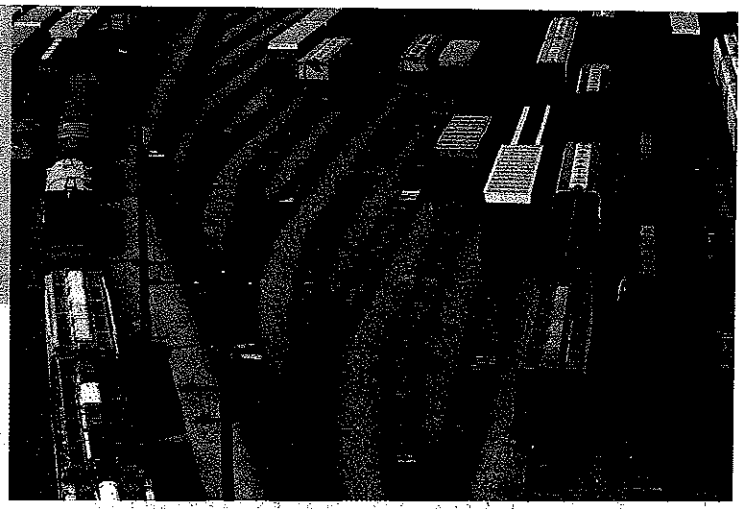
CHAPTER 3

DECISIONS



CHAPTER GOALS

To implement decisions using if statements

To compare integers, floating-point numbers, and strings

To write statements using Boolean expressions

To develop strategies for testing your programs

To validate user input

CHAPTER CONTENTS

One of the essential features of computer programs is their ability to make decisions. Like a train that changes tracks depending on how the switches are set, a program can take different actions depending on inputs and other circumstances.

In this chapter, you will learn how to program simple and complex decisions. You will apply what you learn to the task of checking user input.

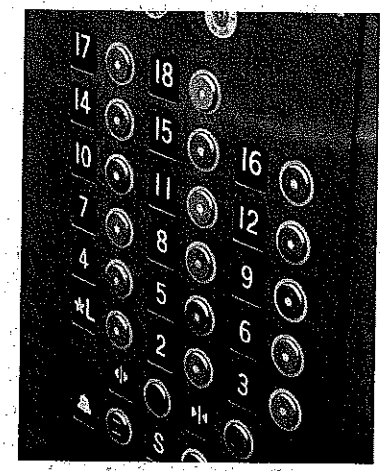
3.1 The `if` Statement

The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.

The `if` statement is used to implement a decision (see Syntax 3.1 on page 94). When a condition is fulfilled, one set of statements is executed. Otherwise, another set of statements is executed.

Here is an example using the `if` statement: In many countries, the number 13 is considered unlucky. Rather than offending superstitious tenants, building owners sometimes skip the thirteenth floor; floor 12 is immediately followed by floor 14. Of course, floor 13 is not usually left empty or, as some conspiracy theorists believe, filled with secret offices and research labs. It is simply called floor 14. The computer that controls the building elevators needs to compensate for this foible and adjust all floor numbers above 13.

Let's simulate this process in Python. We will ask the user to type in the desired floor number and then compute the actual floor. When the input is above 13, then we need to decrement the input to obtain the actual floor.



This elevator panel "skips" the thirteenth floor. The floor is not actually missing—the computer that controls the elevator adjusts the floor numbers above 13.

An `if` statement is like a fork in the road. Depending upon a decision, different parts of the program are executed.

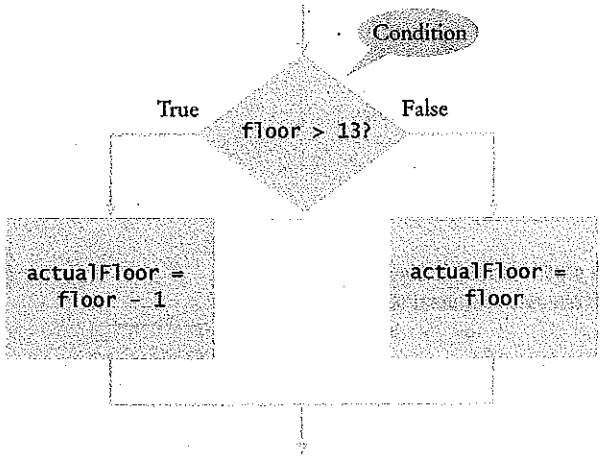


Figure 1
Flowchart for `if` Statement

For example, if the user provides an input of 20, the program determines the actual floor as 19. Otherwise, we simply use the supplied floor number.

```
actualFloor = 0

if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor
```

The flowchart in Figure 1 shows the branching behavior.

In our example, each branch of the `if` statement contains a single statement. You can include as many statements in each branch as you like. Sometimes, it happens that there is nothing to do in the `else` branch of the statement. In that case, you can omit it entirely, such as in this example:

```
actualFloor = floor

if floor > 13 :
   actualFloor = actualFloor - 1
```

See Figure 2 for the flowchart.

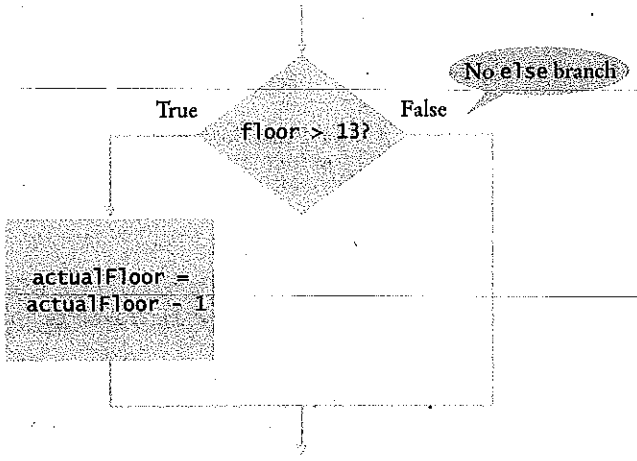


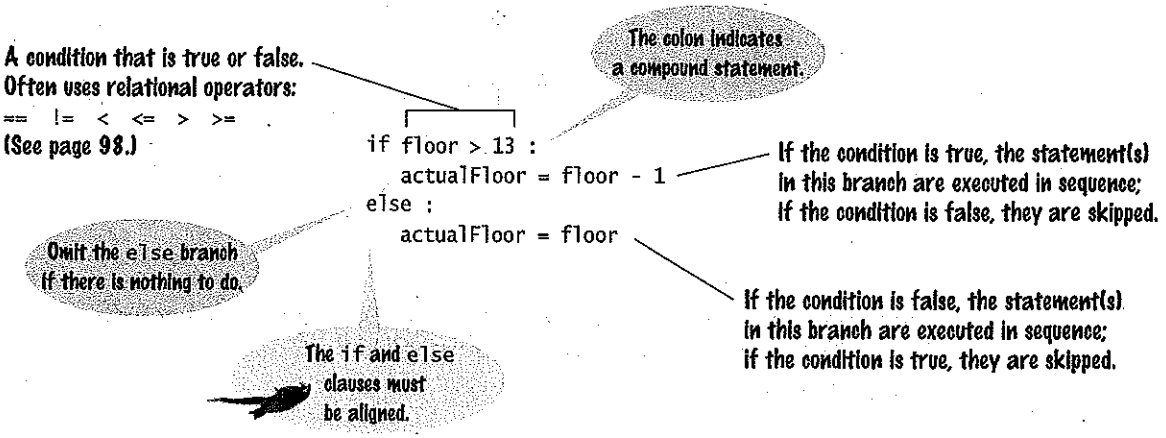
Figure 2
Flowchart for `if` Statement with No `else` Branch

Syntax 3.1 if Statement

Syntax

```
if condition :
   statements
```

```
if condition :
   statements1
else :
   statements2
```



The following program puts the `if` statement to work. This program asks for the desired floor and then prints out the actual floor.

ch03/elevatorsim.py

```
1  ##
2  #  This program simulates an elevator panel that skips the 13th floor.
3  #
4
5  # Obtain the floor number from the user as an integer.
6  floor = int(input("Floor: "))
7
8  # Adjust floor if necessary.
9  if floor > 13 :
10    actualFloor = floor - 1
11 else :
12    actualFloor = floor
13
14 # Print the result.
15 print("The elevator will travel to the actual floor", actualFloor)
```

Program Run

```
Floor: 20
The elevator will travel to the actual floor 19
```

The Python instructions we have used so far have been simple statements that must be contained on a single line (or explicitly continued to the next line—see

Compound statements consist of a header and a statement block.

Special Topic 2.3). Some constructs in Python are **compound statements**, which span multiple lines and consist of a *header* and a **statement block**. The `if` statement is an example of a compound statement.

```
if totalSales > 100.0 :    # The header ends in a colon.
   discount = totalSales * 0.05    # Lines in the block are indented to the same level
   totalSales = totalSales - discount
   print("You received a discount of", discount)
```

Compound statements require a colon (:) at the end of the header. The statement block is a group of one or more statements, all of which are indented to the same indentation level. A statement block begins on the line following the header and ends at the first statement indented less than the first statement in the block. You can use any number of spaces to indent statements within a block, but all statements within the block must have the same indentation level. Note that comments are not statements and thus can be indented to any level.

