *cv* means when you look at the code three months from now?

Programming Tip 2.2



Do Not Use Magic Numbers

A **magic number** is a numeric constant that appears in your code without explanation. For example,

```
totalVolume = bottles * 2
```

Why 2? Are bottles twice as voluminous as cans? No, the reason is that every bottle contains 2 liters. Use a named constant to make the code self-documenting:

```
BOTTLE_VOLUME = 2.0
totalVolume = bottles * BOTTLE_VOLUME
```

There is another reason for using named constants. Suppose circumstances change, and the bottle volume is now 1.5 liters. If you used a named constant, you make a single change, and you are done. Otherwise, you have to look at every value of 2 in your program and ponder whether it meant a bottle volume or something else. In a program that is more than a few pages long, that is incredibly tedious and error-prone.

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
DAYS_PER_YEAR = 365
```



We prefer programs that are easy to understand over those that appear to work by magic.

2.2 Arithmetic

In the following sections, you will learn how to carry out arithmetic calculations in Python.

2.2.1 Basic Arithmetic Operations



Python supports the same four basic arithmetic operations as a calculator—addition, subtraction, multiplication, and division—but it uses different symbols for multiplication and division.

You must write $a * b$ to denote multiplication. Unlike in mathematics, you cannot write $a b$, $a \cdot b$, or $a \times b$. Similarly, division is always indicated with a $/$, never a $+$ or a fraction bar.

For example, $\frac{a+b}{2}$ becomes $(a + b) / 2$.

The symbols $+ - * /$ for the arithmetic operations are called **operators**. The combination of variables, literals, operators, and parentheses is called an **expression**. For example, $(a + b) / 2$ is an expression.

Parentheses are used just as in algebra: to indicate in which order the parts of the expression should be computed. For example, in the expression $(a + b) / 2$, the sum $a + b$ is computed first, and then the sum is divided by 2. In contrast, in the expression

$$a + b / 2$$

only b is divided by 2, and then the sum of a and $b / 2$ is formed. As in regular algebraic notation, multiplication and division have a *higher precedence* than addition and subtraction. For example, in the expression $a + b / 2$, the $/$ is carried out first, even though the $+$ operation occurs further to the left. Again, as in algebra, operators with the same precedence are executed left-to-right. For example, $10 - 2 - 3$ is $8 - 3$ or 5.

If you mix integer and floating-point values in an arithmetic expression, the result is a floating-point value. For example, $7 + 4.0$ is the floating-point value 11.0.

Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.

2.2.2 Powers

Python uses the exponential operator `**` to denote the power operation. For example, the Python equivalent of the mathematical expression a^2 is `a ** 2`. Note that there can be no space between the two asterisks. As in mathematics, the exponential operator has a higher order of precedence than the other arithmetic operators. For example, `10 * 2 ** 3` is $10 \cdot 2^3 = 80$. Unlike the other arithmetic operators, power operators are evaluated from right to left. Thus, the Python expression `10 ** 2 ** 3` is equivalent to $10^{(2^3)} = 10^8 = 100,000,000$.

In algebra, you use fractions and exponents to arrange expressions in a compact two-dimensional form. In Python, you have to write all expressions in a linear arrangement. For example, the mathematical expression

$$b \times \left(1 + \frac{r}{100}\right)^n$$

becomes

