

PROGRAMMING WITH NUMBERS AND STRINGS

CHAPTER GOALS

- To define and use variables and constants
- To understand the properties and limitations of integers and floating-point numbers
- To appreciate the importance of comments and good code layout
- To write arithmetic expressions and assignment statements
- To create programs that read and process inputs, and display the results
- To learn how to use Python strings
- To create simple graphics programs using basic shapes and text

CHAPTER CONTENTS

2.1 VARIABLES 28

Syntax 2.1: Assignment 29

Common Error 2.1: Using Undefined Variables 34

Programming Tip 2.1: Choose Descriptive Variable Names 34

Programming Tip 2.2: Do Not Use Magic Numbers 35

2.2 ARITHMETIC 35

Syntax 2.2: Calling Functions 38

Common Error 2.2: Roundoff Errors 41

Common Error 2.3: Unbalanced Parentheses 41

Programming Tip 2.3: Use Spaces in Expressions 42

Special Topic 2.1: Other Ways to Import Modules 42

Special Topic 2.2: Combining Assignment and Arithmetic 42

Special Topic 2.3: Line Joining 43

2.3 PROBLEM SOLVING: FIRST DO IT BY HAND 43

Worked Example 2.1: Computing Travel Time 45

2.4 STRINGS 46

Special Topic 2.4: Character Values 51

Special Topic 2.5: Escape Sequences 52

Computing & Society 2.1: International Alphabets and Unicode 52

2.5 INPUT AND OUTPUT 53

Syntax 2.3: String Format Operator 55

Programming Tip 2.4: Don't Wait to Convert 58

How To 2.1: Writing Simple Programs 58

Worked Example 2.2: Computing the Cost of Stamps 61

Computing & Society 2.2: The Pentium Floating-Point Bug 63

2.6 GRAPHICS: SIMPLE DRAWINGS 63

How To 2.2: Graphics: Drawing Graphical Shapes 70

Toolbox 2.1: Symbolic Processing with SymPy 73



© samxmeg/iStockphoto.



© samxmeg/iStockphoto.

Numbers and character strings (such as the ones on this display board) are important data types in any Python program. In this chapter, you will learn how to work with numbers and text, and how to write simple programs that perform useful tasks with them.

2.1 Variables

When your program carries out computations, you will want to store values so that you can use them later. In a Python program, you use **variables** to store values. In this section, you will learn how to define and use variables.

To illustrate the use of variables, we will develop a program that solves the following problem. Soft drinks are sold in cans and bottles. A store offers a six-pack of 12-ounce cans for the same price as a two-liter bottle. Which should you buy? (Twelve fluid ounces equal approximately 0.355 liters.)



(cans) © blackred/iStockphoto; (bottle) © travismanley/iStockphoto.

What contains more soda? A six-pack of 12-ounce cans or a two-liter bottle?

In our program, we will define variables for the number of cans per pack and for the volume of each can. Then we will compute the volume of a six-pack in liters and print out the answer.

2.1.1 Defining Variables

A variable is a storage location with a name.

A **variable** is a storage location in a computer program. Each variable has a name and holds a value.

A variable is similar to a parking space in a parking garage. The parking space has an identifier (such as "J 053"), and it can hold a vehicle. A variable has a name (such as `cansPerPack`), and it can hold a value (such as 6).



Javier Larreal/AGE Fotostock.

Like a variable in a computer program, a parking space has an identifier and a contents.

Syntax 2.1 Assignment

Syntax `variableName = value`

A variable is defined the first time it is assigned a value.

```
total = 0
.
.
total = bottles * BOTTLE_VOLUME
.
.
total = total + cans * CAN_VOLUME
```

The same name can occur on both sides. See Figure 2.

Names of previously defined variables

The expression that replaces the previous value

Names of previously defined variables

An assignment statement stores a value in a variable.

You use the **assignment statement** to place a value into a variable. Here is an example

```
cansPerPack = 6
```

The left-hand side of an assignment statement consists of a variable. The right-hand side is an expression that has a value. That value is stored in the variable.

The first time a variable is assigned a value, the variable is created and initialized with that value. After a variable has been defined, it can be used in other statements. For example,

```
print(cansPerPack)
```

will print the value stored in the variable `cansPerPack`.

If an existing variable is assigned a new value, that value replaces the previous contents of the variable. For example,

```
cansPerPack = 8
```

changes the value contained in variable `cansPerPack` from 6 to 8. Figure 1 illustrates the two assignment statements used above.

The = sign does not mean that the left-hand side is *equal* to the right-hand side. Instead, the value on the right-hand side is placed into the variable on the left.

Do not confuse this **assignment operator** with the = used in algebra to denote equality. Assignment is an instruction to do something—namely, place a value into a variable.

1 Because this is the first assignment, the variable is created.

```
cansPerPack =
```

2 The variable is initialized.

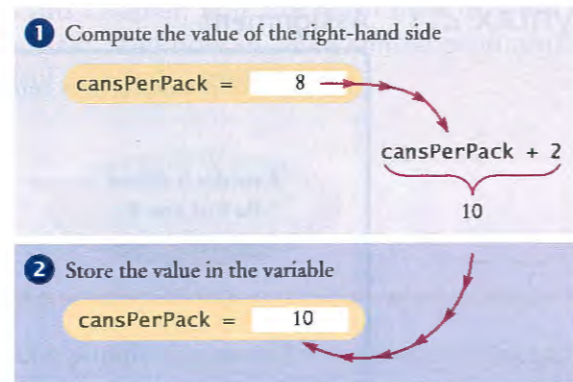
```
cansPerPack = 6
```

3 The second assignment overwrites the stored value.

```
cansPerPack = 8
```

Figure 1 Executing Two Assignments

Figure 2
Executing the Assignment
`cansPerPack = cansPerPack + 2`



For example, in Python, it is perfectly legal to write

```
cansPerPack = cansPerPack + 2
```

The second statement means to look up the value stored in the variable `cansPerPack`, add 2 to it, and place the result back into `cansPerPack`. (See Figure 2.) The net effect of executing this statement is to increment `cansPerPack` by 2. If `cansPerPack` was 8 before execution of the statement, it is set to 10 afterwards. Of course, in mathematics it would make no sense to write that $x = x + 2$. No value can equal itself plus 2.

2.1.2 Number Types

The data type of a value specifies how the value is stored in the computer and what operations can be performed on the value.

Integers are whole numbers without a fractional part.

Floating-point numbers contain a fractional part.

Computers manipulate data values that represent information and these values can be of different types. In fact, each value in a Python program is of a specific type. The **data type** of a value determines how the data is represented in the computer and what operations can be performed on that data. A data type provided by the language itself is called a **primitive data type**. Python supports quite a few data types: numbers, text strings, files, containers, and many others. Programmers can also define their own **user-defined data types**, which we will cover in detail in Chapter 9.

In Python, there are several different types of numbers. An **integer** value is a whole number without a fractional part. For example, there must be an integer number of cans in any pack of cans—you cannot have a fraction of a can. In Python, this type is called `int`. When a fractional part is required (such as in the number 0.355), we use **floating-point** numbers, which are called `float` in Python.

When a value such as 6 or 0.355 occurs in a Python program, it is called a **number literal**. If a number literal has a decimal point, it is a floating-point number; otherwise, it is an integer. Table 1 shows how to write integer and floating-point literals in Python.

A variable in Python can store a value of any type. The data type is associated with the *value*, not the variable. For example, consider this variable that is initialized with a value of type `int`:

```
taxRate = 5
```

The same variable can later hold a value of type `float`:

```
taxRate = 5.5
```

Once a variable is initialized with a value of a particular type, it should always store values of that same type.

Table 1 Number Literals in Python

Number	Type	Comment
6	int	An integer has no fractional part.
-6	int	Integers can be negative.
0	int	Zero is an integer.
0.5	float	A number with a fractional part has type float.
1.0	float	An integer with a fractional part .0 has type float.
1E6	float	A number in exponential notation: 1×10^6 or 1000000. Numbers in exponential notation always have type float.
2.96E-2	float	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
⊘ 100,000		Error: Do not use a comma as a decimal separator.
⊘ 3 1/2		Error: Do not use fractions; use decimal notation: 3.5.

It could even hold a string:

```
taxRate = "Non-taxable" # Not recommended
```

However, that is not a good idea. If you use the variable and it contains a value of an unexpected type, an error will occur in your program. Instead, once you have initialized a variable with a value of a particular type, you should take care that you keep storing values of the same type in that variable.

For example, because tax rates are not necessarily integers, it is a good idea to initialize the `taxRate` variable with a floating-point value, even if it happens to be a whole number:

```
taxRate = 5.0 # Tax rates can have fractional parts
```

This helps you remember that `taxRate` can contain a floating-point value, even though the initial value has no fractional part.

2.1.3 Variable Names

When you define a variable, you need to give it a name that explains its purpose. Whenever you name something in Python, you must follow a few simple rules:

- Names must start with a letter or the underscore (`_`) character, and the remaining characters must be letters, numbers, or underscores.
- You cannot use other symbols such as `?` or `%`. Spaces are not permitted inside names either. You can use uppercase letters to denote word boundaries, as in `cansPerPack`. This naming convention is called *camel case* because the uppercase letters in the middle of the name look like the humps of a camel.



© GlobalPR/istockphoto.

Table 2 Variable Names in Python

Variable Name	Comment
canVolume1	Variable names consist of letters, numbers, and the underscore character.
x	In mathematics, you use short variable names such as <i>x</i> or <i>y</i> . This is legal in Python, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 34).
CanVolume	Caution: Variable names are case sensitive. This variable name is different from <code>canVolume</code> , and it violates the convention that variable names should start with a lowercase letter.
6pack	Error: Variable names cannot start with a number.
can volume	Error: Variable names cannot contain spaces.
class	Error: You cannot use a reserved word as a variable name.
ltr/fl.oz	Error: You cannot use symbols such as <code>.</code> or <code>/</code> .

- Names are **case sensitive**, that is, `canVolume` and `canvoLume` are different names.
- You cannot use **reserved words** such as `if` or `class` as names; these words are reserved exclusively for their special Python meanings. (See Appendix B for a listing of all reserved words in Python.)

These are firm rules of the Python language. There are two “rules of good taste” that you should also respect.

- It is better to use a descriptive name, such as `cansPerPack`, than a terse name, such as `cpp`.
- Most Python programmers use names for variables that start with a lowercase letter (such as `cansPerPack`). In contrast, names that are all uppercase (such as `CAN_VOLUME`) indicate constants. Names that start with an uppercase letter are commonly used for user-defined data types (such as `GraphicsWindow`).

Table 2 shows examples of legal and illegal variable names in Python.

2.1.4 Constants

A constant variable, or simply a **constant**, is a variable whose value should not be changed after it has been assigned an initial value. Some languages provide an explicit mechanism for marking a variable as a constant and will generate a syntax error if you attempt to assign a new value to the variable. Python leaves it to the programmer to make sure that constants are not changed. Thus, it is common practice to specify a constant variable with the use of all capital letters for its name.

```
BOTTLE_VOLUME = 2.0
MAX_SIZE = 100
```

By following this convention, you provide information to yourself and others that you intend for a variable in all capital letters to be constant throughout the program.

It is good programming style to use named constants in your program to explain numeric values.

For example, compare the statements

```
totalVolume = bottles * 2
```

and

```
totalVolume = bottles * BOTTLE_VOLUME
```

A programmer reading the first statement may not understand the significance of the number 2. The second statement, with a named constant, makes the computation much clearer.

2.1.5 Comments

As your programs get more complex, you should add **comments**, explanations for human readers of your code. For example, here is a comment that explains the value used in a constant:

```
CAN_VOLUME = 0.355 # Liters in a 12-ounce can
```

This comment explains the significance of the value 0.355 to a human reader. The interpreter does not execute comments at all. It ignores everything from a `#` delimiter to the end of the line.

It is a good practice to provide comments. This helps programmers who read your code understand your intent. In addition, you will find comments helpful when you review your own programs. Provide a comment at the top of your source file that explains the purpose of the program. In the textbook, we use the following style for these comments,

```
##
# This program computes the volume (in liters) of a six-pack of soda cans.
#
```

Now that you have learned about variables, constants, the assignment statement, and comments, we are ready to write a program that solves the problem from the beginning of chapter. The program displays the volume of a six-pack of cans and the total volume of the six-pack and a two-liter bottle. We use constants for the can and bottle volumes. The `totalVolume` variable is initialized with the volume of the cans. Using an assignment statement, we add the bottle volume. As you can see from the program output, the six-pack of cans contains over two liters of soda.

ch02/sec01/volume1.py

```
1 ##
2 # This program computes the volume (in liters) of a six-pack of soda
3 # cans and the total volume of a six-pack and a two-liter bottle.
4 #
5
6 # Liters in a 12-ounce can and a two-liter bottle.
7 CAN_VOLUME = 0.355
8 BOTTLE_VOLUME = 2.0
9
10 # Number of cans per pack.
11 cansPerPack = 6
```



Just as a television commentator explains the news, you use comments in your program to explain its behavior.

Use comments to add explanations for humans who read your code. The interpreter ignores comments.

By convention, variable names should start with a lowercase letter.

Use constants for values that should remain unchanged throughout your program.

```

12
13 # Calculate total volume in the cans.
14 totalVolume = cansPerPack * CAN_VOLUME
15 print("A six-pack of 12-ounce cans contains", totalVolume, "liters.")
16
17 # Calculate total volume in the cans and a 2-liter bottle.
18 totalVolume = totalVolume + BOTTLE_VOLUME
19 print("A six-pack and a two-liter bottle contain", totalVolume, "liters.")

```

Program Run

A six-pack of 12-ounce cans contains 2.13 liters.
A six-pack and a two-liter bottle contain 4.13 liters.



1. Define a variable suitable for holding the number of bottles in a case.
2. What is wrong with the following statement?
`ounces per liter = 28.35`
3. Define two variables, `unitPrice` and `quantity`, to contain the unit price of a single bottle and the number of bottles purchased. Use reasonable initial values.
4. Use the variables declared in Self Check 3 to print the total purchase price.
5. Some drinks are sold in four-packs instead of six-packs. How would you change the `volume1.py` program to compute the total volume?
6. Why can't the variable `totalVolume` in the `volume1.py` program be a constant variable?
7. How would you explain assignment using the parking space analogy?

Practice It Now you can try these exercises at the end of the chapter: R2.1, R2.2, P2.1.