Statement blocks, which can be nested inside other blocks, signal that one or more statements are part of the given compound statement. In the case of the `if` construct, the statement block specifies the instructions that will be executed if the condition is true or skipped if the condition is false.

1. In some Asian countries, the number 14 is considered unlucky. Some building owners play it safe and skip *both* the thirteenth and the fourteenth floor. How would you modify the sample program to handle such a building?
2. Consider the following `if` statement to compute a discounted price:

   ```
   if originalPrice > 100 :
      discountedPrice = originalPrice - 20
   else :
      discountedPrice = originalPrice - 10
   ```

   What is the discounted price if the original price is 95? 100? 105?
3. Compare this `if` statement with the one in Self Check 2:

   ```
   if originalPrice < 100 :
      discountedPrice = originalPrice - 10
   else :
      discountedPrice = originalPrice - 20
   ```

   Do the two statements always compute the same value? If not, when do the values differ?
4. Consider the following statements to compute a discounted price:

   ```
   discountedPrice = originalPrice
   if originalPrice > 100 :
      discountedPrice = originalPrice - 10
   ```

   What is the discounted price if the original price is 95? 100? 105?
5. The variables `fuelAmount` and `fuelCapacity` hold the actual amount of fuel and the size of the fuel tank of a vehicle. If less than 10 percent is remaining in the tank, a status light should show a red color; otherwise it shows a green color. Simulate this process by printing out either `"red"` or `"green"`.

Practice It Now you can try these exercises at the end of the chapter: R3.5, R3.6, P3.32.

Common Error 3.1

Tabs

Block-structured code has the property that nested statements are indented by one or more levels:

```
if totalSales > 100.0 :
|  discount = totalSales * 0.05
|  totalSales = totalSales - discount
|  print("You received a discount of $%.2f" % discount)
else :
|  diff = 100.0 - totalSales
|  if diff < 10.0 :
|  |  print("If you were to purchase our item of the day you can receive a 5% discount,")
|  else :
|  |  print("You need to spend $%.2f more to receive a 5% discount." % diff)
|  |  |
|  |  |
0  1  2  Indentation level
```

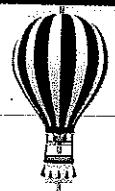
Python requires block-structured code as part of its syntax. The alignment of statements within a Python program specifies which statements are part of a given statement block.

How do you move the cursor from the leftmost column to the appropriate indentation level? A perfectly reasonable strategy is to hit the space bar a sufficient number of times. With most editors, you can use the Tab *key* instead. A tab moves the cursor to the next indentation level. Some editors even have an option to fill in the tabs automatically.

While the Tab *key* is nice, some editors use *tab characters* for alignment, which is not so nice. Python is very picky as to how you align the statements within a statement block. All of the statements must be aligned with either blank spaces or tab characters, but not a mixture of the two. In addition, tab characters can lead to problems when you send your file to another person or a printer. There is no universal agreement on the width of a tab character, and some software will ignore tab characters altogether. It is therefore best to save your files with spaces instead of tabs. Most editors have a setting to automatically convert all tabs to spaces.

Look at the documentation of your development environment to find out how to activate this useful setting.

Programming Tip 3.1



Avoid Duplication in Branches

Look to see whether you *duplicate code* in each branch. If so, move it out of the `if` statement. Here is an example of such duplication:

```
if floor > 13 :
   actualFloor = floor - 1
   print("Actual floor:", actualFloor)
else :
   actualFloor = floor
   print("Actual floor:", actualFloor)
```

The output statement is exactly the same in both branches. This is not an error—the program will run correctly. However, you can simplify the program by moving the duplicated statement, like this:

```
if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor
print("Actual floor:", actualFloor)
```

Removing duplication is particularly important when programs are maintained for a long time. When there are two sets of statements with the same effect, it can easily happen that a programmer modifies one set but not the other.

Special Topic 3.1

Conditional Expressions

Python has a conditional operator of the form

value$_1$ `if` *condition* `else` *value*$_2$

The value of that expression is either *value*$_1$ if the condition is true or *value*$_2$ if it is false. For example, we can compute the actual floor number as

```
actualFloor = floor - 1 if floor > 13 else floor
```

which is equivalent to

```
if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor
```

Note that a conditional expression is a single statement that must be contained on a single line or continued to the next line (see Special Topic 2.3). Also note that a colon is not needed because a conditional expression is not a compound statement.

You can use a conditional expression anywhere that a value is expected, for example:

```
print("Actual floor:", floor - 1 if floor > 13 else floor)
```

We don't use the conditional expression in this book, but it is a convenient construct that you will find in some Python programs.

3.2 Relational Operators

Use relational operators (< <= > >= == !=) to compare numbers and strings.

In this section, you will learn how to compare numbers and strings in Python.

Every `if` statement contains a condition. In many cases, the condition involves comparing two values. For example, in the previous examples we tested `floor > 13`. The comparison `>` is called a **relational operator**. Python has six relational operators (see Table 1).

In Python, you use a relational operator to check whether one value is greater than another.

As you can see, only two Python relational operators (`>` and `<`) look as you would expect from the mathematical notation. Computer keyboards do not have keys for ≥, ≤, or ≠, but the `>=`, `<=`, and `!=` operators are easy to remember because they look similar. The `==` operator is initially confusing to most newcomers to Python.

Table 1 Relational Operators

Python	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

In Python, = already has a meaning, namely assignment. The == operator denotes equality testing:

```
floor = 13   # Assign 13 to floor
if floor == 13 :   # Test whether floor equals 13
```

You must remember to use == inside tests and to use = outside tests.

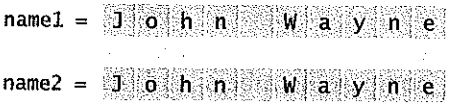
Strings can also be compared using Python's relational operators. For example, to test whether two strings are equal, use the == operator

```
if name1 == name2 :
   print("The strings are identical.")
```

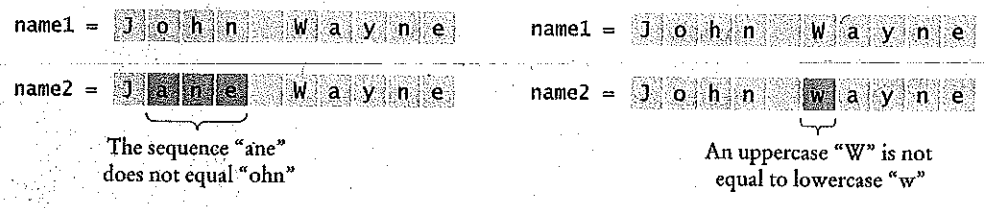
or to test if they are not equal, use the != operator

```
if name1 != name2 :
   print("The strings are not identical.")
```

For two strings to be equal, they must be of the same length and contain the same sequence of characters:



If even one character is different, the two strings will not be equal:



The relational operators in Table 1 have a lower precedence than the arithmetic operators. That means you can write arithmetic expressions on either side of the relational operator without using parentheses. For example, in the expression

```
floor - 1 < 13
```

both sides (`floor - 1` and `13`) of the < operator are evaluated, and the results are compared. Appendix B shows a table of the Python operators and their precedences.

Table 2 Relational Operator Examples

	Expression	Value	Comment
	`3 <= 4`	`True`	3 is less than 4; <= tests for "less than or equal".
⊘	`3 =< 4`	**Error**	The "less than or equal" operator is <=, not =<. The "less than" symbol comes first.
	`3 > 4`	`False`	> is the opposite of <=.
	`4 < 4`	`False`	The left-hand side must be strictly smaller than the right-hand side.
	`4 <= 4`	`True`	Both sides are equal; <= tests for "less than or equal".
	`3 == 5 - 2`	`True`	== tests for equality.
	`3 != 5 - 1`	`True`	!= tests for inequality. It is true that 3 is not 5 – 1.
⊘	`3 = 6 / 2`	**Error**	Use == to test for equality.
	`1.0 / 3.0 == 0.333333333`	`False`	Although the values are very close to one another, they are not exactly equal. See Common Error 3.2 on page 101.
⊘	`"10" > 5`	**Error**	You cannot compare a string to a number.