```
b * (1 + r / 100) ** n
```

Figure 3 shows how to analyze such an expression.

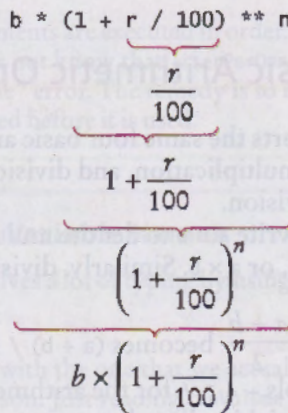


Figure 3 Analyzing an Expression



2.2.3 Floor Division and Remainder

The // operator computes floor division in which the remainder is discarded.

When you divide two integers with the / operator, you get a floating-point value. For example,

$$7 / 4$$

yields 1.75. However, we can also perform **floor division** using the // operator. For positive integers, floor division computes the quotient and discards the fractional part. The floor division

$$7 // 4$$

evaluates to 1 because 7 divided by 4 is 1.75 with a fractional part of 0.75 (which is discarded).

If you are interested in the remainder of a floor division, use the % operator. The value of the expression

$$7 \% 4$$

is 3, the remainder of the floor division of 7 by 4. The % symbol has no analog in algebra. It was chosen because it looks similar to /, and the remainder operation is related to division. The operator is called **modulus**. (Some people call it *modulo* or *mod*.) It has no relationship with the percent operation that you find on some calculators.

Here is a typical use for the // and % operations. Suppose you have an amount of pennies in a piggybank:

$$\text{pennies} = 1729$$

You want to determine the value in dollars and cents. You obtain the dollars through a floor division by 100:

$$\text{dollars} = \text{pennies} // 100 \quad \# \text{ Sets dollars to } 17$$

The floor division discards the remainder. To obtain the remainder, use the % operator:

$$\text{cents} = \text{pennies} \% 100 \quad \# \text{ Sets cents to } 29$$

See Table 3 for additional examples.

Floor division and modulus are also defined for negative integers and floating-point numbers. However, those definitions are rather technical, and we do not cover them in this book.



Floor division and the % operator yield the dollar and cent values of a piggybank full of pennies.

Table 3 Floor Division and Remainder

Expression (where $n = 1729$)	Value	Comment
$n \% 10$	9	For any positive integer n , $n \% 10$ is the last digit of n .
$n // 10$	172	This is n without the last digit.
$n \% 100$	29	The last two digits of n .
$n \% 2$	1	$n \% 2$ is 0 if n is even, 1 if n is odd (provided n is not negative)
$-n // 10$	-173	-173 is the largest integer ≤ -172.9 . We will not use floor division for negative numbers in this book.

2.2.4 Calling Functions

A function can return a value that can be used as if it were a literal value.

You learned in Chapter 1 that a function is a collection of programming instructions that carry out a particular task. We have been using the `print` function to display information, but there are many other functions available in Python. In this section, you will learn more about functions that work with numbers.

Most functions **return** a value. That is, when the function completes its task, it passes a value back to the point where the function was called. One example is the `abs` function that returns the absolute value—the value without a sign—of its numerical argument. For example, the call `abs(-173)` returns the value 173.

The value returned by a function can be stored in a variable:

```
distance = abs(x)
```

In fact, the returned value can be used anywhere that a value of the same type can be used:

```
print("The distance from the origin is", abs(x))
```

The `abs` function requires data to perform its task, namely the number from which to compute the absolute value. As you learned earlier, data that you provide to a function are the arguments of the call. For example, in the call

```
abs(-10)
```

the value `-10` is the argument passed to the `abs` function.

When calling a function, you must provide the correct number of arguments. The `abs` function takes exactly one argument. If you call

```
abs(-10, 2)
```

or

```
abs()
```

your program will generate an error message.

Some functions have optional arguments that you only provide in certain situations. An example is the `round` function. When called with one argument, such as

```
round(7.625)
```

the function returns the nearest integer; in this case, 8. When called with two arguments, the second argument specifies the desired number of fractional digits.

Syntax 2.2 Calling Functions

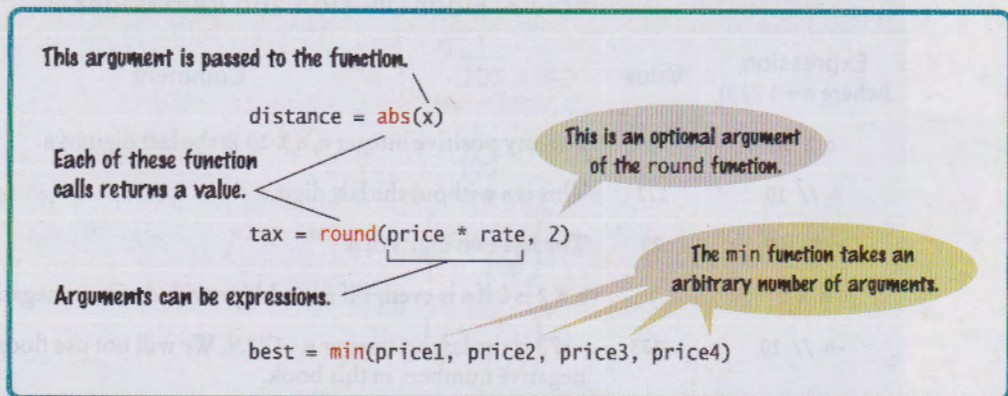


Table 4 Built-in Mathematical Functions

Function	Returns
<code>abs(x)</code>	The absolute value of x .
<code>round(x)</code> <code>round(x, n)</code>	The floating-point value x rounded to a whole number or to n decimal places.
<code>max(x₁, x₂, ..., x_n)</code>	The largest value from among the arguments.
<code>min(x₁, x₂, ..., x_n)</code>	The smallest value from among the arguments.

For example,

```
round(7.625, 2)
```

is 7.63.

There are two common styles for illustrating optional arguments. One style, which we use in this book, shows different function calls with and without the optional arguments.