Common Error 2.1**Using Undefined Variables**

A variable must be created and initialized before it can be used for the first time. For example, a program starting with the following sequence of statements would not be legal:

```

canVolume = 12 * literPerOunce # Error: literPerOunce has not yet been created.
literPerOunce = 0.0296

```

In your program, the statements are executed in order. When the first statement is executed by the virtual machine, it does not know that `literPerOunce` will be created in the next line, and it reports an "undefined name" error. The remedy is to reorder the statements so that each variable is created and initialized before it is used.

Programming Tip 2.1**Choose Descriptive Variable Names**

We could have saved ourselves a lot of typing by using shorter variable names, as in

```
cv = 0.355
```

Compare this declaration with the one that we actually used, though. Which one is easier to read? There is no comparison. Just reading `canVolume` is a lot less trouble than reading `cv` and then *figuring out* it must mean "can volume".

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.

© FinnBrandt/iStockphoto.

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 · b`, or `a × 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.

© hocus-focus/iStockphoto.



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 $a + b / 2$

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

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.

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.

$$\begin{array}{c}
 b * (1 + r / 100) ** n \\
 \underbrace{\hspace{1.5cm}} \\
 \frac{r}{100} \\
 \underbrace{\hspace{1.5cm}} \\
 1 + \frac{r}{100} \\
 \underbrace{\hspace{1.5cm}} \\
 \left(1 + \frac{r}{100}\right)^n \\
 \underbrace{\hspace{1.5cm}} \\
 b \times \left(1 + \frac{r}{100}\right)^n
 \end{array}$$

Figure 3 Analyzing an Expression

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

The % operator computes the remainder of a floor division.

2.2.3 Floor Division and Remainder

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.

© Michael Flipppo/istockphoto.

Table 3 Floor Division and Remainder

Expression (where n = 1729)	Value	Comment
<code>n % 10</code>	9	For any positive integer n, <code>n % 10</code> is the last digit of n.
<code>n // 10</code>	172	This is n without the last digit.
<code>n % 100</code>	29	The last two digits of n.
<code>n % 2</code>	1	<code>n % 2</code> is 0 if n is even, 1 if n is odd (provided n is not negative)
<code>-n // 10</code>	-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

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.

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

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

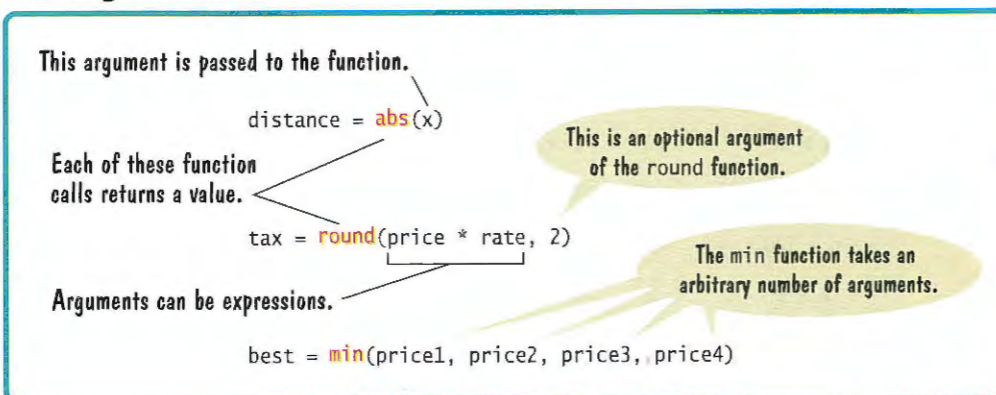


Table 4 Built-in Mathematical Functions

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(7.627, 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

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

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)
```

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

Table 5 Selected Functions in the math Module

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

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



- 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?

- 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.

- What is the value of 1729 // 10 and 1729 % 10?
- 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, P2.3, P2.4.

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.

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

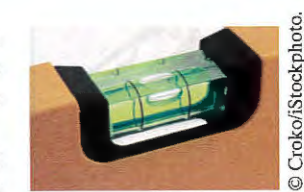
This expression has three (and three), but it still is not correct. In the middle of the expression,

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

↑

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



© Crokovi/Stockphoto.

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

```
(a + b) * t) / (2 * (1 - t)
1   0  -1
```

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
```

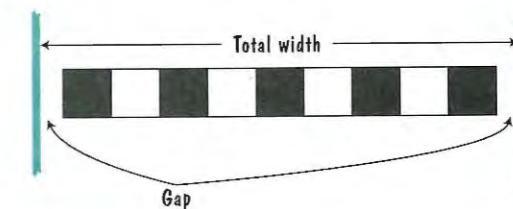
You must be very careful not to put any spaces or tabs after the backslash. In this book, we only use the first form of line joining.

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.



Pick concrete values for a typical situation to use in a hand calculation.

To make the problem more concrete, let's assume the following dimensions:

- Total width: 100 inches
- Tile width: 5 inches

The obvious solution would be to fill the space with 20 tiles, but that would not work—the last tile would be white.

Instead, look at the problem this way: The first tile must always be black, and then we add some number of white/black pairs:



The first tile takes up 5 inches, leaving 95 inches to be covered by pairs. Each pair is 10 inches wide. Therefore the number of pairs is $95 / 10 = 9.5$. However, we need to discard the fractional part since we can't have fractions of tile pairs.

Therefore, we will use 9 tile pairs or 18 tiles, plus the initial black tile. Altogether, we require 19 tiles.

The tiles span $19 \times 5 = 95$ inches, leaving a total gap of $100 - 19 \times 5 = 5$ inches.

The gap should be evenly distributed at both ends. At each end, the gap is $(100 - 19 \times 5) / 2 = 2.5$ inches.

This computation gives us enough information to devise an algorithm with arbitrary values for the total width and tile width.

$$\text{number of pairs} = \text{integer part of } (\text{total width} - \text{tile width}) / (2 \times \text{tile width})$$

$$\text{number of tiles} = 1 + 2 \times \text{number of pairs}$$

$$\text{gap at each end} = (\text{total width} - \text{number of tiles} \times \text{tile width}) / 2$$

As you can see, doing a hand calculation gives enough insight into the problem that it becomes easy to develop an algorithm. See `ch02/sec03/tiles.py` in your source code for the complete program.

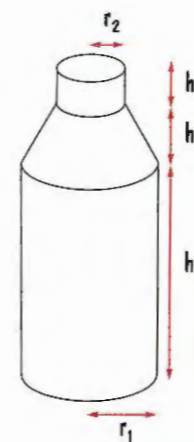
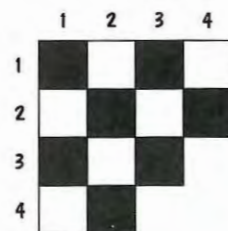


- Translate the pseudocode for computing the number of tiles and the gap width into Python.
- Suppose the architect specifies a pattern with black, gray, and white tiles, like this:



Again, the first and last tile should be black. How do you need to modify the algorithm?

- A robot needs to tile a floor with alternating black and white tiles. Develop an algorithm that yields the color (0 for black, 1 for white), given the row and column number. Start with specific values for the row and column, and then generalize.



- For a particular car, repair and maintenance costs in year 1 are estimated at \$100; in year 10, at \$1,500. Assuming that the repair cost increases by the same amount every year, develop pseudocode to compute the repair cost in year 3 and then generalize to year n .
- The shape of a bottle is approximated by two cylinders of radius r_1 and r_2 and heights h_1 and h_2 , joined by a cone section of height h_3 .

Using the formulas for the volume of a cylinder, $V = \pi r^2 h$, and a cone section,

$$V = \pi \frac{(r_1^2 + r_1 r_2 + r_2^2) h_3}{3},$$

develop pseudocode to compute the volume of the bottle. Using an actual bottle with known volume as a sample, make a hand calculation of your pseudocode.

Practice It Now you can try these exercises at the end of the chapter: R2.15, R2.17, R2.18.

WORKED EXAMPLE 2.1 Computing Travel Time



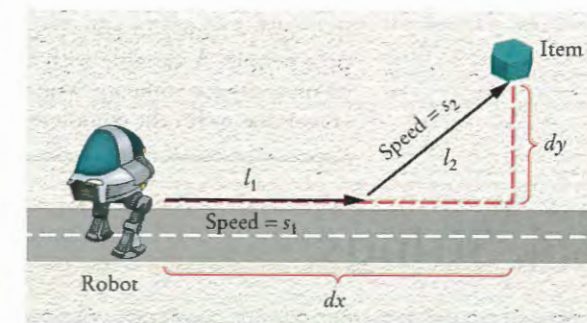
Problem Statement A robot needs to retrieve an item that is located in rocky terrain next to a road. The robot can travel at a faster speed on the road than on the rocky terrain, so it will want to do so for a certain distance before moving in a straight line to the item. Calculate by hand how much time it takes to reach the item.



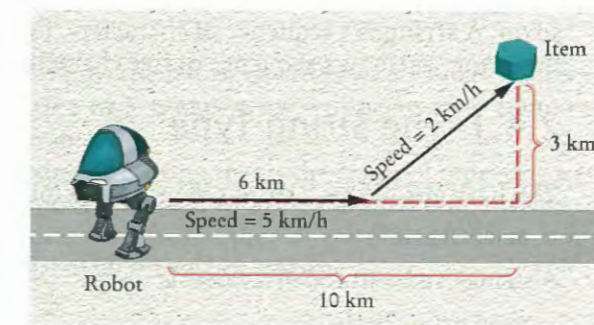
Courtesy of NASA.

Your task is to compute the total time taken by the robot to reach its goal, given the following inputs:

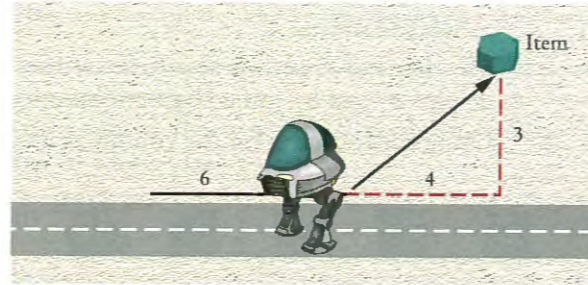
- The distance between the robot and the item in the x - and y -direction (dx and dy)
- The speed of the robot on the road and the rocky terrain (s_1 and s_2)
- The length l_1 of the first segment (on the road)



To make the problem more concrete, let's assume the following dimensions:



The total time is the time for traversing both segments. The time to traverse the first segment is simply the length of the segment divided by the speed: 6 km divided by 5 km/h, or 1.2 hours.



To compute the time for the second segment, we first need to know its length. It is the hypotenuse of a right triangle with side lengths 3 and 4.

Therefore, its length is $\sqrt{3^2 + 4^2} = 5$. At 2 km/h, it takes 2.5 hours to traverse it. That makes the total travel time 3.7 hours.

This computation gives us enough information to devise an algorithm for the total travel time with arbitrary arguments:

Time for segment 1 = l_1 / s_1
 Length of segment 2 = square root of $[(dx - l_1)^2 + dy^2]$
 Time for segment 2 = length of segment 2 / s_2
 Total time = time for segment 1 + time for segment 2

Translated into Python, the computations are

```
segment1Time = segment1Length / segment1Speed
segment2Length = sqrt((xDistance - segment1Length) ** 2 + yDistance ** 2)
segment2Time = segment2Length / segment2Speed
totalTime = segment1Time + segment2Time
```

Note that we use variable names that are longer and more descriptive than dx or s_1 . When you do hand calculations, it is convenient to use the shorter names, but you should change them to descriptive names in your program. In real life, programs are commonly developed by multiple people. A variable with a short name like s_1 may have meaning to you, but it may have no meaning for someone who works on the program at a later time. See `chD2/worked_example_1/traveltime.py` for the complete program.

2.4 Strings

Strings are sequences of characters.

Many programs process text, not numbers. Text consists of **characters**: letters, numbers, punctuation, spaces, and so on. A **string** is a sequence of characters. For example, the string "Hello" is a sequence of five characters.

2.4.1 The String Type

You have already seen strings in print statements such as

```
print("Hello")
```

A string can be stored in a variable

```
greeting = "Hello"
```



A string literal denotes a particular string.

and later accessed when needed just as numerical values can be:

```
print(greeting)
```

A **string literal** denotes a particular string (such as "Hello"), just as a number literal (such as 2) denotes a particular number. In Python, string literals are specified by enclosing a sequence of characters within a matching pair of either single or double quotes.

```
print("This is a string.", 'So is this.')
```

By allowing both types of delimiters, Python makes it easy to include an apostrophe or quotation mark within a string.

```
message = 'He said "Hello"'
```

In this book, we use double quotation marks around strings because this is a common convention in many other programming languages. However, the interactive Python interpreter always displays strings with single quotation marks.

The number of characters in a string is called the *length* of the string. For example, the length of "Harry" is 5. You can compute the length of a string using Python's `len` function:

```
length = len("World!") # length is 6
```

A string of length 0 is called the *empty string*. It contains no characters and is written as "" or ''.

The len function returns the number of characters in a string.

2.4.2 Concatenation and Repetition

Use the + operator to concatenate strings; that is, to put them together to yield a longer string.

Given two strings, such as "Harry" and "Morgan", you can **concatenate** them to one long string. The result consists of all characters in the first string, followed by all characters in the second string. In Python, you use the + operator to concatenate two strings. For example,

```
firstName = "Harry"
lastName = "Morgan"
name = firstName + lastName
```

results in the string

```
"HarryMorgan"
```

What if you'd like the first and last name separated by a space? No problem:

```
name = firstName + " " + lastName
```

This statement concatenates three strings: `firstName`, the string literal " ", and `lastName`. The result is

```
"Harry Morgan"
```

When the expression to the left or the right of a + operator is a string, the other one must also be a string or a syntax error will occur. You cannot concatenate a string with a numerical value.

You can also produce a string that is the result of repeating a string multiple times. For example, suppose you need to print a dashed line. Instead of specifying a literal string with 50 dashes, you can use the * operator to create a string that is comprised of the string "-" repeated 50 times. For example,

```
dashes = "-" * 50
```

A string can be repeated using the * operator.

results in the string

"-----"

A string of any length can be repeated using the * operator. For example, the statements

```
message = "Echo..."
print(message * 5)
```

display

```
Echo...Echo...Echo...Echo...Echo...
```

The factor by which the string is replicated must be an integer value. The factor can appear on either side of the * operator, but it is common practice to place the string on the left side and the integer factor on the right.

2.4.3 Converting Between Numbers and Strings

Sometimes it is necessary to convert a numerical value to a string. For example, suppose you need to append a number to the end of a string. You cannot concatenate a string and a number:

```
name = "Agent " + 1729 # Error: Can only concatenate strings
```

Because string concatenation can only be performed between two strings, we must first convert the number to a string.