Table 2 summarizes how to compare values in Python. The following program demonstrates comparisons using logical expressions.

ch03/compare.py

```
1  ##
2  #  This program demonstrates comparisons of numbers and strings.
3  #
4
5  from math import sqrt
6
7  # Comparing integers
8  m = 2
9  n = 4
10
11 if m * m == n :
12    print("2 times 2 is four.")
13
14 # Comparing floating-point numbers
15 x = sqrt(2)
16 y = 2.0
17
18 if x * x == y :
19    print("sqrt(2) times sqrt(2) is 2")
20 else :
21    print("sqrt(2) times sqrt(2) is not four but %.18f" % (x * x))
22
23 EPSILON = 1E-14
```

```
24 if abs(x * x - y) < EPSILON :
25    print("sqrt(2) times sqrt(2) is approximately 2")
26
27 # Comparing strings
28 s = "120"
29 t = "20"
30
31 if s == t :
32    comparison = "is the same as"
33 else :
34    comparison = "is not the same as"
35
36 print("The string '%s' %s the string '%s'." % (s, comparison, t))
37
38 u = "1" + t
39 if s != u :
40    comparison = "not "
41 else :
42    comparison = ""
43
44 print("The strings '%s' and '%s' are %sidentical." % (s, u, comparison))
```

Program Run

```
2 times 2 is four.
sqrt(2) times sqrt(2) is not four but 2.000000000000000444
sqrt(2) times sqrt(2) is approximately 2
The string '120' is not the same as the string '20'.
The strings '120' and '120' are identical.
```

6. Which of the following conditions are true, provided `a` is 3 and `b` is 4?
   a. `a + 1 <= b`
   b. `a + 1 >= b`
   c. `a + 1 != b`
7. Give the opposite of the condition
   `floor > 13`
8. What is the error in this statement?
   ```
   if scoreA = scoreB :
      print("Tie")
   ```
9. Supply a condition in this `if` statement to test whether the user entered a Y:
   ```
   userInput = input("Enter Y to quit.")
   if . . . :
      print("Goodbye")
   ```
10. How do you test that a string `userInput` is the empty string?
11. Consider the two strings
    `"This is a long string."`
    `"This is a lOng string;"`
    Why are the two strings not equal?

Practice It Now you can try these exercises at the end of the chapter: R3.4, R3.7.

Common Error 3.2

Exact Comparison of Floating-Point Numbers

Take limited precision into account when comparing floating-point numbers.

Floating-point numbers have only a limited precision, and calculations can introduce roundoff errors. You must take these inevitable roundoffs into account when comparing floating-point numbers. For example, the following code multiplies the square root of 2 by itself. Ideally, we expect to get the answer 2:

```
from math import sqrt

r = sqrt(2.0)
if r * r == 2.0 :
   print("sqrt(2.0) squared is 2.0")
else :
   print("sqrt(2.0) squared is not 2.0 but", r * r)
```

This program displays

```
sqrt(2.0) squared is not 2.0 but 2.0000000000000004
```

It does not make sense in most circumstances to compare floating-point numbers exactly. Instead, we should test whether they are *close enough*. That is, the magnitude of their difference should be less than some threshold. Mathematically, we would write that x and y are close enough if

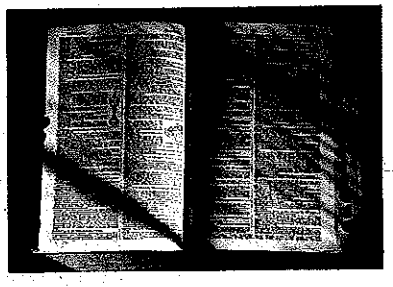
$$|x - y| < \varepsilon$$

for a very small number, ε. ε is the Greek letter epsilon, a letter used to denote a very small quantity. It is common to set ε to 10^{-14} when comparing floating-point numbers:

```
from math import sqrt

EPSILON = 1E-14
r = sqrt(2.0)
if abs(r * r - 2.0) < EPSILON :
   print("sqrt(2.0) squared is approximately 2.0")
```

Special Topic 3.2

Lexicographic Ordering of Strings



To see which of two terms comes first in the dictionary, consider the first letter in which they differ.

If two strings are not identical to each other, you still may want to know the relationship between them. Python's relational operators compare strings in "lexicographic" order. This ordering is very similar to the way in which words are sorted in a dictionary. If

```
string1 < string2
```

then the string `string1` comes before the string `string2` in the dictionary. For example, this is the case if `string1` is `"Harry"`, and `string2` is `"Hello"`. If

```
string1 > string2
```

then `string1` comes after `string2` in dictionary order.

As you have seen in the preceding section, if

```
string1 == string2
```

then `string1` and `string2` are equal.

There are a few technical differences between the ordering in a dictionary and the lexicographic ordering in Python.

The relational operators compare strings in lexicographic order.

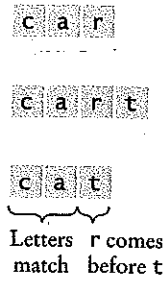
In Python:

- All uppercase letters come before the lowercase letters. For example, "Z" comes before "a".
- The space character comes before all printable characters.
- Numbers come before letters.
- For the ordering of punctuation marks, see Appendix A.

When comparing two strings, you compare the first letters of each word, then the second letters, and so on, until one of the strings ends or you find the first letter pair that doesn't match.

If one of the strings ends, the longer string is considered the "larger" one. For example, compare "car" with "cart". The first three letters match, and we reach the end of the first string. Therefore "car" comes before "cart" in the lexicographic ordering.

When you reach a mismatch, the string containing the "larger" character is considered "larger". For example, let's compare "cat" with "cart". The first two letters match. Because t comes after r, the string "cat" comes after "cart" in the lexicographic ordering.



Lexicographic Ordering

HOW TO 3.1 Implementing an if Statement

This How To walks you through the process of implementing an if statement.

Problem Statement The university bookstore has a Kilobyte Day sale every October 24, giving an 8 percent discount on all computer accessory purchases if the price is less than $128, and a 16 percent discount if the price is at least $128. Write a program that asks the cashier for the original price and then prints the discounted price.

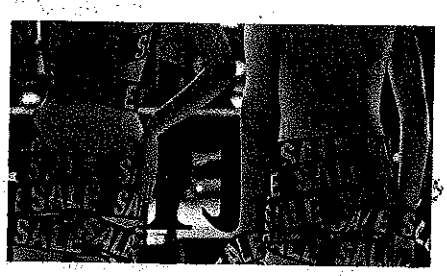
Step 1 Decide upon the branching condition.

In our sample problem, the obvious choice for the condition is:

```
original price < 128?
```

That is just fine, and we will use that condition in our solution.

But you could equally well come up with a correct solution if you choose the opposite condition: Is the original price at least $128? You might choose this condition if you put yourself into the position of a shopper who wants to know when the bigger discount applies.



Sales discounts are often higher for expensive products. Use the if statement to implement such a decision.

Step 2 Give pseudocode for the work that needs to be done when the condition is true.

In this step, you list the action or actions that are taken in the "positive" branch. The details depend on your problem. You may want to print a message, compute values, or even exit the program.

In our example, we need to apply an 8 percent discount:

```
discounted price = 0.92 x original price
```

Step 3 Give pseudocode for the work (if any) that needs to be done when the condition is *not* true.

What do you want to do in the case that the condition of Step 1 is not satisfied? Sometimes, you want to do nothing at all. In that case, use an if statement without an else branch.

In our example, the condition tested whether the price was less than $128. If that condition is *not* true, the price is at least $128, so the higher discount of 16 percent applies to the sale:

```
discounted price = 0.84 x original price
```

Step 4 Double-check relational operators.

First, be sure that the test goes in the right *direction*. It is a common error to confuse > and <. Next, consider whether you should use the < operator or its close cousin, the <= operator.

What should happen if the original price is exactly $128? Reading the problem carefully, we find that the lower discount applies if the original price is *less than* $128, and the higher discount applies when it is *at least* $128. A price of $128 should therefore *not* fulfill our condition, and we must use <, not <=.

Step 5 Remove duplication.

Check which actions are common to both branches, and move them outside.

In our example, we have two statements of the form

```
discounted price = ___ x original price
```

They only differ in the discount rate. It is best to just set the rate in the branches, and to do the computation afterwards:

```
If original price < 128
   discount rate = 0.92
Else
   discount rate = 0.84
discounted price = discount rate x original price
```

Step 6 Test both branches.

Formulate two test cases, one that fulfills the condition of the `if` statement, and one that does not. Ask yourself what should happen in each case. Then follow the pseudocode and act each of them out.

In our example, let us consider two scenarios for the original price: $100 and $200. We expect that the first price is discounted by $8, the second by $32.

When the original price is 100, then the condition 100 < 128 is true, and we get

```
discount rate = 0.92
discounted price = 0.92 x 100 = 92
```

When the original price is 200, then the condition 200 < 128 is false, and

```
discount rate = 0.84
discounted price = 0.84 x 200 = 168
```

In both cases, we get the expected answer.

Step 7 Assemble the `if` statement in Python.

Type the skeleton

```
if :
else :
```

and fill it in, as shown in Syntax 3.1 on page 94. Omit the `else` branch if it is not needed.

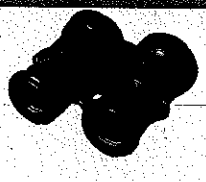
In our example, the completed statement is

```
if originalPrice < 128 :
   discountRate = 0.92
else :
   discountRate = 0.84
discountedPrice = discountRate * originalPrice
```

ch03/sale.py

```
1  ##
2  #  Compute the discount for a given purchase.
3  #
4
5  # Obtain the original price.
6  originalPrice = float(input("Original price before discount: "))
7
8  # Determine the discount rate.
9  if originalPrice < 128 :
10    discountRate = 0.92
11 else :
12    discountRate = 0.84
13
14 # Compute and print the discount.
15 discountedPrice = discountRate * originalPrice
16 print("Discounted price: %.2f" % discountedPrice)
```

WORKED EXAMPLE 3.1 Extracting the Middle



Problem Statement Your task is to extract a string containing the middle character from a given string. For example, if the string is "crate", the result is the string "a". However, if the string has an even number of letters, extract the middle two characters. If the string is "crates", the result is "at".

