CHAPTER 2

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 30

Syntax 2.1: Assignment 31

Common Error 2.1: Using Undefined Variables 36

Programming Tip 2.1: Choose Descriptive Variable Names 36

Programming Tip 2.2: Do Not Use Magic Numbers 37

2.2 ARITHMETIC 37

Syntax 2.2: Calling Functions 40

Common Error 2.2: Roundoff Errors 43

Common Error 2.3: Unbalanced Parentheses 43

Programming Tip 2.3: Use Spaces in Expressions 44

Special Topic 2.1: Other Ways to Import Modules 44

Special Topic 2.2: Combining Assignment and Arithmetic 44

Special Topic 2.3: Line Joining 45

2.3 PROBLEM SOLVING: FIRST DO IT BY HAND 45

Worked Example 2.1: Computing Travel Time 47

2.4 STRINGS 48

Special Topic 2.4: Character Values 53

Special Topic 2.5: Escape Sequences 54

Computing & Society 2.1: International Alphabets and Unicode 54

2.5 INPUT AND OUTPUT 55

Syntax 2.3: String Format Operator 57

Programming Tip 2.4: Don't Wait to Convert 60

How To 2.1: Writing Simple Programs 60

Worked Example 2.2: Computing the Cost of Stamps 63

Computing & Society 2.2: The Pentium Floating-Point Bug 65

2.6 GRAPHICS: SIMPLE DRAWINGS 65

How To 2.2: Drawing Graphical Shapes 72

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.)

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.

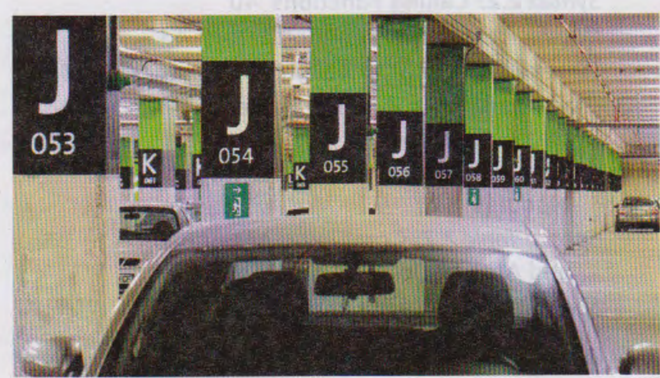
What contains more soda? A six-pack of 12-ounce cans or a two-liter bottle?

2.1.1 Defining Variables

A variable is a storage location with a name.

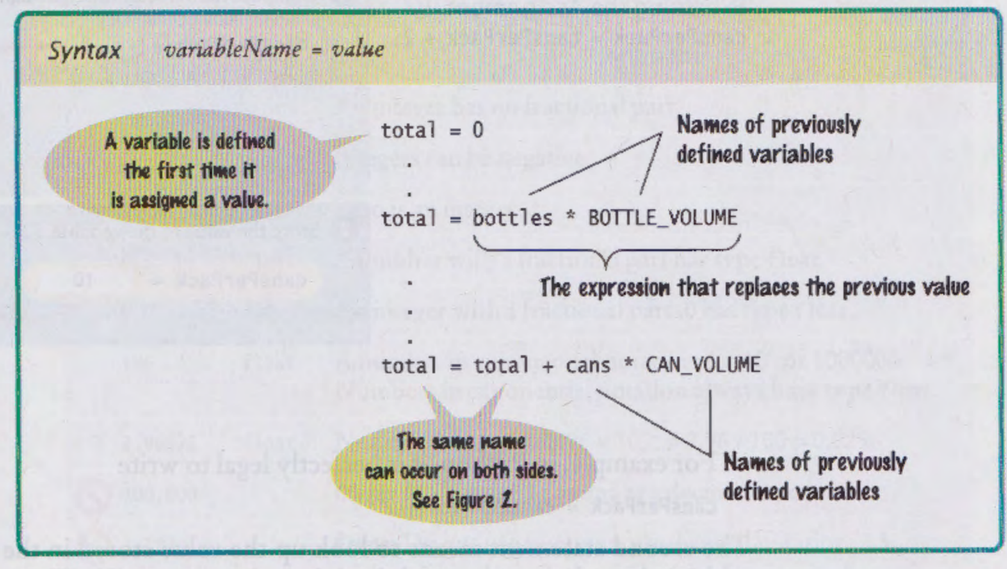
A **variable** is a storage location in a computer program. Each variable has a name and holds a value.