To produce the string representation of a numerical value, use the str function. The statement

```
str(1729)
```

converts the integer value 1729 to the string "1729". The str function solves our problem:

```
id = 1729
name = "Agent " + str(id)
```

The str function converts an integer or floating-point value to a string.

The str function can also be used to convert a floating-point value to a string.

Conversely, to turn a string containing a number into a numerical value, use the int and float functions:

```
id = int("1729")
price = float("17.29")
```

The int and float functions convert a string containing a number to the numerical value.

This conversion is important when the strings come from user input (see Section 2.5.1).

The string passed to the int or float functions can only consist of those characters that comprise a literal value of the indicated type. For example, the statement

```
value = float("17x29")
```

will generate a run-time error because the letter "x" cannot be part of a floating-point literal.

Blank spaces at the front or back will be ignored: int(" 1729 ") is still 1729.

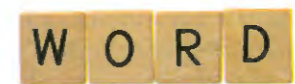
2.4.4 Strings and Characters

Strings are sequences of Unicode characters (see Computing & Society 2.1). You can access the individual characters of a string based on their position within the string. This position is called the *index* of the character.

String positions are counted starting with 0.

The first character has index 0, the second has index 1, and so on.

```
H a r r y
0 1 2 3 4
```



A string is a sequence of characters.

© slp/x/Stockphoto.

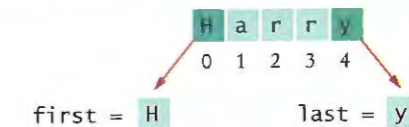
An individual character is accessed using a special subscript notation in which the position is enclosed within square brackets. For example, if the variable name is defined as

```
name = "Harry"
```

the statements

```
first = name[0]
last = name[4]
```

extract two different characters from the string. The first statement extracts the first character as the string "H" and stores it in variable first. The second statement extracts the character at position 4, which in this case is the last character, and stores it in variable last.



The index value must be within the valid range of character positions or an "index out of range" exception will be generated at run time. The len function can be used to determine the position of the last index, or the last character in a string.

```
pos = len(name) - 1 # Length of "Harry" is 5
last = name[pos] # last is set to "y"
```

The following program puts these concepts to work. The program initializes two variables with strings, one with your name and the other with that of your significant other. It then prints out your initials.

The operation first[0] makes a string consisting of one character, taken from the start of first. The operation second[0] does the same for the second name. Finally, you concatenate the resulting one-character strings with the string literal "&" to get a string of length 3, the initials string. (See Figure 4.)

```
first = R o d o l f o
      0 1 2 3 4 5 6
second = S a l l y
        0 1 2 3 4

initials = R & S
          0 1 2
```

Figure 4 Building the initials String

ch02/sec04/initials.py

```
1 ##
2 # This program prints a pair of initials.
3 #
4
```



Initials are formed from the first letter of each name.

© Rich Legg/Stockphoto.

```

5 # Set the names of the couple.
6 first = "Rodolfo"
7 second = "Sally"
8
9 # Compute and display the initials.
10 initials = first[0] + "&" + second[0]
11 print(initials)

```

Table 7 String Operations

Statement	Result	Comment
<code>string = "Py"</code> <code>string = string + "thon"</code>	string is set to "Python"	When applied to strings, + denotes concatenation.
<code>print("Please" +</code> <code>" enter your name: ")</code>	Prints Please enter your name:	Use concatenation to break up strings that don't fit into one line.
<code>team = str(49) + "ers"</code>	team is set to "49ers"	Because 49 is an integer, it must be converted to a string.
<code>greeting = "H & S"</code> <code>n = len(greeting)</code>	n is set to 5	Each space counts as one character.
<code>string = "Sally"</code> <code>ch = string[1]</code>	ch is set to "a"	Note that the initial position is 0.
<code>last = string[len(string) - 1]</code>	last is set to the string containing the last character in string	The last character has position <code>len(string) - 1</code> .

2.4.5 String Methods

In computer programming, an **object** is a software entity that represents a value with certain behavior. The value can be simple, such as a string, or complex, like a graphical window or data file. You will learn much more about objects in Chapter 9. For now, you need to master a small amount of notation for working with string objects.

The behavior of an object is given through its **methods**. A method, like a function, is a collection of programming instructions that carry out a particular task. But unlike a function, which is a standalone operation, a method can only be applied to an object of the type for which it was defined. For example, you can apply the `upper` method to any string, like this:

```

name = "John Smith"
uppercaseName = name.upper() # Sets uppercaseName to "JOHN SMITH"

```

Note that the method name follows the object, and that a dot (.) separates the object and method name.

There is another string method called `lower` that yields the lowercase version of a string:

```
print(name.lower()) # Prints john smith
```

It is a bit arbitrary when you need to call a function (such as `len(name)`) and when you need to call a method (`name.lower()`). You will simply need to remember or look it up in a printed or online Python reference.

Just like function calls, method calls can have arguments. For example, the string method `replace` creates a new string in which every occurrence of a given substring is replaced with a second string. Here is a call to that method with two arguments:

```
name2 = name.replace("John", "Jane") # Sets name2 to "Jane Smith"
```

Note that none of the method calls change the contents of the string on which they are invoked. After the call `name.upper()`, the `name` variable still holds "John Smith". The method call returns the uppercase version. Similarly, the `replace` method returns a new string with the replacements, without modifying the original.

Table 8 lists the string methods introduced in this section.

Table 8 Useful String Methods

Method	Returns
<code>s.lower()</code>	A lowercase version of string <code>s</code> .
<code>s.upper()</code>	An uppercase version of <code>s</code> .
<code>s.replace(old, new)</code>	A new version of string <code>s</code> in which every occurrence of the substring <code>old</code> is replaced by the string <code>new</code> .



- What is the length of the string "Python Program"?
- Given this string variable, give a method call that returns the string "gram".
`title = "Python Program"`
- Use string concatenation to turn the string variable `title` from Self Check 19 into "Python Programming".
- What does the following statement sequence print?

```
string = "Harry"
n = len(string)
mystery = string[0] + string[n - 1]
print(mystery)
```

Practice It Now you can try these exercises at the end of the chapter: R2.7, R2.11, P2.15, P2.22.

Special Topic 2.4



Character Values

A character is stored internally as an integer value. The specific value used for a given character is based on a standard set of codes. You can find the values of the characters that are used in Western European languages in Appendix D. For example, if you look up the value for the character "H", you can see that it is actually encoded as the number 72.

Python provides two functions related to character encodings. The `ord` function returns the number used to represent a given character. The `chr` function returns the character associated with a given code. For example,

```
print("The letter H has a code of", ord("H"))
print("Code 97 represents the character", chr(97))
```

produces the following output

```
The letter H has a code of 72
Code 97 represents the character a
```

Special Topic 2.5

Escape Sequences



Sometimes you may need to include both single and double quotes in a literal string. For example, to include double quotes around the word `Welcome` in the literal string "You're Welcome", precede the quotation marks with a backslash (`\`), like this:

```
"You're \"Welcome\""
```

The backslash is not included in the string. It indicates that the quotation mark that follows should be a part of the string and not mark the end of the string. The sequence `\` is called an **escape sequence**.

To include a backslash in a string, use the escape sequence `\\`, like this:

```
"C:\\Temp\\Secret.txt"
```

Another common escape sequence is `\n`, which denotes a **newline** character. Printing a newline character causes the start of a new line on the display. For example, the statement

```
print("*\n**\n***")
```

prints the characters

```
*
**
***
```

on three separate lines.



Computing & Society 2.1 International Alphabets and Unicode

The English alphabet is pretty simple: upper- and lowercase *a* to *z*. Other European languages have accent marks and special characters. For example, German has three so-called *umlaut* characters, *ä*, *ö*, *ü*, and a *double-s* character *ß*. These are not optional frills; you couldn't write a page of German text without using these characters a few times. German keyboards have keys for these characters.

© pvachieri/iStockphoto.



The German Keyboard Layout

Many countries don't use the Roman script at all. Russian, Greek, Hebrew, Arabic, and Thai letters, to name just a few, have completely different shapes. To complicate matters, Hebrew and Arabic are typed from right to left. Each of these alphabets has about as many characters as the English alphabet.



Hebrew, Arabic, and English

The Chinese languages as well as Japanese and Korean use Chinese characters. Each character represents an

idea or thing. Words are made up of one or more of these ideographic characters. Over 70,000 ideographs are known.

Starting in 1988, a consortium of hardware and software manufacturers developed a uniform encoding scheme called **Unicode** that is capable of encoding text in essentially all written languages of the world.

Today Unicode defines over 100,000 characters. There are even plans to add codes for extinct languages, such as Egyptian hieroglyphics.



The Chinese Script

2.5 Input and Output

Most interesting programs ask the program user to provide input values, then the programs produce outputs that depend on the user input. In the following sections, you will see how to read user input and how to control the appearance of the output that your programs produce.

2.5.1 User Input

You can make your programs more flexible if you ask the program user for inputs rather than using fixed values. Consider, for example, the `initials.py` program from Section 2.4.4 that prints a pair of initials. The two names from which the initials are derived are specified as literal values. If the program user entered the names as inputs, the program could be used for any pair of names.

When a program asks for user input, it should first print a message that tells the user which input is expected. Such a message is called a **prompt**. In Python, displaying a prompt and reading the keyboard input is combined in one operation.

```
first = input("Enter your first name: ")
```

The `input` function displays the string argument in the console window and places the cursor on the same line, immediately following the string.

```
Enter your first name: █
```

Note the space between the colon and the cursor. This is common practice in order to visually separate the prompt from the input. After the prompt is displayed, the program waits until the user types a name. After the user supplies the input,

```
Enter your first name: Rodolfo█
```

the user presses the Enter key. Then the sequence of characters is returned from the `input` function as a string. In our example, we store the string in the variable `first` so it can be used later. The program then continues with the next statement.

The following version of the `initials.py` program is changed to obtain the two names from the user.

ch02/sec05/initials2.py

```
1 ##
2 # This program obtains two names from the user and prints a pair of initials.
3 #
4
5 # Obtain the two names from the user.
6 first = input("Enter your first name: ")
7 second = input("Enter your significant other's first name: ")
8
9 # Compute and display the initials.
10 initials = first[0] + "&" + second[0]
11 print(initials)
```

Program Run

```
Enter your first name: Rodolfo
Enter your significant other's first name: Sally
R&S
```

Use the `input` function to read keyboard input.

2.5.2 Numerical Input

The input function can only obtain a string of text from the user. But what if we need to obtain a numerical value? Consider, for example, a program that asks for the price and quantity of soda containers. To compute the total price, the number of soda containers needs to be an integer value, and the price per container needs to be a floating-point value.

To read an integer value, first use the input function to obtain the data as a string, then convert it to an integer using the int function.

```
userInput = input("Please enter the number of bottles: ")
bottles = int(userInput)
```

In this example, userInput is a temporary variable that is used to store the string representation of the integer value (see Figure 5). After the input string is converted to an integer value and stored in bottles, it is no longer needed.

To read a floating-point value from the user, the same approach is used, except the input string has to be converted to a float.

```
userInput = input("Enter price per bottle: ")
price = float(userInput)
```

To read an integer or floating-point value, use the input function followed by the int or float function.

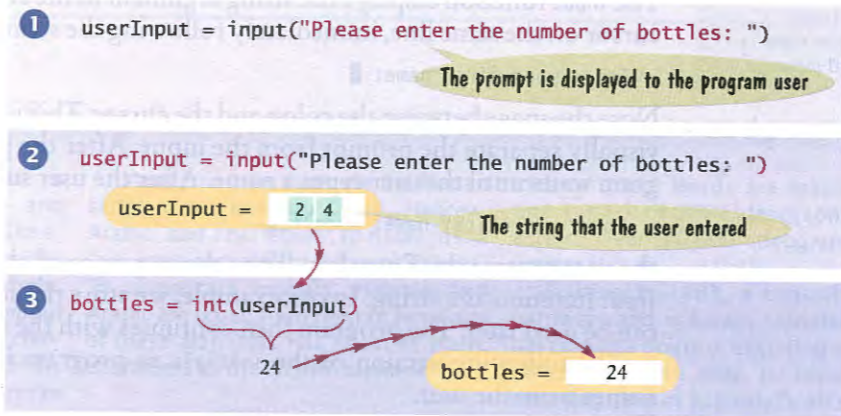


Figure 5 Extracting an Integer Value

2.5.3 Formatted Output

When you print the result of a computation, you often want to control its appearance. For example, when you print an amount in dollars and cents, you usually want it to be rounded to two significant digits. That is, you want the output to look like

```
Price per liter: 1.22
```

instead of

```
Price per liter: 1.215962441314554
```

The following command displays the price with two digits after the decimal point:

```
print("%.2f" % price) # Prints 1.22
```

You can also specify a field width (the total number of characters, including spaces), like this:

```
print("%10.2f" % price)
```

Syntax 2.3 String Format Operator

Syntax `formatString % (value1, value2, ..., valuen)`

The format string can contain one or more format specifiers and literal characters.

No parentheses are needed to format a single value.

```
print("Quantity: %d Total: %10.2f" % (quantity, total))
```

It is common to print a formatted string.

Format specifiers

The values to be formatted. Each value replaces one of the format specifiers in the resulting string.

The price is printed right-justified using ten characters: six spaces followed by the four characters 1.22.



The argument passed to the print function

```
"%10.2f" % price
```

specifies how the string is to be formatted. The result is a string that can be printed or stored in a variable.

You learned earlier that the % symbol is used to compute the remainder of floor division, but that is only the case when the values left and right of the operator are both numbers. If the value on the left is a string, then the % symbol becomes the **string format operator**.

The construct `%10.2f` is called a *format specifier*: it describes how a value should be formatted. The letter `f` at the end of the format specifier indicates that we are formatting a floating-point value. Use `d` for an integer value and `s` for a string; see Table 9 on page 57 for examples.