Step 1 Decide on the branching condition.

We need to take different actions for strings of odd and even length. Therefore, the condition is

Is the length of the string odd?

In Python, you use the remainder of division by 2 to find out whether a value is even or odd. Then the test becomes

len(string) % 2 == 1?

Step 2 Give pseudocode for the work that needs to be done when the condition is true.

We need to find the position of the middle character. If the length is 5, the position is 2.

In general,

position = len(string) / 2 (with the remainder discarded)
result = string[position]

Step 3 Give pseudocode for the work (if any) that needs to be done when the condition is *not* true.

Again, we need to find the position of the middle characters. If the length is 6, the starting position is 2, and the ending position is 3. That is, we would call

result = string[2] + string[3]

In general,

```
position = len(string) / 2 - 1 (with the remainder discarded)
result = string[position] + string[position +1]
```

Step 4 Double-check relational operators.

Do we really want `len(string) % 2 == 1`? For example, when the length is 5, `5 % 2` is the remainder of the division 5 / 2, which is 1. In general, dividing an odd number by 2 leaves a remainder of 1. Therefore, our condition is correct.

Step 5 Remove duplication.

Here is the statement that we have developed:

```
If len(string) % 2 == 1
   position = len(string) / 2 (with remainder discarded)
   result = string[position]
Else
   position = len(string) / 2 - 1 (with remainder discarded)
   result = string[position] + string[position + 1]
```

The first statement in each branch is almost identical. Could we make them the same? We can, if we adjust the position in the second branch:

```
If len(string) % 2 == 1
   position = len(string) / 2 (with remainder discarded)
   result = string[position]
Else
   position = len(string) / 2 (with remainder discarded)
   result = string[position - 1] + string[position]
```

Now we can move the duplicated computation outside the `if` statement:

```
position = len(string) / 2 (with remainder discarded)
If len(string) % 2 == 1
   result = string[position]
Else
   result = string[position - 1] + string[position]
```

Step 6 Test both branches.

We will use a different set of strings for testing. For an odd-length string, consider `"monitor"`. We get

```
position = len(string) / 2 = 7 / 2 = 3 (with remainder discarded)
result = string[3] = "i"
```

For the even-length string `"monitors"`, we get

```
position = len(string) / 2 = 4
result = string[3] + string[4] = "it"
```

Step 7 Assemble the `if` statement in Python.

Here's the completed code segment.

```
position = len(string) // 2
if len(string) % 2 == 1 :
   result = string[position]
else :
   result = string[position - 1] + string[position]
```

3.3 Nested Branches

When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.

It is often necessary to include an `if` statement inside another. Such an arrangement is called a *nested* set of statements.

Here is a typical example: In the United States, different tax rates are used depending on the taxpayer's marital status. There are different tax schedules for single and for married taxpayers. Married taxpayers add their income together and pay taxes on the total. Table 3 gives the tax rate computations, using a simplification of the schedules in effect for the 2008 tax year. A different tax rate applies to each "bracket". In this schedule, the income in the first bracket is taxed at 10 percent, and the income in the second bracket is taxed at 25 percent. The income limits for each bracket depend on the marital status.

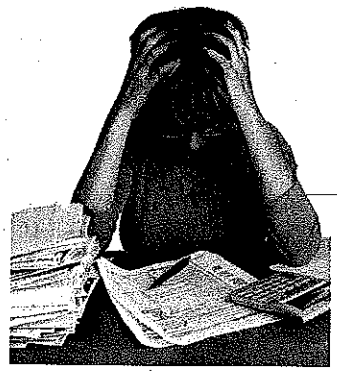
Table 3 Federal Tax Rate Schedule

If your status is Single and if the taxable income is	the tax is	of the amount over
at most $32,000	10%	$0
over $32,000	$3,200 + 25%	$32,000
If your status is Married and if the taxable income is	**the tax is**	**of the amount over**
at most $64,000	10%	$0
over $64,000	$6,400 + 25%	$64,000

Nested decisions are required for problems that have multiple levels of decision making.

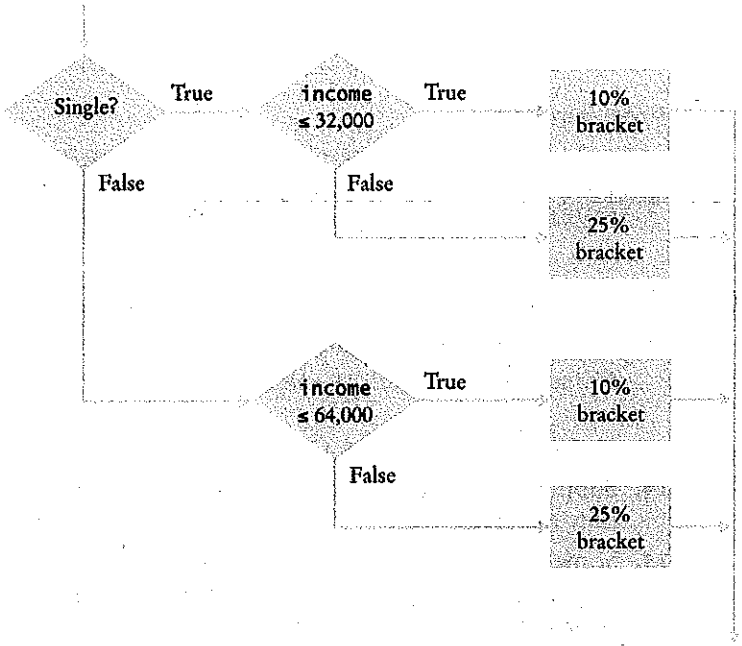
Now compute the taxes due, given a marital status and an income figure. The key point is that there are two *levels* of decision making. First, you must branch on the marital status. Then, for each marital status, you must have another branch on income level.

The two-level decision process is reflected in two levels of `if` statements in the program at the end of this section. (See Figure 3 for a flowchart.) In theory, nesting can go deeper than two levels. A three-level decision process (first by state, then by marital status, then by income level) requires three nesting levels.



Computing income taxes requires multiple levels of decisions.

Figure 3
Income Tax Computation



ch03/taxes.py

```
1  ##
2  #  This program computes income taxes, using a simplified tax schedule.
3  #
4
5  # Initialize constant variables for the tax rates and rate limits.
6  RATE1 = 0.10
7  RATE2 = 0.25
8  RATE1_SINGLE_LIMIT = 32000.0
9  RATE1_MARRIED_LIMIT = 64000.0
10
11 # Read income and marital status.
12 income = float(input("Please enter your income: "))
13 maritalStatus = input("Please enter s for single, m for married: ")
14
15 # Compute taxes due.
16 tax1 = 0.0
17 tax2 = 0.0
18
19 if maritalStatus == "s" :
20    if income <= RATE1_SINGLE_LIMIT :
21       tax1 = RATE1 * income
22    else :
23       tax1 = RATE1 * RATE1_SINGLE_LIMIT
24       tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT)
25 else :
26    if income <= RATE1_MARRIED_LIMIT :
27       tax1 = RATE1 * income
28    else :
29       tax1 = RATE1 * RATE1_MARRIED_LIMIT
30       tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)
31
32 totalTax = tax1 + tax2
33
```

```
34 # Print the results.
35 print("The tax is $%.2f" % totalTax)
```

Program Run

```
Please enter your income: 80000
Please enter s for single, m for married: m
The tax is $10400.00
```

12. What is the amount of tax that a single taxpayer pays on an income of $32,000?
13. Would that amount change if the first nested `if` statement changed from
    `if income <= RATE1_SINGLE_LIMIT :`
    to
    `if income < RATE1_SINGLE_LIMIT :`
14. Suppose Harry and Sally each make $40,000 per year. Would they save taxes if they married?
15. Some people object to higher tax rates for higher incomes, claiming that you might end up with less money after taxes when you get a raise for working hard. What is the flaw in this argument?

Practice It Now you can try these exercises at the end of the chapter: R3.9, P3.20, P3.23.

Programming Tip 3.2

Hand-Tracing

A very useful technique for understanding whether a program works correctly is called *hand-tracing*. You simulate the program's activity on a sheet of paper. You can use this method with pseudocode or Python code.

Get an index card, a cocktail napkin, or whatever sheet of paper is within reach. Make a column for each variable. Have the program code ready. Use a marker, such as a paper clip, to mark the current statement. In your mind, execute statements one at a time. Every time the value of a variable changes, cross out the old value and write the new value below the old one.

Let's trace the taxes.py program on page 107 with the inputs from the program run that follows it. In lines 12 and 13, `income` and `maritalStatus` are initialized by input statements.