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).



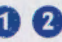
Like a variable in a computer program, a parking space has an identifier and a contents.

Syntax 2.1 Assignment

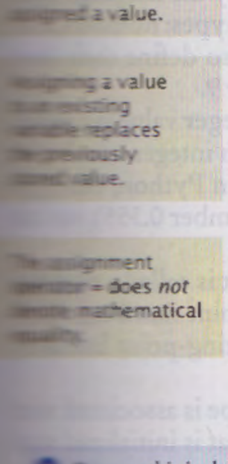


An assignment statement stores a value in a variable.

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.



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

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`.

Assigning a value to an existing variable replaces the previously stored value.

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

```
cansPerPack = 8
```
❸

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

The assignment operator = does *not* denote mathematical equality.

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.

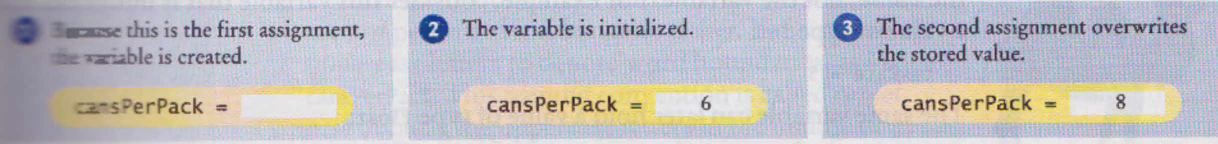
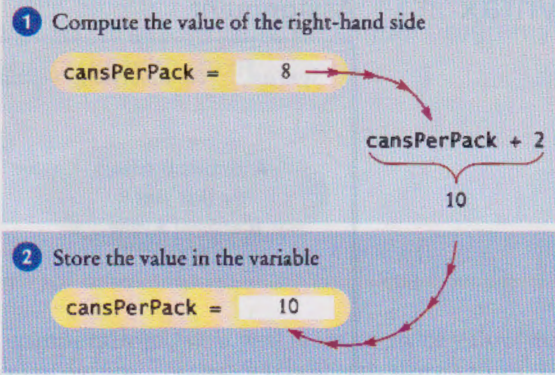


Figure 1 Executing Two Assignments

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 comptuer and what operations can be performed on the value.

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.

Integers are whole numbers without a fractional part.

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.

Floating-point numbers contain a fractional part.

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
```

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:

1. Names must start with a letter or the underscore (_) character, and the remaining characters must be letters, numbers, or underscores.
2. 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.

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 *x* or *y*. This is legal in Python, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 36).
⚠ CanVolume	**Caution:** Variable names are case sensitive. This variable name is different from canVolume, 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 / or.

3. Names are **case sensitive**, that is, canVolume and canvolume are different names.
4. You cannot use **reserved words** such as if or class as names; these words are reserved exclusively for their special Python meanings. (See Appendix C 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.

1. It is better to use a descriptive name, such as cansPerPack, than a terse name, such as cpp.
2. 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).

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

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

2.1.4 Constants

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

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.

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

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,

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

```
##
#  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/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
```

```
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 display 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, 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

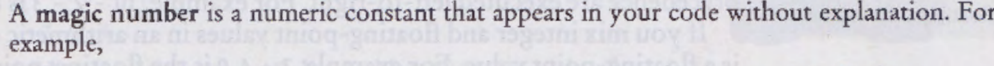
```
cv = 0.355
```

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
```

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

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
```

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.