The *format string* (the string on the left side of the string format operator) can contain one or more format specifiers and literal characters. Any characters that are not format specifiers are included verbatim. For example, the command

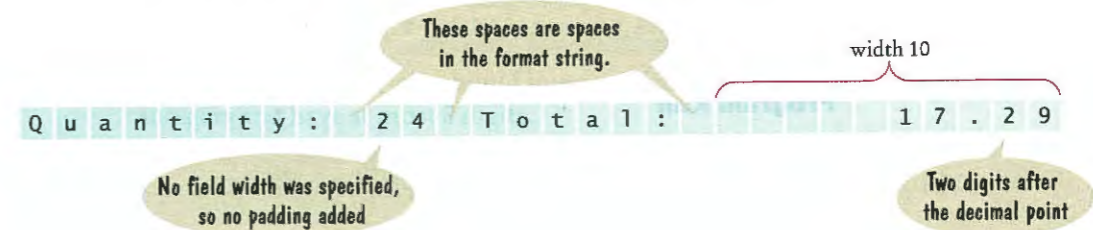
```
"Price per liter:%10.2f" % price
```

produces the string

```
"Price per liter: 1.22"
```

You can format multiple values with a single string format operation, but you must enclose them in parentheses and separate them by commas. Here is a typical example:

```
print("Quantity: %d Total: %10.2f" % (quantity, total))
```



Use the string format operator to specify how values should be formatted.

The values to be formatted (quantity and total in this case) are used in the order listed. That is, the first value is formatted based on the first format specifier (%d), the second value (stored in total) is based on the second format specifier (%10.2f), and so on.

When a field width is specified, the values are right-justified within the given number of columns. While this is the common layout used with numerical values printed in table format, it's not the style used with string data. For example, the statements

```
title1 = "Quantity:"
title2 = "Price:"
print("%10s %10d" % (title1, 24))
print("%10s %10.2f" % (title2, 17.29))
```

result in the following output:

```
Quantity:      24
Price:       17.29
```

The output would look nicer, however, if the titles were left-justified. To specify left justification, add a minus sign before the string field width:

```
print("%-10s %10d" % (title1, 24))
print("%-10s %10.2f" % (title2, 17.29))
```

The result is the far more pleasant

```
Quantity:      24
Price:       17.29
```

Our next example program will prompt for the price of a six-pack and the volume of each can, then print out the price per ounce. The program puts to work what you just learned about reading input and formatting output.

ch02/sec05/volume2.py

```
1 ##
2 # This program prints the price per ounce for a six-pack of cans.
3 #
4
5 # Define constant for pack size.
6 CANS_PER_PACK = 6
7
8 # Obtain price per pack and can volume.
9 userInput = input("Please enter the price for a six-pack: ")
10 packPrice = float(userInput)
11
12 userInput = input("Please enter the volume for each can (in ounces): ")
13 canVolume = float(userInput)
14
15 # Compute pack volume.
16 packVolume = canVolume * CANS_PER_PACK
17
18 # Compute and print price per ounce.
19 pricePerOunce = packPrice / packVolume
20 print("Price per ounce: %8.2f" % pricePerOunce)
```

Program Run

```
Please enter the price for a six-pack: 2.95
Please enter the volume for each can (in ounces): 12
Price per ounce: 0.04
```

Table 9 Format Specifier Examples

Format String	Sample Output	Comments
"%d"	2 4	Use d with an integer.
"%5d"	2 4	Spaces are added so that the field width is 5.
"%05d"	0 0 0 2 4	If you add 0 before the field width, zeroes are added instead of spaces.
"Quantity:%5d"	Q u a n t i t y : 2 4	Characters inside a format string but outside a format specifier appear in the output.
"%f"	1 . 2 1 9 9 7	Use f with a floating-point number.
"%.2f"	1 . 2 2	Prints two digits after the decimal point.
"%7.2f"	1 . 2 2	Spaces are added so that the field width is 7.
"%s"	H e l l o	Use s with a string.
"%d %.2f"	2 4 1 . 2 2	You can format multiple values at once.
"%9s"	H e l l o	Strings are right-justified by default.
"%-9s"	H e l l o	Use a negative field width to left-justify.
"%d%%"	2 4 %	To add a percent sign to the output, use %.



22. Write statements to prompt for and read the user's age.
23. What is problematic about the following statement sequence?


```
userInput = input("Please enter the unit price: ")
unitPrice = int(userInput)
```
24. What is problematic about the following statement sequence?


```
userInput = input("Please enter the number of cans")
cans = int(userInput)
```
25. What is the output of the following statement sequence?


```
volume = 10
print("The total volume is %5d" % volume)
```
26. Using the string format operator, print the values of the variables bottles and cans so that the output looks like this:


```
Bottles:      8
Cans:         24
```

The numbers to the right should line up. (You may assume that the numbers are integers and have at most 8 digits.)

Practice It Now you can try these exercises at the end of the chapter: R2.10, P2.6, P2.7.

Programming Tip 2.4

**Don't Wait to Convert**

When obtaining numerical values from input, you should convert the string representation to the corresponding numerical value immediately after the input operation.

Obtain the string and save it in a temporary variable that is then converted to a number by the next statement. Don't save the string representation and convert it to a numerical value every time it's needed in a computation:

```
unitPrice = input("Enter the unit price: ")
price1 = float(unitPrice)
price2 = 12 * float(unitPrice) # Bad style
```

It is bad style to repeat the same computation multiple times. And if you wait, you could forget to perform the conversion.

Instead, convert the string input immediately to a number:

```
unitPriceInput = input("Enter the unit price: ")
unitPrice = float(unitPriceInput) # Do this immediately after reading the input
price1 = unitPrice
price2 = 12 * unitPrice
```

Or, even better, combine the calls to `input` and `float` in a single statement:

```
unitPrice = float(input("Enter the unit price: "))
```

The string returned by the `input` function is passed directly to the `float` function, not saved in a variable.

HOW TO 2.1

Writing Simple Programs

This How To shows you how to turn a problem statement into pseudocode and, ultimately, a Python program.

Problem Statement Write a program that simulates a vending machine. A customer selects an item for purchase and inserts a bill into the vending machine. The vending machine dispenses the purchased item and gives change. Assume that all item prices are multiples of 25 cents, and the machine gives all change in dollar coins and quarters. Your task is to compute how many coins of each type to return.

Step 1 Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there are two inputs:

- The denomination of the bill that the customer inserts
- The price of the purchased item



A vending machine takes bills and gives change in coins.

Jupiterimages/Getty Images, Inc.

There are two desired outputs:

- The number of dollar coins that the machine returns
- The number of quarters that the machine returns

Step 2 Work out examples by hand.

This is a very important step. If you can't compute a couple of solutions by hand, it's unlikely that you'll be able to write a program that automates the computation.

Let's assume that a customer purchased an item that cost \$2.25 and inserted a \$5 bill. The customer is due \$2.75, or two dollar coins and three quarters, in change.

That is easy for you to see, but how can a Python program come to the same conclusion? The key is to work in pennies, not dollars. The change due the customer is 275 pennies. Dividing by 100 yields 2, the number of dollars. Dividing the remainder (75) by 25 yields 3, the number of quarters.

Step 3 Write pseudocode for computing the answers.

In the previous step, you worked out a specific instance of the problem. You now need to come up with a method that works in general.

Given an arbitrary item price and payment, how can you compute the coins due? First, compute the change due in pennies:

```
change due = 100 x bill value - item price in pennies
```

To get the dollars, divide by 100 and discard the fractional part:

```
num dollar coins = change due divided by 100 (without the fractional part)
```

If you prefer, you can use the Python symbol for floor division.

```
num dollar coins = change due // 100
```

But you don't have to. The purpose of pseudocode is to describe the computation in a humanly readable form, not to use the syntax of a particular programming language.

The remaining change due can be computed in two ways. If you are aware that one can compute the remainder of a floor division (in Python, with the modulus operator), you can simply compute

```
change due = remainder of dividing change due by 100
```

Alternatively, subtract the penny value of the dollar coins from the change due:

```
change due = change due - 100 x num dollar coins
```

To get the quarters due, divide by 25:

```
num quarters = change due // 25
```

Step 4 Declare the variables and constants that you need, and decide what types of values they hold.

Here, we have five variables:

- billValue
- itemPrice
- changeDue
- dollarCoins
- quarters

Should we introduce constants to explain 100 and 25 as `PENNIES_PER_DOLLAR` and `PENNIES_PER_QUARTER`? Doing so will make it easier to convert the program to international markets, so we will take this step.

Because we use floor division and the modulus operator, we want all values to be integers.

Step 5 Turn the pseudocode into Python statements.

If you did a thorough job with the pseudocode, this step should be easy. Of course, you have to know how to express mathematical operations (such as floor division and modulus) in Python.

```
changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice
dollarCoins = changeDue // PENNIES_PER_DOLLAR
changeDue = changeDue % PENNIES_PER_DOLLAR
quarters = changeDue // PENNIES_PER_QUARTER
```

Step 6 Provide input and output.

Before starting the computation, we prompt the user for the bill value and item price:

```
userInput = input("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ")
billValue = int(userInput)
userInput = input("Enter item price in pennies: ")
itemPrice = int(userInput)
```

When the computation is finished, we display the result. For extra credit, we format the output strings to make sure that the output lines up neatly:

```
print("Dollar coins: %6d" % dollarCoins)
print("Quarters:    %6d" % quarters)
```

Step 7 Provide a Python program.

Your computation needs to be placed into a program. Find a name for the program that describes the purpose of the computation. In our example, we will choose the name `vending`.

In the program, you need to declare constants and variables (Step 4), carry out computations (Step 5), and provide input and output (Step 6). Clearly, you will want to first get the input, then do the computations, and finally show the output. Define the constants at the beginning of the program, and define each variable just before it is needed.

Here is the complete program:

ch02/how_to_1/vending.py

```
1  ##
2  # This program simulates a vending machine that gives change.
3  #
4
5  # Define constants.
6  PENNIES_PER_DOLLAR = 100
7  PENNIES_PER_QUARTER = 25
8
9  # Obtain input from user.
10 userInput = input("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ")
11 billValue = int(userInput)
12 userInput = input("Enter item price in pennies: ")
13 itemPrice = int(userInput)
14
15 # Compute change due.
16 changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice
17 dollarCoins = changeDue // PENNIES_PER_DOLLAR
18 changeDue = changeDue % PENNIES_PER_DOLLAR
19 quarters = changeDue // PENNIES_PER_QUARTER
20
21 # Print change due.
22 print("Dollar coins: %6d" % dollarCoins)
23 print("Quarters:    %6d" % quarters)
```

Program Run

```
Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): 5
Enter item price in pennies: 225
Dollar coins:    2
Quarters:       3
```

WORKED EXAMPLE 2.2**Computing the Cost of Stamps**

Problem Statement Simulate a postage stamp vending machine. A customer inserts dollar bills into the vending machine and then pushes a “purchase” button. The vending machine gives out as many first-class stamps as the customer paid for, and returns the change in penny (one-cent) stamps. A first-class stamp cost 44 cents at the time this book was written.

Step 1 Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there is one input:

- The amount of money the customer inserts

There are two desired outputs:

- The number of first-class stamps the machine returns
- The number of penny stamps the machine returns

Step 2 Work out examples by hand.

Let’s assume that a first-class stamp costs 44 cents and the customer inserts \$1.00. That’s enough for two stamps (88 cents) but not enough for three stamps (\$1.32). Therefore, the machine returns two first-class stamps and 12 penny stamps.

Step 3 Write pseudocode for computing the answers.

Given an amount of money and the price of a first-class stamp, how can you compute how many first-class stamps can be purchased with the money? Clearly, the answer is related to the quotient

$$\frac{\text{amount of money}}{\text{price of first-class stamp}}$$

For example, suppose the customer paid \$1.00. Use a pocket calculator to compute the quotient: $\$1.00/\$0.44 = 2.27$.

How do you get “2 stamps” out of 2.27? It’s the quotient without the fractional part. In Python, this is easy to compute if both arguments are integers. Therefore, let’s switch our computation to pennies. Then we have

$$\text{number of first-class stamps} = 100 / 44 \text{ (without remainder)}$$

What if the user inputs two dollars? Then the numerator becomes 200. What if the price of a stamp goes up? A more general equation is

$$\text{number of first-class stamps} = 100 \times \text{dollars} / \text{price of first-class stamps in cents (without remainder)}$$

How about the change? Here is one way of computing it. When the customer gets the stamps, the change is the customer payment, reduced by the value of the stamps purchased. In our example, the change is 12 cents—the difference between 100 and $2 \cdot 44$. Here is the general formula:

$$\text{change} = 100 \times \text{dollars} - \text{number of first-class stamps} \times \text{price of first-class stamp}$$

Step 4 Define the variables and constants that you need, and decide what types of values they hold.

Here, we have three variables:

- dollars
- firstClassStamps
- change

There is one constant, `FIRST_CLASS_STAMP_PRICE`. By using a constant, we can change the price in one place without having to search and replace every occurrence of 44 used as the stamp price in the program.

The variable `dollars` and constant `FIRST_CLASS_STAMP_PRICE` must be integers because the computation of `firstClassStamps` uses floor division. The remaining variables are also integers, counting the number of first-class and penny stamps.

Step 5 Turn the pseudocode into Python statements.

Our computation depends on the number of dollars that the user provides. Translating the math into Python yields the following statements:

```
firstClassStamps = 100 * dollars // FIRST_CLASS_STAMP_PRICE
change = 100 * dollars - firstClassStamps * FIRST_CLASS_STAMP_PRICE
```

Step 6 Provide input and output.

Before the computation, we prompt the user for the number of dollars and obtain the value:

```
dollarStr = input("Enter number of dollars: ")
dollars = int(dollarStr)
```

When the computation is finished, we display the result.

```
print("First class stamps: %6d" % firstClassStamps)
print("Penny stamps:      %6d" % change)
```

Step 7 Write a Python program.

Here is the complete program:

ch02/worked_example_2/stamps.py

```
1 ##
2 # This program simulates a stamp machine that receives dollar bills and
3 # dispenses first class and penny stamps.
4 #
5
6 # Define the price of a stamp in pennies.
7 FIRST_CLASS_STAMP_PRICE = 44
8
9 # Obtain the number of dollars.
10 dollarStr = input("Enter number of dollars: ")
11 dollars = int(dollarStr)
12
13 # Compute and print the number of stamps to dispense.
14 firstClassStamps = 100 * dollars // FIRST_CLASS_STAMP_PRICE
15 change = 100 * dollars - firstClassStamps * FIRST_CLASS_STAMP_PRICE
16 print("First class stamps: %6d" % firstClassStamps)
17 print("Penny stamps:      %6d" % change)
```

Program Run

```
Enter number of dollars: 4
First class stamps:    9
Penny stamps:         4
```



Computing & Society 2.2 The Pentium Floating-Point Bug

In 1994, Intel Corporation released what was then its most powerful processor, the Pentium. Unlike previous generations of its processors, it had a very fast floating-point unit. Intel's goal was to compete aggressively with the makers of higher-end processors for engineering workstations. The Pentium was a huge success immediately.