Hand-tracing helps you understand whether a program works correctly.

```
 5 # Initialize constant variables for the tax rates and rate limits.
 6 RATE1 = 0.10
 7 RATE2 = 0.25
 8 RATE1_SINGLE_LIMIT = 32000.0
 9 RATE1_MARRIED_LIMIT = 64000.0
10
11 # Read income and marital status.
12 income = float(input("Please enter your income: "))
13 maritalStatus = input("Please enter s for single, m for married: ")
```

tax1	tax2	income	marital status
		80000	m

In lines 16 and 17, `tax1` and `tax2` are initialized to 0.0.

```
16 tax1 = 0.0
17 tax2 = 0.0
```

tax1	tax2	income	marital status
0	0	80000	m

Because `maritalStatus` is not `"s"`, we move to the `else` branch of the outer `if` statement (line 25).

```
19 if maritalStatus == "s" :
20    if income <= RATE1_SINGLE_LIMIT :
21       tax1 = RATE1 * income
22    else :
23       tax1 = RATE1 * RATE1_SINGLE_LIMIT
24       tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT)
25 else :
```

Because `income` is not <= 64000, we move to the `else` branch of the inner `if` statement (line 28).

```
26    if income <= RATE1_MARRIED_LIMIT :
27       tax1 = RATE1 * income
28    else :
29       tax1 = RATE1 * RATE1_MARRIED_LIMIT
30       tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)
```

The values of tax1 and tax2 are updated.

```
28    else :
29       tax1 = RATE1 * RATE1_MARRIED_LIMIT
30       tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)
```

tax1	tax2	income	marital status
0	0	80000	m
6400	4000		

The sum `totalTax` is computed and printed. Then the program ends.

```
32 totalTax = tax1 + tax2
   . . .
35 print("The tax is $%.2f" % totalTax)
```

tax1	tax2	income	marital status	total tax
0	0	80000	m	
6400	4000			10400

Because the program trace shows the expected output ($10,400), it successfully demonstrated that this test case works correctly.

3.4 Multiple Alternatives

Multiple `if` statements can be combined to evaluate complex decisions.

In Section 3.1, you saw how to program a two-way branch with an `if` statement. In many situations, there are more than two cases. In this section, you will see how to implement a decision with multiple alternatives.

For example, consider a program that displays the effect of an earthquake, as measured by the Richter scale (see Table 4).

The 1989 Loma Prieta earthquake that damaged the Bay Bridge in San Francisco and destroyed many buildings measured 7.1 on the Richter scale.

Table 4 Richter Scale

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

The Richter scale is a measurement of the strength of an earthquake. Every step in the scale, for example from 6.0 to 7.0, signifies a tenfold increase in the strength of the quake.

In this case, there are five branches: one each for the four descriptions of damage, and one for no destruction. Figure 4 shows the flowchart for this multiple-branch statement.

You could use multiple `if` statements to implement multiple alternatives, like this:

```
if richter >= 8.0 :
   print("Most structures fall")
else :
   if richter >= 7.0 :
      print("Many buildings destroyed")
   else :
      if richter >= 6.0 :
         print("Many buildings considerably damaged, some collapse")
      else :
         if richter >= 4.5 :
            print("Damage to poorly constructed buildings")
         else :
            print("No destruction of buildings")
```

but this becomes difficult to read and, as the number of branches increases, the code begins to shift further and further to the right due to the required indentation. Python provides the special construct `elif` for creating `if` statements containing multiple branches. Using the `elif` statement, the above code segment can be rewritten as

```
if richter >= 8.0 :
   print("Most structures fall")
elif richter >= 7.0 :
   print("Many buildings destroyed")
elif richter >= 6.0 :
   print("Many buildings considerably damaged, some collapse")
elif richter >= 4.5 :
   print("Damage to poorly constructed buildings")
else :
   print("No destruction of buildings")
```

As soon as one of the four tests succeeds, the effect is displayed, and no further tests are attempted. If none of the four cases applies, the final `else` clause applies, and a default message is printed.

Here you must sort the conditions and test against the largest cutoff first. Suppose we reverse the order of tests:

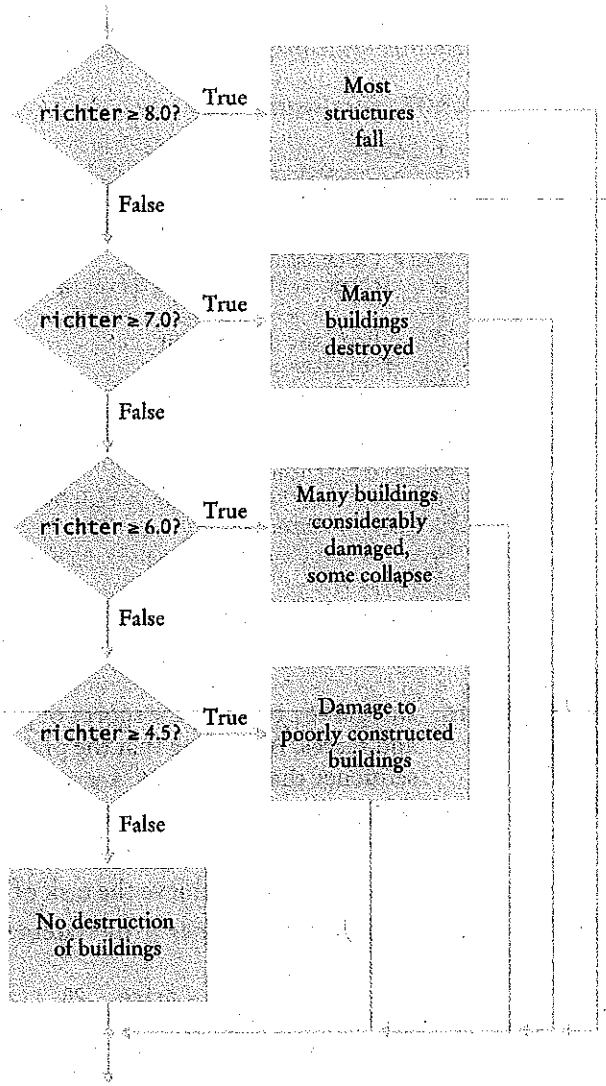
```
if richter >= 4.5 :   # Tests in wrong order
   print("Damage to poorly constructed buildings")
elif richter >= 6.0 :
   print("Many buildings considerably damaged, some collapse")
elif richter >= 7.0 :
   print("Many buildings destroyed")
elif richter >= 8.0 :
   print("Most structures fall")
```

When using multiple `if` statements, test general conditions after more specific conditions.

This does not work. Suppose the value of `richter` is 7.1. That value is at least 4.5, matching the first case. The other tests will never be attempted.

The remedy is to test the more specific conditions first. Here, the condition `richter >= 8.0` is more specific than the condition `richter >= 7.0`, and the condition `richter >= 4.5` is more general (that is, fulfilled by more values) than either of the first two.

Figure 4
Multiple Alternatives



In this example, it is also important that we use an `if/elif` sequence, not just multiple independent `if` statements. Consider this sequence of independent tests.

```
if (richter >= 8.0) :    # Didn't use else
   print("Most structures fall")
if richter >= 7.0 :
   print("Many buildings destroyed")
if richter >= 6.0 :
   print("Many buildings considerably damaged, some collapse")
if richter >= 4.5 :
   print("Damage to poorly constructed buildings")
```

Now the alternatives are no longer exclusive. If `richter` is 7.1, then the last *three* tests all match, and three messages are printed.

The complete program for printing the description of an earthquake given the Richter scale magnitude is provided below.

ch03/earthquake.py

```
##
#  This program prints a description of an earthquake, given the Richter scale magnitude.
#

# Obtain the user input.
richter = float(input("Enter a magnitude on the Richter scale: "))

# Print the description.
if richter >= 8.0 :
   print("Most structures fall")
elif richter >= 7.0 :
   print("Many buildings destroyed")
elif richter >= 6.0 :
   print("Many buildings considerably damaged, some collapse")
elif richter >= 4.5 :
   print("Damage to poorly constructed buildings")
else :
   print("No destruction of buildings")
```

16. In a game program, the scores of players A and B are stored in variables `scoreA` and `scoreB`. Assuming that the player with the larger score wins, write an `if/elif` sequence that prints out `"A won"`, `"B won"`, or `"Game tied"`.
17. Write a conditional statement with three branches that sets s to 1 if x is positive, to –1 if x is negative, and to 0 if x is zero.
18. How could you achieve the task of Self Check 17 with only two branches?
19. Beginners sometimes write statements such as the following:

```
if price > 100 :
   discountedPrice = price - 20
elif price <= 100 :
   discountedPrice = price - 10
```

    Explain how this code can be improved.
20. Suppose the user enters `-1` into the earthquake program. What is printed?
21. Suppose we want to have the earthquake program check whether the user entered a negative number. What branch would you add to the `if` statement, and where?

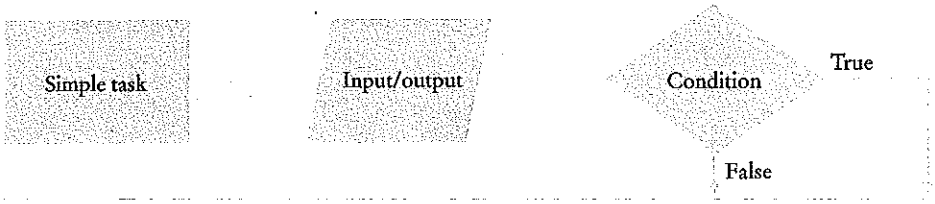
Practice It Now you can try these exercises at the end of the chapter: R3.22, P3.9, P3.34.