```
round(x) # Returns x rounded to a whole number.
round(x, n) # Returns x rounded to n decimal places.
```

The second style, which is used in Python's standard documentation, uses square brackets to denote the optional arguments.

```
round(x[, n]) # Returns x rounded to a whole number or to n decimal places.
```

Finally, some functions, such as the `max` and `min` functions, take an arbitrary number of arguments. For example, the call

```
cheapest = min(7.25, 10.95, 5.95, 6.05)
```

sets the variable `cheapest` to the minimum of the function's arguments; in this case, the number 5.95.

Table 4 shows the functions that we introduced in this section.

2.2.5 Mathematical Functions

The Python language itself is relatively simple, but Python contains a standard library that can be used to create powerful programs. A **library** is a collection of code that has been written and translated by someone else, ready for you to use in your program. A **standard library** is a library that is considered part of the language and must be included with any Python system.

Python's standard library is organized into **modules**. Related functions and data types are grouped into the same module. Functions defined in a module must be explicitly loaded into your program before they can be used. Python's `math` module includes a number of mathematical functions. To use any function from this module, you must first import the function. For example, to use the `sqrt` function, which computes the square root of its argument, first include the statement

```
from math import sqrt
```

at the top of your program file. Then you can simply call the function as

```
y = sqrt(x)
```

Python has a standard library that provides functions and data types for your code.

Library function must be imported into your program before it can be used.

Table 5 Selected Functions in the math Module


Function	Returns
<code>sqrt(x)</code>	The square root of x . ($x \geq 0$)
<code>trunc(x)</code>	Truncates floating-point value x to an integer.
<code>cos(x)</code>	The cosine of x in radians.
<code>sin(x)</code>	The sine of x in radians.
<code>tan(x)</code>	The tangent of x in radians.
<code>exp(x)</code>	e^x
<code>degrees(x)</code>	Convert x radians to degrees (i.e., returns $x \cdot 180/\pi$)
<code>radians(x)</code>	Convert x degrees to radians (i.e., returns $x \cdot \pi/180$)
<code>log(x)</code> <code>log(x, base)</code>	The natural logarithm of x (to base e) or the logarithm of x to the given <i>base</i> .

Table 5 shows additional functions defined in the `math` module.

While most functions are defined in a module, a small number of functions (such as `print` and the functions introduced in the preceding section) can be used without importing any module. These functions are called **built-in** functions because they are defined as part of the language itself and can be used directly in your programs.

Table 6 Arithmetic Expression Examples

Mathematical Expression	Python Expression	Comments
$\frac{x+y}{2}$	<code>(x + y) / 2</code>	The parentheses are required; <code>x + y / 2</code> computes $x + \frac{y}{2}$.
$\frac{xy}{2}$	<code>x * y / 2</code>	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1 + \frac{r}{100}\right)^n$	<code>(1 + r / 100) ** n</code>	The parentheses are required.
$\sqrt{a^2 + b^2}$	<code>sqrt(a ** 2 + b ** 2)</code>	You must import the <code>sqrt</code> function from the <code>math</code> module.
π	<code>pi</code>	<code>pi</code> is a constant declared in the <code>math</code> module.


SELF CHECK

- A bank account earns interest once per year. In Python, how do you compute the interest earned in the first year? Assume variables `percent` and `balance` both contain floating-point values.
- In Python, how do you compute the side length of a square whose area is stored in the variable `area`?

10. The volume of a sphere is given by

$$V = \frac{4}{3}\pi r^3$$

If the radius is given by a variable `radius` that contains a floating-point value, write a Python expression for the volume.

11. What is the value of `1729 // 10` and `1729 % 10`?
 12. If `n` is a positive number, what is `(n // 10) % 10`?

Practice It Now you can try these exercises at the end of the chapter: R2.3, R2.5, P2.4, P2.5.

Common Error 2.2



Roundoff Errors

Roundoff errors are a fact of life when calculating with floating-point numbers. You probably have encountered that phenomenon yourself with manual calculations. If you calculate $1/3$ to two decimal places, you get 0.33. Multiplying again by 3, you obtain 0.99, not 1.00.

In the processor hardware, numbers are represented in the binary number system, using only digits 0 and 1. As with decimal numbers, you can get roundoff errors when binary digits are lost. They just may crop up at different places than you might expect.

Here is an example:

```
price = 4.35
quantity = 100
total = price * quantity # Should be 100 * 4.35 = 435
print(total) # Prints 434.99999999999994
```

In the binary system, there is no exact representation for 4.35, just as there is no exact representation for $1/3$ in the decimal system. The representation used by the computer is just a little less than 4.35, so 100 times that value is just a little less than 435.

You can deal with roundoff errors by rounding to the nearest integer or by displaying a fixed number of digits after the decimal separator (see Section 2.5.3).

Common Error 2.3



Unbalanced Parentheses

Consider the expression

$$((a + b) * t / 2 * (1 - t))$$

What is wrong with it? Count the parentheses. There are three (and two). The parentheses are *unbalanced*. This kind of typing error is very common with complicated expressions. Now consider this expression.

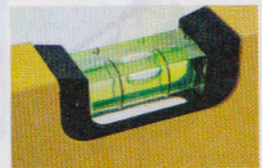
$$(a + b) * t) / (2 * (1 - t))$$

This expression has three (and three), but it still is not correct. In the middle of the expression,

$$(a + b) * t) / (2 * (1 - t)$$

there is only one (but two), which is an error. At any point in an expression, the count of (must be greater than or equal to the count of), and at the end of the expression the two counts must be the same.

Here is a simple trick to make the counting easier without using pencil and paper. It is difficult for the brain to keep two counts



simultaneously. Keep only one count when scanning the expression. Start with 1 at the first opening parenthesis, add 1 whenever you see an opening parenthesis, and subtract one whenever you see a closing parenthesis. Say the numbers aloud as you scan the expression. If the count ever drops below zero, or is not zero at the end, the parentheses are unbalanced. For example, when scanning the previous expression, you would mutter

$$\begin{array}{cccc} (a + b) * t) / (2 * (1 - t) \\ 1 & 0 & -1 & \end{array}$$

and you would find the error.

Programming Tip 2.3



Use Spaces in Expressions

It is easier to read

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c)) / (2 * a)
```

than

```
x1=(-b+sqrt(b**2-4*a*c))/(2*a)
```

Simply put spaces around all operators (+ - * / % =, and so on). However, don't put a space after a *unary* minus: a - used to negate a single quantity, such as -b. That way, it can be easily distinguished from a *binary* minus, as in a - b.

It is customary not to put a space after a function name. That is, write `sqrt(x)` and not `sqrt (x)`.

Special Topic 2.1



Other Ways to Import Modules

Python provides several different ways to import functions from a module into your program. You can import multiple functions from the same module like this:

```
from math import sqrt, sin, cos
```

You can also import the entire contents of a module into your program:

```
from math import *
```

Alternatively, you can import the module with the statement

```
import math
```

With this form of the `import` statement, you need to add the module name and a period before each function call, like this:

```
y = math.sqrt(x)
```

Some programmers prefer this style because it makes it very explicit to which module a particular function belongs.

Special Topic 2.2



Combining Assignment and Arithmetic

In Python, you can combine arithmetic and assignment. For example, the instruction

```
total += cans
```

is a shortcut for

```
total = total + cans
```

Similarly,

```
total *= 2
```


is another way of writing

```
total = total * 2
```

Many programmers find this a convenient shortcut especially when incrementing or decrementing by 1:

```
count += 1
```

If you like it, go ahead and use it in your own code. For simplicity, we won't use it in this book.

Special Topic 2.3



Line Joining

If you have an expression that is too long to fit on a single line, you can continue it on another line *provided the line break occurs inside parentheses*. For example,

```
x1 = ((-b + sqrt(b ** 2 - 4 * a * c))
      / (2 * a)) # Ok
```

However, if you omit the outermost parentheses, you get an error:

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c))
      / (2 * a) # Error
```

The first line is a complete statement, which the Python interpreter processes. The next line, `/ (2 * a)`, makes no sense by itself.

There is a second form of joining long lines. If the *last* character of a line is a backslash, the line is joined with the one following it:

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c)) \
      / (2 * a) # Ok
```

You must be very careful not to put any spaces or tabs after the backslash. In this book, we only use the first form of line joining.

2.3 Problem Solving: First Do It By Hand

In the preceding section, you learned how to express computations in Python. When you are asked to write a program for solving a problem, you may naturally think about the Python syntax for the computations. However, before you start programming, you should first take a very important step: carry out the computations *by hand*. If you can't compute a solution yourself, it's unlikely that you'll be able to write a program that automates the computation.

To illustrate the use of hand calculations, consider the following problem: A row of black and white tiles needs to be placed along a wall. For aesthetic reasons, the architect has specified that the first and last tile shall be black.

Your task is to compute the number of tiles needed and the gap at each end, given the space available and the width of each tile.