In the summer of 1994, Dr. Thomas Nicely of Lynchburg College in Virginia ran an extensive set of computations to analyze the sums of reciprocals of certain sequences of prime numbers. The results were not always what his theory predicted, even after he took into account the inevitable roundoff errors. Then Dr. Nicely noted that the same program did produce the correct results when running on the slower 486 processor that preceded the Pentium in Intel's lineup. This should not have happened. The optimal roundoff behavior of floating-point calculations has been standardized by the Institute for Electrical and Electronics Engineers (IEEE) and Intel claimed to adhere to the IEEE standard in both the 486 and the Pentium processors. Upon further checking, Dr. Nicely discovered that indeed there was a very small set of numbers for which the product of two numbers was computed differently on the two processors. For example,

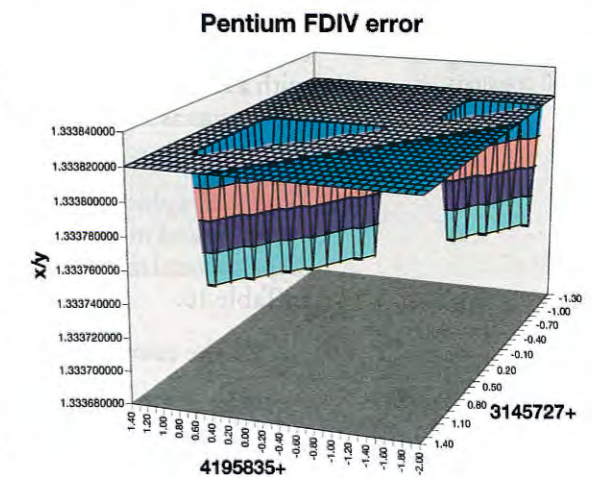
$$4,195,835 - ((4,195,835/3,145,727) \times 3,145,727)$$

is mathematically equal to 0, and it did compute as 0 on a 486 processor. On his Pentium processor the result was 256.

As it turned out, Intel had independently discovered the bug in its testing and had started to produce chips that fixed it. The bug was caused by an error in a table that was used to speed up the floating-point multiplication algorithm of the processor. Intel determined that the problem was exceedingly rare. They claimed that under normal use, a typical consumer would only notice the problem once every 27,000 years. Unfortunately for Intel, Dr. Nicely had not been a normal user.

Now Intel had a real problem on its hands. It figured that the cost of replacing all Pentium processors that it had sold so far would cost a great deal of money. Intel already had more orders for the chip than it could produce, and it would be particularly galling to have to give out the scarce chips as free replacements instead of selling them. Intel's management decided to punt on the issue and initially offered to replace the processors only for those customers who could prove that their work required absolute precision in mathematical calculations. Naturally, that did not go over well with the hundreds of thousands of customers who had paid retail prices of \$700 and more for a Pentium chip and did not want to live with the nagging feeling that perhaps, one day, their income tax program would produce a faulty return.

Ultimately, Intel caved in to public demand and replaced all defective chips, at a cost of about 475 million dollars.



This graph shows a set of numbers for which the original Pentium processor obtained the wrong quotient.

2.6 Graphics: Simple Drawings

There are times when you may want to include simple drawings such as figures, graphs, or charts in your programs. Although the Python library provides a module for creating full graphical applications, it is beyond the scope of this book.

To help you create simple drawings, we have included an `ezgraphics` module with the book that is a simplified version of Python's more complex library module. The module code and usage instructions are included with the source code on this book's companion web site. In the following sections, you will learn all about this module, and how to use it to create simple drawings that consist of basic geometric shapes and text.



You can make simple drawings out of lines, rectangles, and circles.

2.6.1 Creating a Window

A graphical application shows information inside a **window** on the desktop with a rectangular area and a title bar, as shown in Figure 6. In the `ezgraphics` module, this window is called a *graphics window*.

To create a graphical application using the `ezgraphics` module, carry out the following:

1. Import the `GraphicsWindow` class:

```
from ezgraphics import GraphicsWindow
```

As you will see in Chapter 9, a class defines the behavior of its objects. We will create a single object of the `GraphicsWindow` class and call methods on it.

2. Create a graphics window:

```
win = GraphicsWindow()
```

The new window will automatically be shown on the desktop and contain a canvas that is 400 pixels wide by 400 pixels tall. To create a graphics window with a canvas that is of a specific size, you can specify the width and height of the canvas as arguments:

```
win = GraphicsWindow(500, 500)
```

When a graphics window is created, the object representing the window is returned and must be stored in a variable, as it will be needed in the following steps. Several methods that can be used with a `GraphicsWindow` object are shown in Table 10.

3. Access the canvas contained in the graphics window:

```
canvas = win.canvas()
```

To create a drawing, you draw the geometric shapes on a canvas just as an artist would to create a painting. An object of the `GraphicsCanvas` class is automatically

A graphics window is used for creating graphical drawings.

Geometric shapes and text are drawn on a canvas that is contained in a graphics window.

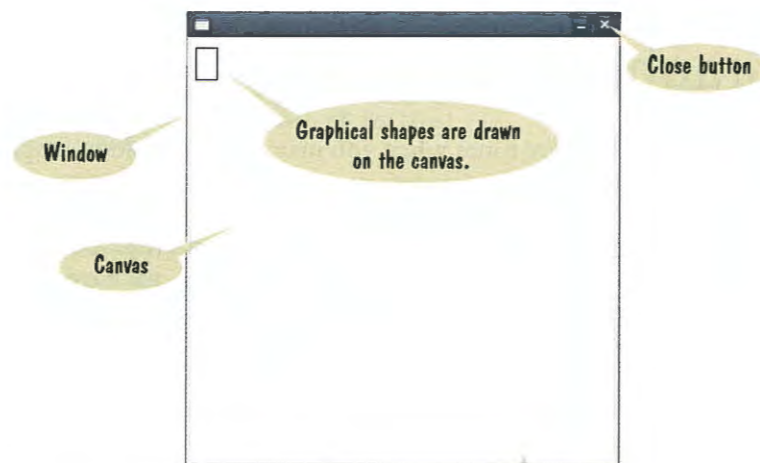


Figure 6 A Graphics Window

created when you create the `GraphicsWindow` object. The `canvas` method gives you access to the object representing that canvas. It will be used in the next step.

4. Create your drawing.

Geometric shapes and text are drawn on the canvas using methods defined in the `GraphicsCanvas` class. These methods will be described in the following sections. For now, we'll draw a rectangle:

```
canvas.drawRect(15, 10, 20, 30)
```

5. Wait for the user to close the graphics window:

```
win.wait()
```

After drawing the scene on the canvas, the program has to stop or pause and wait for the user to close the window (by clicking the close button). Without this statement, the program would terminate immediately and the graphics window would disappear, leaving no time for you to see your drawing.

The simple program below produces the graphics window shown in Figure 6.

ch02/sec06/window.py

```
1 ##
2 # This program creates a graphics window with a rectangle. It provides the
3 # template used with all of the graphical programs used in the book.
4 #
5
6 from ezgraphics import GraphicsWindow
7
8 # Create the window and access the canvas.
9 win = GraphicsWindow()
10 canvas = win.canvas()
11
12 # Draw on the canvas.
13 canvas.drawRect(5, 10, 20, 30)
14
15 # Wait for the user to close the window.
16 win.wait()
```

Table 10 `GraphicsWindow` Methods

Method	Description
<code>w = GraphicsWindow()</code> <code>w = GraphicsWindow(width, height)</code>	Creates a new graphics window with an empty canvas. The size of the canvas is 400 × 400 unless another size is specified.
<code>w.canvas()</code>	Returns the object representing the canvas contained in the graphics window.
<code>w.wait()</code>	Keeps the graphics window open and waits for the user to click the "close" button.

2.6.2 Lines and Polygons

The canvas has methods for drawing lines, rectangles, and other shapes.

To draw a shape on the canvas, you call one of the “draw” methods defined for a canvas. The call

```
canvas.drawLine(x1, y1, x2, y2)
```

draws a line on the canvas between the points (x_1, y_1) and (x_2, y_2) . The call

```
canvas.drawRect(x, y, width, height)
```

draws a rectangle that has its upper-left corner positioned at (x, y) and the given width and height.

Geometric shapes and text are drawn on a canvas by specifying points in the two-dimensional discrete Cartesian coordinate system. The coordinate system, however, is different from the one used in mathematics. The origin $(0, 0)$ is at the upper-left corner of the canvas and the y -coordinate grows downward.

The points on the canvas correspond to pixels on the screen. Thus, the actual size of the canvas and the geometric shapes depends on the resolution of your screen.

Here is the code for a simple program that draws the bar chart shown in Figure 7.

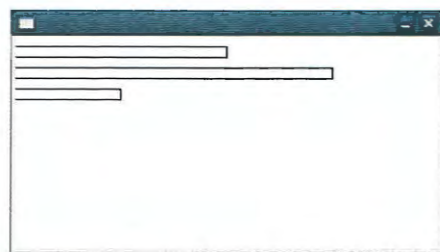
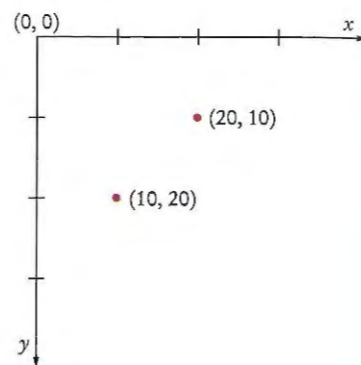


Figure 7
Drawing a Bar Chart

ch02/sec06/barchart1.py

```
1 ##
2 # This program draws three rectangles on a canvas.
3 #
4
5 from ezgraphics import GraphicsWindow
6
7 # Create the window and access the canvas.
8 win = GraphicsWindow(400, 200)
9 canvas = win.canvas()
10
11 # Draw on the canvas.
12 canvas.drawRect(0, 10, 200, 10)
13 canvas.drawRect(0, 30, 300, 10)
14 canvas.drawRect(0, 50, 100, 10)
15
16 # Wait for the user to close the window.
17 win.wait()
```

2.6.3 Filled Shapes and Color

The canvas stores the current drawing parameters used to draw shapes and text.

The canvas stores the drawing parameters (the current color, font, line width, and so on) that are used for drawing shapes and text. When you first start drawing on a canvas, all shapes are drawn using a black pen.

To change the pen color, use one of the method calls,

```
canvas.setOutline(red, green, blue)
canvas.setOutline(colorName)
```

The method arguments can be integer values between 0 and 255 that specify a color value, or one of the strings describing a color in Table 11.

For example, to draw a red rectangle, call

```
canvas.setOutline(255, 0, 0)
canvas.drawRect(10, 20, 100, 50)
```

or

```
canvas.setOutline("red")
canvas.drawRect(10, 20, 100, 50)
```

The geometric shapes can be drawn in one of three styles—outlined, filled, or outlined and filled.



The style used to draw a specific shape depends on the current *fill color* and *outline color* as set in the canvas. If you use the default setting (not changing the fill or outline), shapes are outlined in black and there is no fill color.

To set the fill color, use one of the method calls

```
canvas.setFill(red, green, blue)
canvas.setFill(colorName)
```

The following statements

```
canvas.setOutline("black")
canvas.setFill(0, 255, 0)
canvas.drawRect(10, 20, 100, 50)
```

draw a rectangle that is outlined in black and filled with green:



Table 11 Common Color Names

Color Name	Color Name	Color Name	Color Name
"black"	"magenta"	"maroon"	"pink"
"blue"	"yellow"	"dark blue"	"orange"
"red"	"white"	"dark red"	"sea green"
"green"	"gray"	"dark green"	"light gray"
"cyan"	"gold"	"dark cyan"	"tan"

Table 12 GraphicsCanvas Color Methods

Method	Description
<code>c.setColor(colorName)</code> <code>c.setColor(red, green, blue)</code>	Sets both the fill and outline color to the same color. Color can be set by the <i>colorName</i> or by values for its <i>red</i> , <i>green</i> , and <i>blue</i> components. (See Section 4.10 for more about RGB values.)
<code>c.setFill()</code> <code>c.setFill(colorName)</code> <code>c.setFill(red, green, blue)</code>	Sets the color used to fill a geometric shape. If no argument is given, the fill color is cleared.
<code>c.setOutline()</code> <code>c.setOutline(colorName)</code> <code>c.setOutline(red, green, blue)</code>	Sets the color used to draw lines and text. If no argument is given, the outline color is cleared.

To fill without an outline, call the `setOutline` method with no arguments:

```
canvas.setOutline() # Clears the outline color
```

You can also clear the fill color by calling the `setFill` method with no arguments. This is necessary if you set a fill color in order to draw a filled shape, but then would like to draw an unfilled shape.

Finally, you can set both fill and outline color to the same color with the `setColor` method. For example, the call

```
canvas.setColor("red")
```

sets both the fill and outline color to red.

The following program is a version of the `barchart1.py` program modified to create three filled rectangles, as shown in Figure 8.

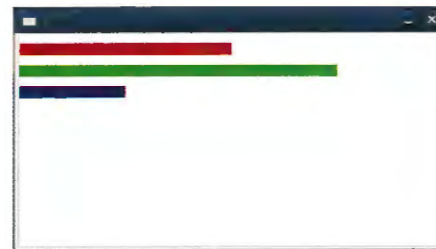


Figure 8
Drawing a Bar Chart with Color Bars

ch02/sec06/barchart2.py

```
1 ##
2 # This program draws three colored rectangles on a canvas.
3 #
4
5 from ezgraphics import GraphicsWindow
6
7 # Create the window and access the canvas.
8 win = GraphicsWindow(400, 200)
9 canvas = win.canvas()
10
11 # Draw on the canvas.
12 canvas.setColor("red")
```

```
13 canvas.drawRect(0, 10, 200, 10)
14
15 canvas.setColor("green")
16 canvas.drawRect(0, 30, 300, 10)
17
18 canvas.setColor("blue")
19 canvas.drawRect(0, 50, 100, 10)
20
21 # Wait for the user to close the window.
22 win.wait()
```

2.6.4 Ovals, Circles, and Text

Now that you've learned how to draw lines and rectangles, let's turn to additional graphical elements.

To draw an oval, you specify its *bounding box* (see Figure 9) in the same way that you would specify a rectangle, namely by the *x*- and *y*-coordinates of the top-left corner and the width and height of the box. To draw an oval, use the method call

```
canvas.drawOval(x, y, width, height)
```

As with a rectangle, the oval will be drawn filled, with an outline, or both depending on the current drawing context. To draw a circle, set the width and height to the same values:

```
canvas.drawOval(x, y, diameter, diameter)
```

Notice that (x, y) is the top-left corner of the bounding box, not the center of the circle.