3.5 Problem Solving: Flowcharts

Flow charts are made up of elements for tasks, input/output, and decisions.

You have seen examples of flowcharts earlier in this chapter. A flowchart shows the structure of decisions and tasks that are required to solve a problem. When you have to solve a complex problem, it is a good idea to draw a flowchart to visualize the flow of control. The basic flowchart elements are shown in Figure 5.

Figure 5
Flowchart Elements



Each branch of a decision can contain tasks and further decisions.

The basic idea is simple enough. Link tasks and input/output boxes in the sequence in which they should be executed. Whenever you need to make a decision, draw a diamond with two outcomes (see Figure 6).

Each branch can contain a sequence of tasks and even additional decisions. If there are multiple choices for a value, lay them out as in Figure 7.

There is one issue that you need to be aware of when drawing flowcharts. Unconstrained branching and merging can lead to "spaghetti code", a messy network of possible pathways through a program.

Never point an arrow inside another branch.

There is a simple rule for avoiding spaghetti code: Never point an arrow *inside another branch*.

To understand the rule, consider this example: Shipping costs are $5 inside the United States, except that to Hawaii and Alaska they are $10. International shipping costs are also $10.

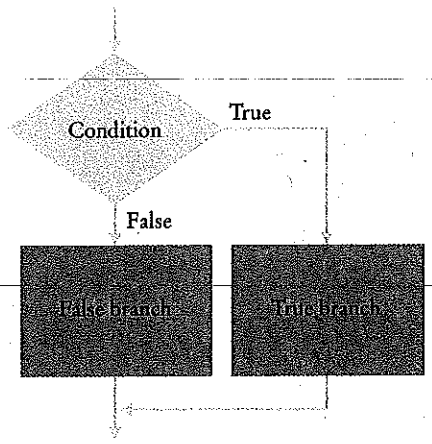


Figure 6 Flowchart with Two Outcomes

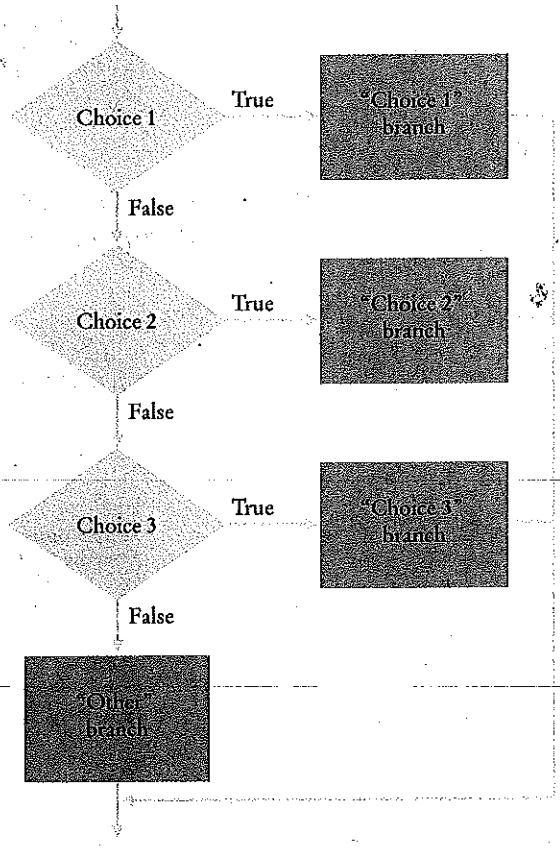
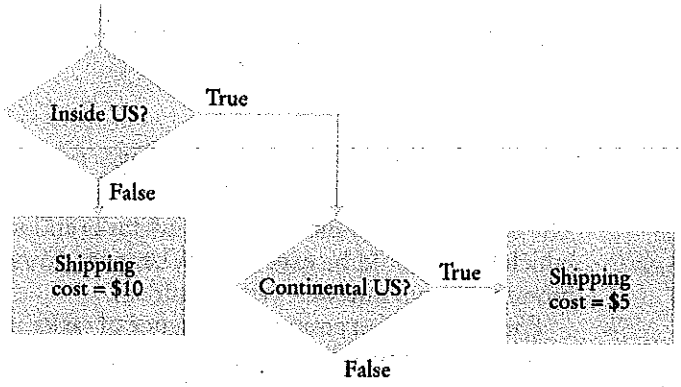
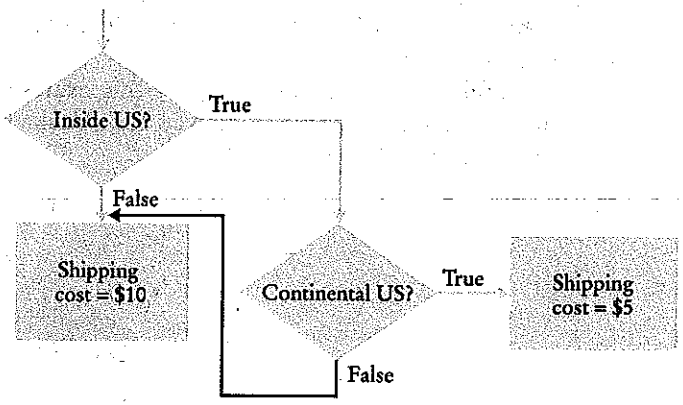


Figure 7 Flowchart with Multiple Choices

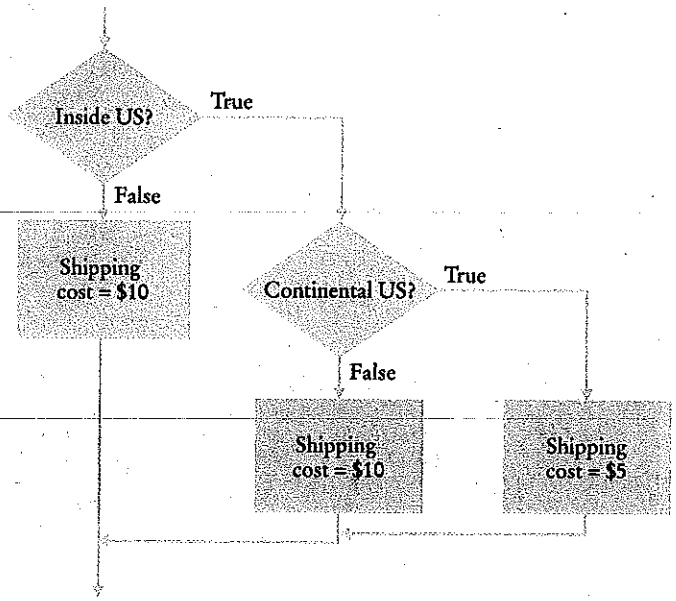
You might start out with a flowchart like the following:



Now you may be tempted to reuse the "shipping cost = $10" task:



Don't do that! The red arrow points inside a different branch. Instead, add another task that sets the shipping cost to $10, like this:



Not only do you avoid spaghetti code, but it is also a better design. In the future it may well happen that the cost for international shipments is different from that to Alaska and Hawaii.

Flowcharts can be very useful for getting an intuitive understanding of the flow of an algorithm. However, they get large rather quickly when you add more details. At that point, it makes sense to switch from flowcharts to pseudocode.

The complete program computing the shipping costs is provided below.

Spaghetti code has so many pathways that it becomes impossible to understand.

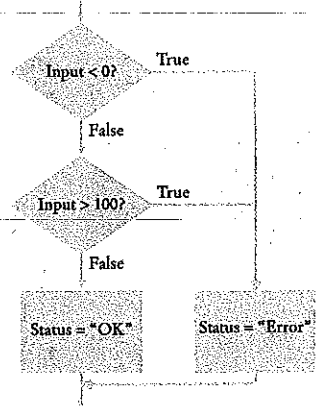
ch03/shipping.py

```
##
#  A program to compute shipping costs.
#

# Obtain the user input.
country = input("Enter the country: ")
state = input("Enter the state or province: ")

# Compute the shipping cost.
shippingCost = 0.0

if country == "USA" :
   if state == "AK" or state == "HI" :    # See Section 3.7 for the or operator
      shippingCost = 10.0
   else :
      shippingCost = 5.0
else :
   shippingCost = 10.0

# Print the results.
print("Shipping cost to %s, %s: $%.2f" % (state, country, shippingCost))
```

Program Run

```
Enter the country: USA
Enter the state or province: VA
Shipping cost to VA, USA: $5.00
```

22. Draw a flowchart for a program that reads a value `temp` and prints "Frozen" if it is less than zero.
23. What is wrong with the flowchart on the right?
24. How do you fix the flowchart of Self Check 23?
25. Draw a flowchart for a program that reads a value `x`. If it is less than zero, print "Error". Otherwise, print its square root.



26. Draw a flowchart for a program that reads a value temp. If it is less than zero, print "Ice". If it is greater than 100, print "Steam". Otherwise, print "Liquid".

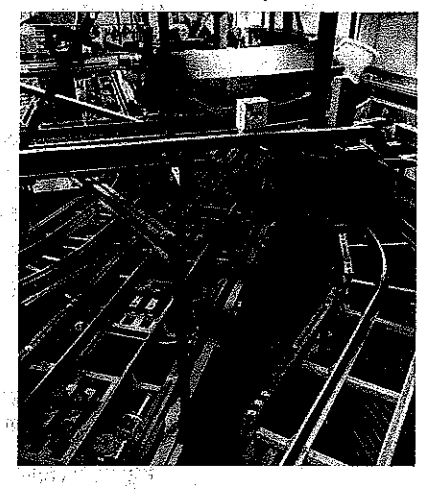
Practice It Now you can try these exercises at the end of the chapter: R3.12, R3.13, R3.14.

Computing & Society 3.1 Denver's Luggage Handling System

Making decisions is an essential part of any computer program. Nowhere is this more obvious than in a computer system that helps sort luggage at an airport. After scanning the luggage identification codes, the system sorts the items and routes them to different conveyor belts. Human operators then place the items onto trucks. When the city of Denver built a huge airport to replace an outdated and congested facility, the luggage system contractor went a step further. The new system was designed to replace the human operators with robotic carts. Unfortunately, the system plainly did not work. It was plagued by mechanical problems, such as luggage falling onto the tracks and jamming carts. Equally frustrating were the software glitches. Carts would uselessly accumulate at some locations when they were needed elsewhere.

The airport had been scheduled to open in 1993, but without a functioning luggage system, the opening was delayed for over a year while the contractor tried to fix the problems. The contractor never succeeded, and ultimately a manual system was installed. The delay cost the city and airlines close to a billion dollars, and the contractor, once the leading luggage systems vendor in the United States, went bankrupt.

Clearly, it is very risky to build a large system based on a technology that has never been tried on a smaller scale. As robots and the software that controls them get better over time, they will take on a larger share of luggage handling in the future. But it is likely that this will happen in an incremental fashion.



The Denver airport originally had a fully automatic system for moving luggage, replacing human operators with robotic carts. Unfortunately, the system never worked and was dismantled before the airport was opened.

3.6 Problem Solving: Test Cases

Consider how to test the tax computation program from Section 3.3. Of course, you cannot try out all possible inputs of marital status and income level. Even if you could, there would be no point in trying them all. If the program correctly computes one or two tax amounts in a given bracket, then we have good reason to believe that all amounts will be correct.

Each branch of your program should be covered by a test case.

You want to aim for complete *coverage* of all decision points. Here is a plan for obtaining a comprehensive set of test cases:

- There are two possibilities for the marital status and two tax brackets for each status, yielding four test cases.
- Test a handful of *boundary* conditions, such as an income that is at the boundary between two brackets, and a zero income.
- If you are responsible for error checking (which is discussed in Section 3.9), also test an invalid input, such as a negative income.

Make a list of the test cases and the expected outputs:

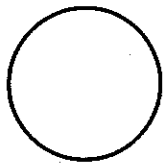
Test Case	Expected Output	Comment
30,000 s	3,000	10% bracket
72,000 s	13,200	3,200 + 25% of 40,000
50,000 m	5,000	10% bracket
104,000 m	16,400	6,400 + 25% of 40,000
32,000 s	3,200	boundary case
0 s	0	boundary case

When you develop a set of test cases, it is helpful to have a flowchart of your program (see Section 3.5). Check off each branch that has a test case. Include test cases for the boundary cases of each decision. For example, if a decision checks whether an input is less than 100, test with an input of 100.

It is always a good idea to design test cases *before* starting to code. Working through the test cases gives you a better understanding of the algorithm that you are about to implement.

It is a good idea to design test cases before implementing a program.

27. Using Figure 1 on page 93 as a guide, follow the process described in Section 3.6 to design a set of test cases for the `elevatorsim.py` program in Section 3.1.
28. What is a boundary test case for the algorithm in How To 3.1 on page 102? What is the expected output?
29. Using Figure 4 on page 111 as a guide, follow the process described in Section 3.6 to design a set of test cases for the `earthquake.py` program in Section 3.3.
30. Suppose you are designing a part of a program for a medical robot that has a sensor returning an x- and y-location (measured in cm). You need to check whether the sensor location is inside the circle, outside the circle, or on the boundary (specifically, having a distance of less than 1 mm from the boundary). Assume the circle has center (0, 0) and a radius of 2 cm. Give a set of test cases.



Practice It Now you can try these exercises at the end of the chapter: R3.15, R3.16.

Programming Tip 3.3

Make a Schedule and Make Time for Unexpected Problems

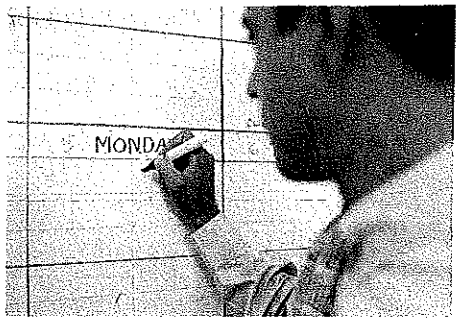
Commercial software is notorious for being delivered later than promised. For example, Microsoft originally promised that its Windows Vista operating system would be available late in 2003, then in 2005, then in March 2006; it finally was released in January 2007. Some of the early promises might not have been realistic. It was in Microsoft's interest to let prospective customers expect the imminent availability of the product. Had customers known the actual delivery date, they might have switched to a different product in the meantime. Undeniably, though, Microsoft had not anticipated the full complexity of the tasks it had set itself to solve.

Microsoft can delay the delivery of its product, but it is likely that you cannot. As a student or a programmer, you are expected to manage your time wisely and to finish your assignments on time. You can probably do simple programming exercises the night before the due date, but an assignment that looks twice as hard may well take four times as long, because more things can go wrong. You should therefore make a schedule whenever you start a programming project.

First, estimate realistically how much time it will take you to:

- Design the program logic.
- Develop test cases.
- Type the program in and fix syntax errors.
- Test and debug the program.

For example, for the income tax program I might estimate an hour for the design; 30 minutes for developing test cases; an hour for data entry and fixing syntax errors; and an hour for testing and debugging. That is a total of 3.5 hours. If I work two hours a day on this project, it will take me almost two days.



Make a schedule for your programming work and build in time for problems.

Then think of things that can go wrong. Your computer might break down. You might be stumped by a problem with the computer system. (That is a particularly important concern for beginners. It is *very* common to lose a day over a trivial problem just because it takes time to track down a person who knows the magic command to overcome it.) As a rule of thumb, *double* the time of your estimate. That is, you should start four days, not two days, before the due date. If nothing went wrong, great; you have the program done two days early. When the inevitable problem occurs, you have a cushion of time that protects you from embarrassment and failure.

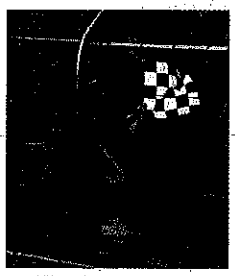
3.7 Boolean Variables and Operators

The Boolean type `bool` has two values, `False` and `True`.

Sometimes, you need to evaluate a logical condition in one part of a program and use it elsewhere. To store a condition that can be true or false, you use a *Boolean variable*. Boolean variables are named after the mathematician George Boole (1815–1864), a pioneer in the study of logic.

In Python, the `bool` data type has exactly two values, denoted `False` and `True`. These values are not strings or integers; they are special values, just for Boolean variables. Here is the initialization of a variable set to `True`:

```
failed = True
```



A Boolean variable is also called a flag because it can be either up (true) or down (false).

You can use the value later in your program to make a decision:

```
if failed :   # Only executed if failed has been set to true
   . . . . . .
```

When you make complex decisions, you often need to combine Boolean values. An operator that combines Boolean conditions is called a **Boolean operator**. In Python, the `and` operator yields `True` only when both conditions are true. The `or` operator yields `True` if at least one of the conditions is true.

Suppose you write a program that processes temperature values, and you want to test whether a given temperature corresponds to liquid water. (At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees.) Water is liquid if the temperature is greater than zero and less than 100:

```
if temp > 0 and temp < 100 :
   print("Liquid")
```

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

A	not A
True	False
False	True

Figure 8 Boolean Truth Tables

Python has two Boolean operators that combine conditions: and and or.

The condition of the test has two parts, joined by the `and` operator. Each part is a Boolean value that can be true or false. The combined expression is true if both individual expressions are true. If either one of the expressions is false, then the result is also false (see Figure 8).

The Boolean operators `and` and `or` have a lower precedence than the relational operators. For that reason, you can write relational expressions on either side of the Boolean operators without using parentheses. For example, in the expression

```
temp > 0 and temp < 100
```

the expressions `temp > 0` and `temp < 100` are evaluated first. Then the `and` operator combines the results. (Appendix B shows a table of the Python operators and their precedences.)

Conversely, let's test whether water is not liquid at a given temperature. That is the case when the temperature is at most 0 or at least 100. Use the `or` operator to combine the expressions:

```
if temp <= 0 or temp >= 100 :
   print("Not liquid")
```

Figure 9 shows flowcharts for these examples.