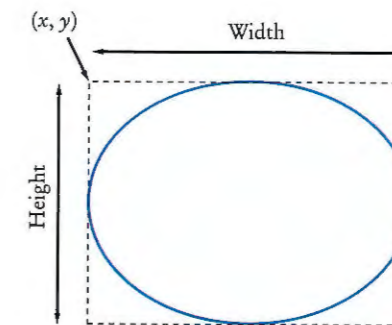


Figure 9 An Oval and its Bounding Box

You often want to put text inside a drawing, for example, to label some of the parts. Use the canvas method `drawText` to draw a string anywhere on a canvas. You must specify the string and the *x*- and *y*-coordinates of the top-left corner of the bounding box (the "anchor point"—see Figure 10). For example

```
canvas.drawText(50, 100, "Message")
```

Figure 10
Bounding Box and Anchor Point



Table 13 GraphicsCanvas Drawing Methods





Method	Result	Notes
<code>c.drawLine(x₁, y₁, x₂, y₂)</code>		(x ₁ , y ₁) and (x ₂ , y ₂) are the endpoints.
<code>c.drawRect(x, y, width, height)</code>		(x, y) is the top-left corner.
<code>c.drawOval(x, y, width, height)</code>		(x, y) is the top-left corner of the box that bounds the ellipse. To draw a circle, use the same value for width and height.
<code>c.drawText(x, y, text)</code>		(x, y) is the anchor point.

Table 13 provides a list of drawing methods available for use with the canvas.



27. How do you modify the program in Section 2.6.2 to draw two squares?
28. What happens if you call `drawOval` instead of `drawRect` in the program of Section 2.6.2?
29. Give instructions to draw a circle with center (100, 100) and radius 25.
30. Give instructions to draw a letter "V" by drawing two line segments.
31. Give instructions to draw a string consisting of the letter "V".
32. How do you draw a yellow square on a red background?

Practice It Now you can try these exercises at the end of the chapter: P2.23, P2.24, P2.25.

HOW TO 2.2

Graphics: Drawing Graphical Shapes



Suppose you want to write a program that displays graphical shapes such as cars, aliens, charts, or any other images that can be obtained from rectangles, lines, and ellipses. These instructions give you a step-by-step procedure for decomposing a drawing into parts and implementing a program that produces the drawing.

Problem Statement Create a program to draw a national flag.

Step 1 Determine the shapes that you need for the drawing.

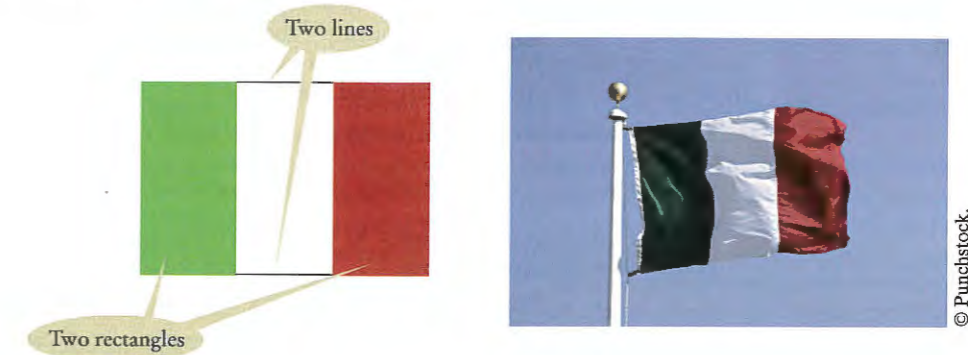
You can use the following shapes:

- Squares and rectangles
- Circles and ovals
- Lines

The outlines of these shapes can be drawn in any color, and you can fill the insides of these shapes with any color. You can also use text to label parts of your drawing.

Some national flag designs consist of three equally wide sections of different colors, side by side, as in the Italian flag shown below.

You could draw such a flag using three rectangles. But if the middle rectangle is white, as it is, for example, in the flag of Italy (green, white, red), it is easier and looks better to draw a line on the top and bottom of the middle portion:



Step 2 Find the coordinates for the shapes.

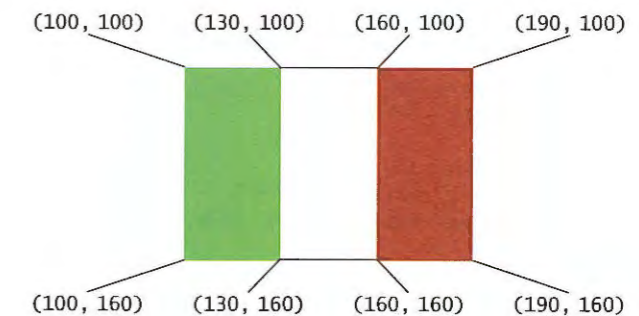
You now need to find the exact positions for the geometric shapes.

- For rectangles, you need the *x*- and *y*-position of the top-left corner, the width, and the height.
- For ellipses, you need the top-left corner, width, and height of the bounding rectangle.
- For lines, you need the *x*- and *y*-positions of the starting point and the end point.
- For text, you need the *x*- and *y*-position of the anchor point.

A commonly-used size for a window is 300 by 300 pixels. You may not want the flag crammed all the way to the top, so perhaps the upper-left corner of the flag should be at point (100, 100).

Many flags, such as the flag of Italy, have a width : height ratio of 3 : 2. (You can often find exact proportions for a particular flag by doing a bit of Internet research on one of several Flags of the World sites.) For example, if you make the flag 90 pixels wide, then it should be 60 pixels tall. (Why not make it 100 pixels wide? Then the height would be $100 \cdot 2 / 3 \approx 67$, which seems more awkward.)

Now you can compute the coordinates of all the important points of the shape:



Step 3 Write Python statements to draw the shapes.

In our example, there are two rectangles and two lines:

```

canvas.setColor("green")
canvas.drawRect(100, 100, 30, 60)

canvas.setColor("red")
canvas.drawRect(160, 100, 30, 60)
    
```

```

canvas.setColor("black")
canvas.drawLine(130, 100, 160, 100)
canvas.drawLine(130, 160, 160, 160)

```

If you are more ambitious, then you can express the coordinates in terms of a few variables. In the case of the flag, we have arbitrarily chosen the top-left corner and the width. All other coordinates follow from those choices. If you decide to follow the ambitious approach, then the rectangles and lines are determined as follows:

```

canvas.drawRect(xLeft, yTop, width / 3, width * 2 / 3)
...
canvas.drawRect(xLeft + 2 * width / 3, yTop, width / 3, width * 2 / 3)
...
canvas.drawLine(xLeft + width / 3, yTop, xLeft + width * 2 / 3, yTop)
canvas.drawLine(xLeft + width / 3, yTop + width * 2 / 3,
                xLeft + width * 2 / 3, yTop + width * 2 / 3)

```

Step 4 Write the program that creates the graphics window and includes the drawing instructions at the proper spot in the template.

```

win = GraphicsWindow("The Italian Flag", 300, 300)
canvas = win.canvas()

```

Drawing instructions

```
win.wait()
```

The complete program for drawing the flag is provided below.

ch02/how_to_2/italianflag.py

```

1  ##
2  # This program draws an Italian flag using the ezgraphics module.
3  #
4
5  from ezgraphics import GraphicsWindow
6
7  win = GraphicsWindow(300, 300)
8  canvas = win.canvas()
9
10 # Define variables with the upper-left position and the size.
11 xLeft = 100
12 yTop = 100
13 width = 90
14
15 # Draw the flag.
16 canvas.setColor("green")
17 canvas.drawRect(xLeft, yTop, width / 3, width * 2 / 3)
18
19 canvas.setColor("red")
20 canvas.drawRect(xLeft + 2 * width / 3, yTop, width / 3, width * 2 / 3)
21
22 canvas.setColor("black")
23 canvas.drawLine(xLeft + width / 3, yTop, xLeft + width * 2 / 3, yTop)
24 canvas.drawLine(xLeft + width / 3, yTop + width * 2 / 3,
25                 xLeft + width * 2 / 3, yTop + width * 2 / 3)
26
27 # Wait for the user to close the window.
28 win.wait()

```

TOOLBOX 2.1



Symbolic Processing with SymPy

This is the first of many optional “toolbox” sections in this book. Python is not only a very nice programming language, but it has a large ecosystem of useful packages. If you need to carry out complex computations in a particular problem domain, chances are that someone has put together a library of code that gets you started. There are packages for statistics, drawing graphs and charts, sending e-mail, analyzing web pages, and many other tasks. Many of them are developed by volunteers and are freely available on the Internet.

In this section, you will be introduced to the SymPy package for symbolic mathematics. In Section 2.2, you saw how to use Python to compute the value of mathematical expressions such as $x^2 \sin(x)$ for a particular value of x . The SymPy package can do much more than that. It can give you a plot of the function and compute a wide variety of formulas. If you have taken a calculus course, you know that there is a formula for computing the derivative of a product. SymPy knows these rules and can carry out all the tedious routine manipulations, so that you don't have to. It is like having a calculus course in a box!

Of course, programs that can process math formulas have been around for over fifty years, but SymPy has two great advantages. It is not a separate program, it is one you use within Python. Second, other math programs come with their own programming languages that are different from Python. When you use SymPy, your investment in mastering Python pays off.

Getting Started

Before being able to use a package such as SymPy, it must be installed on your system. Your instructor may have given you specific installation instructions. If not, we recommend that you follow the instructions at <http://horstmann.com/python4everyone/install.html>.

The activities in this section work best when you run them in interactive mode (see Programming Tip 1.1). And if you use the IPython console, you can get a very attractive display of your results. If you follow our installation instructions, you can use the IPython console inside the Spyder IDE.

The functionality of a third-party code package such as SymPy is organized in one or more *modules*. You need to *import* the modules that you need, as described in Special Topic 2.1. Here, we import the entire contents of the *sympy* module:

```
from sympy import *
```

Now we have access to the functions in that module.

Working with Expressions

One useful function is *sympify*, which turns an expression contained in a string into SymPy form. For example,

```
f = sympify("x ** 2 * sin(x)")
```

When you print *f*, you will see

```
x**2*sin(x)
```

What you get is a symbolic expression, not Python code. The letters *x* that you see in the display are not Python variables but *symbols*, a special data type that is manipulated by SymPy. You can see that by displaying *sympify(x * x ** 2)*. The result is

```
x ** 3
```

SymPy knows that $x^2 \cdot x = x^3$.

Alternatively, you can first define the symbolic expression *x* and store it in a variable. It is convenient to name that variable *x* as well. Then use operators and functions to build up a SymPy expression:

```
x = sympify("x")
f = x ** 2 * sin(x)
```


Figure 11
An IPython Notebook
with SymPy Computations

```

IPython
File Edit View Kernel Magic Window Help
In [1]: from sympy import *
In [2]: init_printing()
In [3]: x = sympify("x")
In [4]: f = x ** 2 * sin(x)
In [5]: f
Out[5]:

$$x^2 \sin(x)$$

In [6]: f * x
Out[6]:

$$x^3 \sin(x)$$

In [7]: plot(f)
Out[7]: <sympy.plotting.plot.Plot at 0x7fa3308c520
In [8]:

```

The `sympy` module contains definitions of the mathematical operators and functions for symbolic expressions, so you can combine symbols in the same way as you would Python expressions.

If you use the IPython notebook, you can display results in mathematical notation with the command

```
init_printing()
```

See Figure 11. We will use mathematical notation for the remainder of this section. If you don't have the IPython notebook, everything will work, but you will see the formulas in the plain computer notation.

As you have seen, working with symbols is useful for simplifying algebraic expressions. Here are a few more examples:

```

expand((x - 1) * (x + 1)) # Yields  $x^2 - 1$ 
expand((x - 1) ** 5) # Yields  $x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$ 

```

Solving Equations

SymPy can solve quadratic equations and many others. When you pass an expression to the `solve` method, you get a list of values where the expression equals zero.

```

solve(x**2 + 2 * x - 8) # Yields [-4, 2]
solve(sin(x) - cos(x)) # Yields [-3π/4, π/4]

```

You still need to know something about mathematics to interpret the results that you get. There are infinitely many solutions to the second equation, because you can add π to any solution and get another. SymPy gives you two solutions from which you can derive the others.

SymPy can compute the derivative:

```
diff(f) # Yields  $x^2 \cos(x) + 2x \sin(x)$ 
```

Computing integrals is just as easy:

```
g = integrate(f) #  $-x^2 \cos(x) + 2x \sin(x) + 2 \cos(x)$ 
```

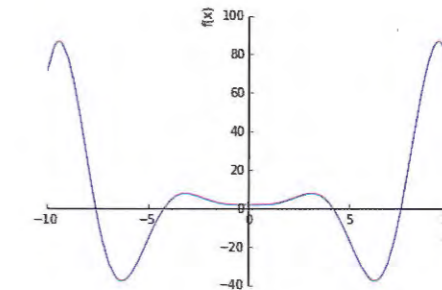
In a typical calculus problem, you often obtain the derivative or integral of a function, and then compute the resulting expression for a given value of x . Substitute a value for the variable with the `subs` method, then turn the expression into a floating-point value with the `evalf` method:

```
result = g.subs(x, 0).evalf() # result is 2.0
```

Finally, you may want to plot a function. Calling `plot` gives you a plot with x ranging from -10 to 10 . For example,

```
plot(-x**2 * cos(x) + 2 * x * sin(x) + 2 * cos(x))
```

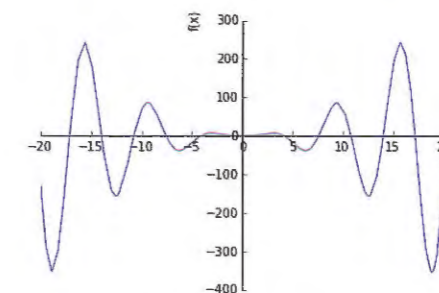
yields



You can provide a different range for x , from -20 to 20 , like this:

```
plot(-x**2 * cos(x) + 2 * x * sin(x) + 2 * cos(x), (x, -20, 20))
```

Here is the result:



If you use the IPython notebook, you can integrate plots into the notebook. If plots are shown in a separate window, use the directive

```
%matplotlib inline
```

As you have seen, SymPy can make quick work of your calculus assignment. It is also a great reason for learning about Python libraries. The creators of SymPy have packaged a great amount of expertise—namely, how to manipulate mathematical symbols—into a form that you can use easily. Other Python packages that we will introduce throughout the book provide expertise from other domains that you can call upon in your programs.

CHAPTER SUMMARY

Declare variables with appropriate names and types.



- A variable is a storage location with a name.
- An assignment statement stores a value in a variable.
- A variable is created the first time it is assigned a value.
- Assigning a value to an existing variable replaces the previously stored value.
- The assignment operator = does *not* denote mathematical equality.
- The data type of a value specifies how the value is stored in the computer and what operations can be performed on the value.



- Integers are whole numbers without a fractional part.
- Floating-point numbers contain a fractional part.
- By convention, variable names should start with a lowercase letter.
- Use constants for values that should remain unchanged throughout your program.
- Use comments to add explanations for humans who read your code. The interpreter ignores comments.



Write arithmetic expressions in Python.



- Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.
- The // operator computes floor division in which the remainder is discarded.
- The % operator computes the remainder of a floor division.
- A function can return a value that can be used as if it were a literal value.
- Python has a standard library that provides functions and data types for your code.
- A library module must be imported into your program before it can be used.

Carry out hand calculations when developing an algorithm.

- Pick concrete values for a typical situation to use in a hand calculation.

Write programs that process strings.



- Strings are sequences of characters.
- A string literal denotes a particular string.
- The len function returns the number of characters in a string.
- Use the + operator to *concatenate* strings; that is, to put them together to yield a longer string.
- A string can be repeated using the * operator.
- The str function converts an integer or floating-point value to a string.

W O R D



- The int and float functions convert a string containing a number to the numerical value.
- String positions are counted starting with 0.

Write programs that read user input and print formatted output.

- Use the input function to read keyboard input.
- To read an integer or floating-point value, use the input function followed by the int or float function.
- Use the string format operator to specify how values should be formatted.

Make simple graphical drawings.



- A graphics window is used for creating graphical drawings.
- Geometric shapes and text are drawn on a canvas that is contained in a graphics window.
- The canvas has methods for drawing lines, rectangles, and other shapes.
- The canvas stores the current drawing parameters used to draw shapes and text.
- Colors can be specified by name or by their red, green, and blue components.

REVIEW EXERCISES

- R2.1 What is the value of `mystery` after this sequence of statements?

```
mystery = 1
mystery = 1 - 2 * mystery
mystery = mystery + 1
```

- R2.2 What is the value of `mystery` after this sequence of statements?

```
mystery = 1
mystery = mystery + 1
mystery = 1 - 2 * mystery
```

- R2.3 Write the following mathematical expressions in Python.

$$s = s_0 + v_0 t + \frac{1}{2} g t^2 \qquad FV = PV \cdot \left(1 + \frac{\text{INT}}{100}\right)^{\text{YRS}}$$

$$G = 4\pi^2 \frac{a^3}{p^2(m_1 + m_2)} \qquad c = \sqrt{a^2 + b^2 - 2ab \cos \gamma}$$

- R2.4 Write the following Python expressions in mathematical notation.

- `dm = m * (sqrt(1 + v / c) / sqrt(1 - v / c) - 1)`
- `volume = pi * r * r * h`
- `volume = 4 * pi * r ** 3 / 3`
- `z = sqrt(x * x + y * y)`

- ■ R2.5 What are the values of the following expressions? In each line, assume that

```
x = 2.5
y = -1.5
m = 18
n = 4
```

- $x + n * y - (x + n) * y$
- $m // n + m \% n$
- $5 * x - n / 5$
- $1 - (1 - (1 - (1 - (1 - n))))$
- $\text{sqrt}(\text{sqrt}(n))$

- R2.6 What are the values of the following expressions, assuming that n is 17 and m is 18?

- $n // 10 + n \% 10$
- $n \% 2 + m \% 2$
- $(m + n) // 2$
- $(m + n) / 2.0$
- $\text{int}(0.5 * (m + n))$
- $\text{int}(\text{round}(0.5 * (m + n)))$

- ■ R2.7 What are the values of the following expressions? In each line, assume that

```
s = "Hello"
t = "World"
```

- $\text{len}(s) + \text{len}(t)$
- $s[1] + s[2]$
- $s[\text{len}(s) // 2]$
- $s + t$
- $t + s$
- $s * 2$

- R2.8 Find at least three *compile-time* errors in the following program.

```
int x = 2
print(x, squared is, x * x)
xcubed = x *** 3
```

- ■ R2.9 Find two *run-time* errors in the following program.

```
from math import sqrt
x = 2
y = 4
print("The product of ", x, "and", y, "is", x + y)
print("The root of their difference is ", sqrt(x - y))
```

- R2.10 Consider the following code segment.

```
purchase = 19.93
payment = 20.00
change = payment - purchase
print(change)
```

The code segment prints the change as 0.07000000000000028. Explain why. Give a recommendation to improve the code so that users will not be confused.

- R2.11 Explain the differences between 2, 2.0, '2', "2", and "2.0".

- R2.12 Explain what each of the following program segments computes.

```
a. x = 2
   y = x + x
b. s = "2"
   t = s + s
```

- ■ R2.13 Write pseudocode for a program that reads a word and then prints the first character, the last character, and the character in the middle. For example, if the input is Harry, the program prints H y r. If the word has even length, print the character right before the middle.

- ■ R2.14 Write pseudocode for a program that prompts the user to enter a name (such as Harold James Morgan) and then prints a monogram consisting of the initial letters of the first, middle, and last name (such as HJM).

- ■ ■ R2.15 Write pseudocode for a program that computes the first and last digit of a number. For example, if the input is 23456, the program should print 2 and 6. Use % and $\log(x, 10)$.

- R2.16 Modify the pseudocode for the program in How To 2.1 so that the program gives change in quarters, dimes, and nickels. You can assume that the price is a multiple of 5 cents. To develop your pseudocode, first work with a couple of specific values.

- ■ R2.17 A cocktail shaker is composed of three cone sections.

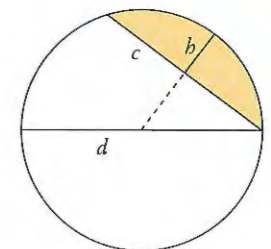
Using realistic values for the radii and heights, compute the total volume, using the formula given in Self Check 17 for a cone section. Then develop an algorithm that works for arbitrary dimensions.

- ■ ■ R2.18 You are cutting off a piece of pie like this, where c is the length of the straight part (called the chord length) and h is the height of the piece.

There is an approximate formula for the area:

$$A \approx \frac{2}{3}cb + \frac{b^3}{2c}$$

However, h is not so easy to measure, whereas the diameter d of a pie is usually well-known. Calculate the area where the diameter of the pie is 12 inches and the chord length of the segment is 10 inches. Generalize to an algorithm that yields the area for any diameter and chord length.



- ■ R2.19 The following pseudocode describes how to obtain the name of a day, given the day number (0 = Sunday, 1 = Monday, and so on.)

```
Define a string called names containing "SunMonTueWedThuFriSat".
Compute the starting position as 3 x the day number.
Get the characters at position, position + 1, position + 2.
Concatenate them.
```

Check this pseudocode, using the day number 4. Draw a diagram of the string that is being computed, similar to Figure 4.

- ■ R2.20 The following pseudocode describes how to swap two letters in a word.

```
We are given a string myString and two letters l1 and l2.
Change all occurrences of l1 to the character *
```



© Media Bakery

Change all occurrences of l_2 to l_1
 Change all occurrences of $*$ to l_2 .

Check this pseudocode, using the string "marmalade" and the letters a and e.

- **R2.21** How do you get the first character of a string? The last character? The middle character (if the length is odd)? The middle two characters (if the length is even?)
- **R2.22** This chapter contains a number of recommendations regarding variables and constants that make programs easier to read and maintain. Briefly summarize these recommendations.
- **R2.23** Give instructions for drawing an outlined oval within its bounding box. Use green lines for the bounding box.
- **Toolbox R2.24** How do you compute the derivative and integral of $f(x) = x^2$ in SymPy?
- **Toolbox R2.25** What is `diff(integrate(f))` in SymPy?
- **Toolbox R2.26** How would you write a Python program that uses SymPy to display the solution for an arbitrary quadratic equation, $ax^2 + bx + c = 0$?
- **Toolbox R2.27** How would you use SymPy to plot the curve $y = \sin(1/x)$, where x ranges from -0.5 to 0.5 ?
- **Toolbox R2.28** When you plot $\sin(x)/x$, what do you guess the limit is as x approaches zero?

PROGRAMMING EXERCISES

- **P2.1** Write a program that displays the dimensions of a letter-size (8.5×11 inch) sheet of paper in millimeters. There are 25.4 millimeters per inch. Use constants and comments in your program.
- **P2.2** Write a program that computes and displays the perimeter of a letter-size (8.5×11 inch) sheet of paper and the length of its diagonal.
- **P2.3** Write a program that reads a number and displays the square, cube, and fourth power. Use the `**` operator only for the fourth power.
- **P2.4** Write a program that prompts the user for two integers and then prints
 - The sum
 - The difference
 - The product
 - The average
 - The distance (absolute value of the difference)
 - The maximum (the larger of the two)
 - The minimum (the smaller of the two)

Hint: Python defines `max` and `min` functions that accept a sequence of values, each separated with a comma.

- **P2.5** Enhance the output of Exercise P2.4 so that the numbers are properly aligned:

```
Sum:      45
Difference: -5
Product:  500
Average:  22.50
Distance:  5
Maximum:  25
Minimum:  20
```

- **P2.6** Write a program that prompts the user for a measurement in meters and then converts it to miles, feet, and inches.
- **P2.7** Write a program that prompts the user for a radius and then prints
 - The area and circumference of a circle with that radius
 - The volume and surface area of a sphere with that radius
- **P2.8** Write a program that asks the user for the lengths of the sides of a rectangle. Then print
 - The area and perimeter of the rectangle
 - The length of the diagonal
- **P2.9** Improve the program discussed in How To 2.1 to allow input of quarters in addition to bills.
- **P2.10** Write a program that helps a person decide whether to buy a hybrid car. Your program's inputs should be:
 - The cost of a new car
 - The estimated miles driven per year
 - The estimated gas price
 - The efficiency in miles per gallon
 - The estimated resale value after 5 years

Compute the total cost of owning the car for five years. (For simplicity, we will not take the cost of financing into account.) Obtain realistic prices for a new and used hybrid and a comparable car from the Web. Run your program twice, using today's gas price and 15,000 miles per year. Include pseudocode and the program runs with your assignment.



- **P2.11** Write a program that asks the user to input
 - The number of gallons of gas in the tank
 - The fuel efficiency in miles per gallon
 - The price of gas per gallon
 Then print the cost per 100 miles and how far the car can go with the gas in the tank.
- **P2.12** *File names and extensions.* Write a program that prompts the user for the drive letter (C), the path (`\Windows\System`), the file name (`Readme`), and the extension (`.txt`). Then print the complete file name `C:\Windows\System\Readme.txt`. (If you use UNIX or a Macintosh, skip the drive name and use `/` instead of `\` to separate directories.)

- P2.13 Write a program that reads a number between 10,000 and 999,999 from the user, where the user enters a comma in the input. Then print the number without a comma. Here is a sample dialog; the user input is in color:

```
Please enter an integer between 10,000 and 99,999: 23,456
23456
```

Hint: Read the input as a string. Turn the strings consisting of the first two characters and the last three characters into numbers, and combine them.

- P2.14 Write a program that reads a number between 1,000 and 999,999 from the user and prints it with a comma separating the thousands.

Here is a sample dialog; the user input is in color:

```
Please enter an integer between 1000 and 999999: 23456
23,456
```

- P2.15 *Printing a grid.* Write a program that prints the following grid to play tic-tac-toe.

```
+---+---+
|   |   |
+---+---+
|   |   |
+---+---+
|   |   |
+---+---+
```

Of course, you could simply write seven statements of the form

```
print("+---+---+")
```

You should do it the smart way, though. Declare string variables to hold two kinds of patterns: a comb-shaped pattern and the bottom line. Print the comb three times and the bottom line once.

- P2.16 Write a program that reads a five-digit positive integer and breaks it into a sequence of individual digits. For example, the input 16384 is displayed as

```
1 6 3 8 4
```

- P2.17 Write a program that reads two times in military format (0900, 1730) and prints the number of hours and minutes between the two times. Here is a sample run. User input is in color.

```
Please enter the first time: 0900
Please enter the second time: 1730
8 hours 30 minutes
```

Extra credit if you can deal with the case where the first time is later than the second:

```
Please enter the first time: 1730
Please enter the second time: 0900
15 hours 30 minutes
```

- P2.18 *Writing large letters.* A large letter H can be produced like this:

```
* *
* *
*****
* *
* *
```

It can be declared as a string literal like this:

```
LETTER_H = "* *\n* *\n*****\n* *\n* *\n"
```

(The `\n` escape sequence denotes a “newline” character that causes subsequent characters to be printed on a new line.) Do the same for the letters E, L, and O. Then write the message

```
H
E
L
L
O
```

in large letters.

- P2.19 Write a program that transforms numbers 1, 2, 3, ..., 12 into the corresponding month names January, February, March, ..., December. *Hint:* Make a very long string “January February March ...”, in which you add spaces such that each month name has *the same length*. Then concatenate the characters of the month that you want. If you are bothered by the trailing spaces, use the `strip` method to remove them.

- P2.20 Write a program that prints a Christmas tree:

```
  / \
 /   \
/     \
-----
"  "
"  "
"  "
```

Remember to use escape sequences.

- P2.21 Easter Sunday is the first Sunday after the first full moon of spring. To compute the date, you can use this algorithm, invented by the mathematician Carl Friedrich Gauss in 1800:

1. Let y be the year (such as 1800 or 2001).
2. Divide y by 19 and call the remainder a . Ignore the quotient.
3. Divide y by 100 to get a quotient b and a remainder c .
4. Divide b by 4 to get a quotient d and a remainder e .
5. Divide $8 * b + 13$ by 25 to get a quotient g . Ignore the remainder.
6. Divide $19 * a + b - d - g + 15$ by 30 to get a remainder h . Ignore the quotient.
7. Divide c by 4 to get a quotient j and a remainder k .
8. Divide $a + 11 * h$ by 319 to get a quotient m . Ignore the remainder.
9. Divide $2 * e + 2 * j - k - h + m + 32$ by 7 to get a remainder r . Ignore the quotient.
10. Divide $h - m + r + 90$ by 25 to get a quotient n . Ignore the remainder.
11. Divide $h - m + r + n + 19$ by 32 to get a remainder p . Ignore the quotient.



© José Luis Gutiérrez/Stockphoto.

Then Easter falls on day p of month n . For example, if y is 2001:

$a = 6$	$h = 18$	$n = 4$
$b = 20, c = 1$	$j = 0, k = 1$	$p = 15$
$d = 5, e = 0$	$m = 0$	
$g = 6$	$r = 6$	

Therefore, in 2001, Easter Sunday fell on April 15. Write a program that prompts the user for a year and prints out the month and day of Easter Sunday.

- **P2.22** Write a program that initializes a string variable and prints the first three characters, followed by three periods, and then the last three characters. For example, if the string is initialized to "Mississippi", then print `Mis...ppi`.

- **Graphics P2.23** Write a graphics program that draws your name in red, contained inside a blue rectangle.
- **Graphics P2.24** Write a graphics program that draws two solid squares: one in pink and one in purple. Use a standard color for one of them and a custom color for the other.
- **Graphics P2.25** Write a program to plot the following face.



- **Graphics P2.26** Draw a "bull's eye"—a set of concentric rings in alternating black and white colors.



- **Graphics P2.27** Write a program that draws a picture of a house. It could be as simple as the accompanying figure, or if you like, make it more elaborate (3-D, skyscraper, marble columns in the entryway, whatever). Use at least three different colors.



- **Graphics P2.28** Draw the coordinate system figure shown in Section 2.6.2.
- **Graphics P2.29** Modify the `italianflag.py` program in How To 2.2 to draw a flag with three horizontal colored stripes, such as the German flag.
- **Graphics P2.30** Write a program that displays the Olympic rings. Color the rings in the Olympic colors.



- **Graphics P2.31** Make a bar chart to plot the following data set. Label each bar.

Bridge Name	Longest Span (ft)
Golden Gate	4,200
Brooklyn	1,595
Delaware Memorial	2,150
Mackinac	3,800

- **Business P2.32** The following pseudocode describes how a bookstore computes the price of an order from the total price and the number of the books that were ordered.

*Read the total book price and the number of books.
 Compute the tax (7.5 percent of the total book price).
 Compute the shipping charge (\$2 per book).
 The price of the order is the sum of the total book price, the tax, and the shipping charge.
 Print the price of the order.*

Translate this pseudocode into a Python program.

- **Business P2.33** The following pseudocode describes how to turn a string containing a ten-digit phone number (such as "415551212") into a more readable string with parentheses and dashes, like this: "(415) 555-1212".

*Take the string consisting of the first three characters and surround it with "(" and ") ". This is the area code.
 Concatenate the area code, the string consisting of the next three characters, a hyphen, and the string consisting of the last four characters. This is the formatted number.*

Translate this pseudocode into a Python program that reads a telephone number into a string variable, computes the formatted number, and prints it.

- **Business P2.34** The following pseudocode describes how to extract the dollars and cents from a price given as a floating-point value. For example, a price of 2.95 yields values 2 and 95 for the dollars and cents.

*Convert the price to an integer and store it in a variable dollars.
 Multiply the difference price - dollars by 100 and add 0.5.
 Convert the result to an integer variable and store it in a variable cents.*

Translate this pseudocode into a Python program. Read a price and print the dollars and cents. Test your program with inputs 2.95 and 4.35.

- **Business P2.35** *Giving change.* Implement a program that directs a cashier how to give change. The program has two inputs: the amount due and the amount received from the customer. Display the dollars, quarters, dimes, nickels, and pennies that the customer should receive in return. In order to avoid roundoff errors, the program user should supply both amounts in pennies, for example 274 instead of 2.74.



- **Business P2.36** An online bank wants you to create a program that shows prospective customers how their deposits will grow. Your program should read the initial balance and the annual interest rate. Interest is compounded monthly. Print out the balances after the first three months. Here is a sample run:

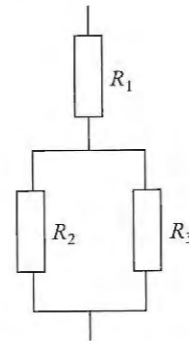
```
Initial balance: 1000
Annual interest rate in percent: 6.0
After first month: 1005.00
After second month: 1010.03
After third month: 1015.08
```

- **Business P2.37** A video club wants to reward its best members with a discount based on the member's number of movie rentals and the number of new members referred by the member. The discount is in percent and is equal to the sum of the rentals and the referrals, but it cannot exceed 75 percent. Write a program to calculate the value of the discount.

Here is a sample run:

```
Enter the number of movie rentals: 56
Enter the number of members referred to the video club: 3
The discount is equal to: 59.00 percent.
```

- **Science P2.38** Consider the following circuit.



Write a program that reads the resistances of the three resistors and computes the total resistance, using Ohm's law.

- **Science P2.39** The dew point temperature T_d can be calculated (approximately) from the relative humidity RH and the actual temperature T by

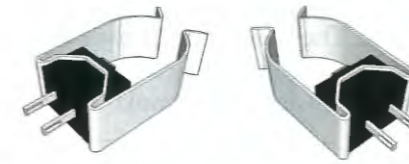
$$T_d = \frac{b \cdot f(T, RH)}{a - f(T, RH)}$$

$$f(T, RH) = \frac{a \cdot T}{b + T} + \ln(RH)$$

where $a = 17.27$ and $b = 237.7^\circ \text{C}$.

Write a program that reads the relative humidity (between 0 and 1) and the temperature (in degrees C) and prints the dew point value. Use the Python function `log` to compute the natural logarithm.

- **Science P2.40** The pipe clip temperature sensors shown here are robust sensors that can be clipped directly onto copper pipes to measure the temperature of the liquids in the pipes.



Each sensor contains a device called a *thermistor*. Thermistors are semiconductor devices that exhibit a temperature-dependent resistance described by:

$$R = R_0 e^{\beta \left(\frac{1}{T} - \frac{1}{T_0} \right)}$$

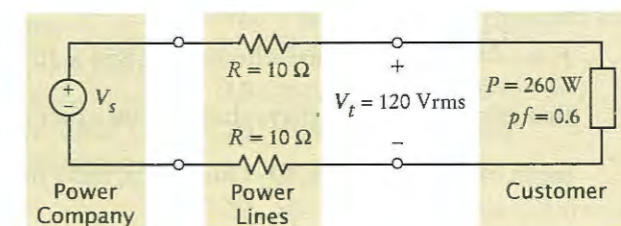
where R is the resistance (in Ω) at the temperature T (in $^\circ\text{K}$), and R_0 is the resistance (in Ω) at the temperature T_0 (in $^\circ\text{K}$). β is a constant that depends on the material used to make the thermistor. Thermistors are specified by providing values for R_0 , T_0 , and β .

The thermistors used to make the pipe clip temperature sensors have $R_0 = 1075 \Omega$ at $T_0 = 85^\circ\text{C}$, and $\beta = 3969^\circ\text{K}$. (Notice that β has units of $^\circ\text{K}$. Recall that the temperature in $^\circ\text{K}$ is obtained by adding 273.15 to the temperature in $^\circ\text{C}$.) The liquid temperature, in $^\circ\text{C}$, is determined from the resistance R , in Ω , using

$$T = \frac{\beta T_0}{T_0 \ln \left(\frac{R}{R_0} \right) + \beta} - 273$$

Write a Python program that prompts the user for the thermistor resistance R and prints a message giving the liquid temperature in $^\circ\text{C}$.

- **Science P2.41** The circuit shown below illustrates some important aspects of the connection between a power company and one of its customers. The customer is represented by three parameters, V_t , P , and pf . V_t is the voltage accessed by plugging into a wall outlet. Customers depend on having a dependable value of V_t in order for their appliances to work properly. Accordingly, the power company regulates the value of V_t carefully. P describes the amount of power used by the customer and is the primary factor in determining the customer's electric bill. The power factor, pf , is less familiar. (The power factor is calculated as the cosine of an angle so that its value will always be between zero and one.) In this problem you will be asked to write a Python program to investigate the significance of the power factor.

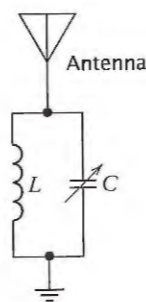


In the figure, the power lines are represented, somewhat simplistically, as resistances in Ohms. The power company is represented as an AC voltage source. The source voltage, V_s , required to provide the customer with power P at voltage V_t can be determined using the formula

$$V_s = \sqrt{\left(V_t + \frac{2RP}{V_t}\right)^2 + \left(\frac{2RP}{pfV_t}\right)^2 (1 - pf^2)}$$

(V_s has units of Vrms.) This formula indicates that the value of V_s depends on the value of pf . Write a Python program that prompts the user for a power factor value and then prints a message giving the corresponding value of V_s , using the values for P , R , and V_t shown in the figure above.

- ■ ■ **Science P2.42** Consider the following tuning circuit connected to an antenna, where C is a variable capacitor whose capacitance ranges from C_{\min} to C_{\max} .



The tuning circuit selects the frequency $f = \frac{2\pi}{\sqrt{LC}}$. To design this circuit for a given frequency, take $C = \sqrt{C_{\min}C_{\max}}$ and calculate the required inductance L from f and C . Now the circuit can be tuned to any frequency in the range $f_{\min} = \frac{2\pi}{\sqrt{LC_{\max}}}$ to $f_{\max} = \frac{2\pi}{\sqrt{LC_{\min}}}$.

Write a Python program to design a tuning circuit for a given frequency, using a variable capacitor with given values for C_{\min} and C_{\max} . (A typical input is $f = 16.7$ MHz, $C_{\min} = 14$ pF, and $C_{\max} = 365$ pF.) The program should read in f (in Hz), C_{\min} and C_{\max} (in F), and print the required inductance value and the range of frequencies to which the circuit can be tuned by varying the capacitance.

- ■ ■ **Science P2.43** According to the Coulomb force law, the electric force between two charged particles of charge Q_1 and Q_2 Coulombs, that are a distance r meters apart, is $F = \frac{Q_1 Q_2}{4\pi\epsilon r^2}$ Newtons, where $\epsilon = 8.854 \times 10^{-12}$ Farads/meter. Write a program that calculates and displays the force on a pair of charged particles, based on the user input of Q_1 Coulombs, Q_2 Coulombs, and r meters.

ANSWERS TO SELF-CHECK QUESTIONS

1. One possible answer is

```
bottlesPerCase = 8
```

You may choose a different variable name or a different initialization value, but your variable should have type int.

2. There are two errors:

- You cannot have spaces in variable names.
- There are about 33.81 ounces per liter, not 28.35.

3.

```
unitPrice = 1.95
quantity = 2
```

4.

```
print("Total price:", unitPrice * quantity)
```

5. Change the declaration of `cansPerPack` to `cansPerPack = 4`

6. Its value is modified by the assignment statement.

7. Assignment would occur when one car is replaced by another in the parking space.

8.

```
interest = balance * percent / 100
```

9.

```
sideLength = sqrt(area)
```

10.

```
4 / 3 * pi * radius ** 3
```

11. 172 and 9

12. It is the second-to-last digit of n . For example, if n is 1729, then $n // 10$ is 172, and $(n // 10) \% 10$ is 2.

13.

```
pairs = (totalWidth - tileWidth) // (2 *
tileWidth)
tiles = 1 + 2 * pairs
gap = (totalWidth -
tiles * tileWidth) / 2.0
```

14. Now there are groups of four tiles (gray/white/gray/black) following the initial black tile. Therefore, the algorithm is now

```
number of groups = integer part of (total width - tile width)
/ (4 x tile width)
number of tiles = 1 + 4 x number of groups
```

The formula for the gap is not changed.

15. Clearly, the answer depends only on whether the row and column numbers are even or odd, so let's first take the remainder after dividing by 2. Then we can enumerate all expected answers:

Row % 2	Column % 2	Color
0	0	0
0	1	1
1	0	1
1	1	0

In the first three entries of the table, the color is simply the sum of the remainders. In the fourth entry, the sum would be 2, but we want a zero. We can achieve that by taking another remainder operation:

```
color = ((row % 2) + (column % 2)) % 2
```

16. In nine years, the repair costs increased by \$1,400. Therefore, the increase per year is $\$1,400 / 9 \approx \156 . The repair cost in year 3 would be $\$100 + 2 \times \$156 = \$412$. The repair cost in year n is $\$100 + n \times \156 . To avoid accumulation of roundoff errors, it is actually a good idea to use the original expression that yielded \$156, that is,

```
Repair cost in year n = 100 + n x 1400 / 9
```

17. The pseudocode follows easily from the equations:

```
bottom volume = pi x r1^2 x h1
```

```
top volume = pi x r2^2 x h2
```

```
middle volume = pi x (r1^2 + r1 x r2 + r2^2) x h3 / 3
```

```
total volume = bottom volume + top volume + middle volume
```

Measuring a typical wine bottle yields $r_1 = 3.6$, $r_2 = 1.2$, $h_1 = 15$, $h_2 = 7$, $h_3 = 6$ (all in centimeters). Therefore,

```
bottom volume = 610.73
```

```
top volume = 31.67
```

```
middle volume = 135.72
```

```
total volume = 778.12
```

The actual volume is 750 ml, which is close enough to our computation to give confidence that it is correct.

18. The length is 14. The space counts as a character.

19.

```
title.replace("Python Pro", "")
```

20.

```
title = title + "ming"
```

21. Hy

22.

```
age = int(input("How old are you? "))
```


23. The second statement calls `int`, not `float`. If the user were to enter a price such as 1.95, the program would be terminated with a “value error”.

24. There is no colon and space at the end of the prompt. A dialog would look like this:
Please enter the number of cans6

25. The total volume is 10
There are four spaces between `is` and `10`. One space originates from the format string (the space between `s` and `%`) and three spaces are added before `10` to achieve the field width of 5.

26. Here is a simple solution:

```
print("Bottles: %8d" % bottles)
print("Cans:    %8d" % cans)
```

Note the spaces after `Cans:`. Alternatively, you can use format specifiers for the strings.

```
print("%-8s %8d" % ("Bottles:", bottles))
print("%-8s %8d" % ("Cans:", cans))
```

27. Here is one possible solution:

```
canvas.drawRect(0, 0, 50, 50)
canvas.drawRect(0, 100, 50, 50)
```

28. The program shows three very elongated ellipses instead of the rectangles.

29. `canvas.drawOval(75, 75, 50, 50)`

30. `canvas.drawLine(0, 0, 10, 30)`
`canvas.drawLine(10, 30, 20, 0)`

31. `canvas.drawText(0, 30, "V")`

32. `win = GraphicsWindow(200, 200)`
`canvas = win.canvas()`
`canvas.setColor("red")`
`canvas.drawRect(0, 0, 200, 200)`
`canvas.setColor("yellow")`
`canvas.drawRect(50, 50, 100, 100)`