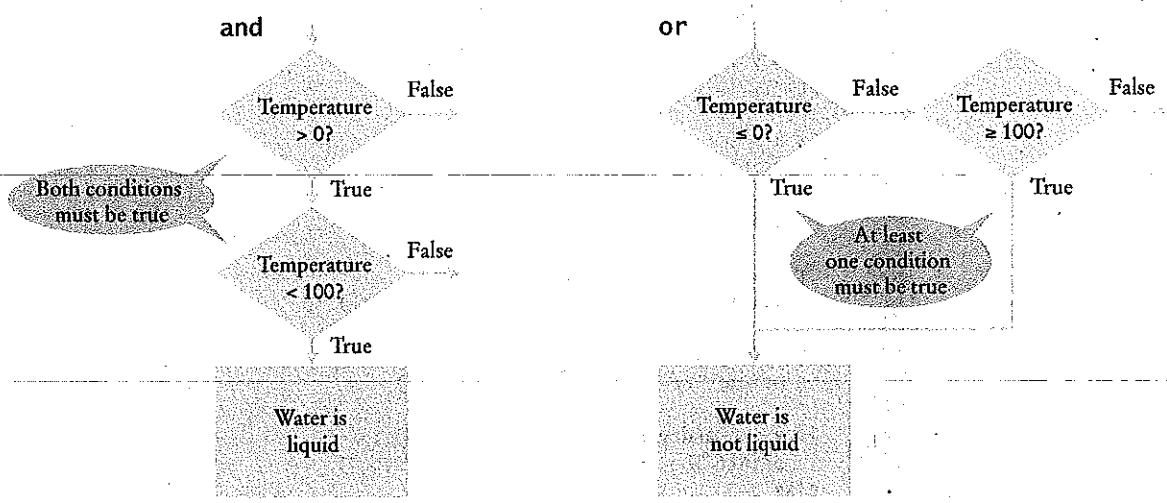
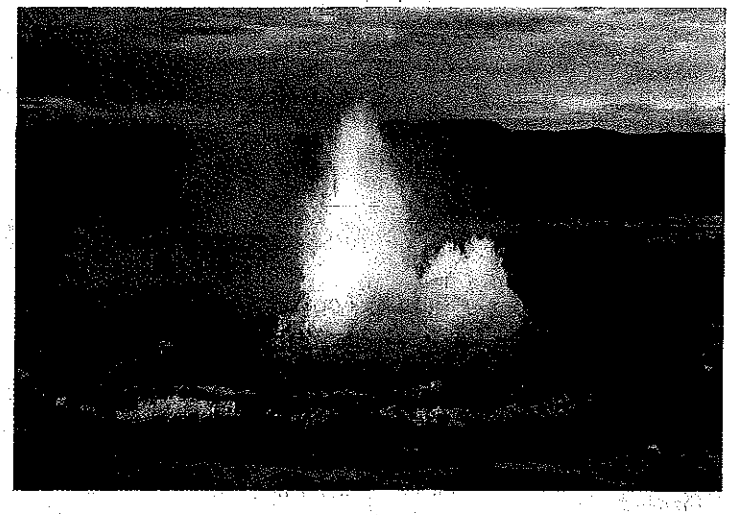


Figure 9 Flowcharts for `and` and `or` Combinations

At this geyser in Iceland, you can see ice, liquid water, and steam.



To invert a condition, use the not operator.

Sometimes you need to *invert* a condition with the `not` Boolean operator. The `not` operator takes a single condition and evaluates to `True` if that condition is false and to `False` if the condition is true. In this example, output occurs if the value of the Boolean variable `frozen` is `False`:

```
if not frozen :
   print("Not frozen")
```

Table 5 illustrates additional examples of evaluating Boolean operators. The following program demonstrates the use of Boolean expressions.

ch03/compare2.py

```
1  ##
2  #  This program demonstrates comparisons of numbers, using Boolean expressions.
3  #
4
5  x = float(input("Enter a number (such as 3.5 or 4.5): "))
6  y = float(input("Enter a second number: "))
7
8  if x == y :
9     print("They are the same.")
10 else :
11    if x > y :
12       print("The first number is larger")
13    else :
14       print("The first number is smaller")
15
16    if -0.01 < x - y and x - y < 0.01 :
17       print("The numbers are close together")
18
19    if x == y + 1 or x == y - 1 :
20       print("The numbers are one apart")
21
22    if x > 0 and y > 0 or x < 0 and y < 0 :
23       print("The numbers have the same sign")
24    else :
25       print("The numbers have different signs")
```

Program Run

```
Enter a number (such as 3.5 or 4.5): 3.25
Enter a second number: -1.02
The first number is larger
The numbers have different signs
```

Table 5 Boolean Operator Examples

Expression	Value	Comment
`0 < 200 and 200 < 100`	`False`	Only the first condition is true.
`0 < 200 or 200 < 100`	`True`	The first condition is true.
`0 < 200 or 100 < 200`	`True`	The `or` is not a test for "either-or". If both conditions are true, the result is true.
`0 < x and x < 100 or x == -1`	`(0 < x and x < 100) or x == -1`	The `and` operator has a higher precedence than the `or` operator (see Appendix B).
`not (0 < 200)`	`False`	`0 < 200` is true, therefore its negation is false.
`frozen == True`	`frozen`	There is no need to compare a Boolean variable with `True`.
`frozen == False`	`not frozen`	It is clearer to use `not` than to compare with `False`.

31. Suppose `x` and `y` are two integers. How do you test whether both of them are zero?
32. How do you test whether at least one of them is zero?
33. How do you test whether *exactly one of them* is zero?
34. What is the value of `not not frozen`?
35. What is the advantage of using the type `bool` rather than strings `"false"`/`"true"` or integers 0/1?

Practice It Now you can try these exercises at the end of the chapter: R3.29, P3.29.

Common Error 3.3

Confusing and and or Conditions

It is a surprisingly common error to confuse `and` and `or` conditions. A value lies between 0 and 100 if it is at least 0 *and* at most 100. It lies outside that range if it is less than 0 *or* greater than 100. There is no golden rule; you just have to think carefully.

Often the `and` or `or` is clearly stated, and then it isn't too hard to implement it. But sometimes the wording isn't as explicit. It is quite common that the individual conditions are nicely set apart in a bulleted list, but with little indication of how they should be combined. Consider these instructions for filing a tax return. You can claim single filing status if any one of the following is true:

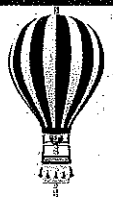
- You were never married.
- You were legally separated or divorced on the last day of the tax year.
- You were widowed, and did not remarry.

Since the test passes if *any one* of the conditions is true, you must combine the conditions with or. Elsewhere, the same instructions state that you may use the more advantageous status of "married filing jointly" if all five of the following conditions are true:

- Your spouse died less than two years ago and you did not remarry.
- You have a child whom you can claim as dependent.
- That child lived in your home for all of the tax year.
- You paid over half the cost of keeping up your home for this child.
- You filed a joint return with your spouse the year he or she died.

Because *all* of the conditions must be true for the test to pass, you must combine them with an and operator.

Programming Tip 3.4



Readability

Programs are more than just instructions to be executed by a computer. A program implements an algorithm and is commonly read by other people. Thus, it is important for your programs not only to be correct but also to be easily read by others. While many programmers focus only on a readable layout for their code, the choice of syntax can also have an impact on the readability.

To help provide readable code, you should never compare against a literal Boolean value (`True` or `False`) in a logical expression. For example, consider the expression in this `if` statement:

```
if frozen == False :
   print("Not frozen")
```

A reader of this code may be confused as to the condition that will cause the `if` statement to be executed. Instead, you should use the more acceptable form

```
if not frozen :
   print("Not frozen")
```

which is easier to read and explicitly states the condition.

It is also important to have appropriate names for variables that contain Boolean values. Choose names such as `done` or `valid`, so that it is clear what action should be taken when the variable is set to `True` or `False`.

Special Topic 3.3

Chaining Relational Operators

In mathematics, it is very common to combine multiple relational operators to compare a variable against multiple values. For example, consider the expression

```
0 <= value <= 100
```

Python also allows you to chain relational operators in this fashion. When the expression is evaluated, the Python interpreter automatically inserts the Boolean operator `and` to form two separate relational expressions

```
value >= 0 and value <= 100
```

Relational operators can be chained arbitrarily. For example, the expression `a < x > b` is perfectly legal. It means the same as `a < x and x > b`. In other words, `x` must exceed both `a` and `b`.

Most programming languages do not allow multiple relational operators to be combined in this fashion; they require explicit Boolean operators. Thus, when first learning to program, it is good practice to explicitly insert the Boolean operators. That way, if you must later change

to a different programming language, you will avoid syntax errors generated by chaining relational operators in a logical expression.

Special Topic 3.4

Short-Circuit Evaluation of Boolean Operators

The `and` and `or` operators are computed using **short-circuit evaluation**. In other words, logical expressions are evaluated from left to right, and evaluation stops as soon as the truth value is determined. When an `and` is evaluated and the first condition is false, the second condition is not evaluated, because it does not matter what the outcome of the second test is.

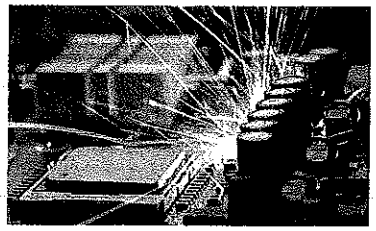
The `and` and `or` operators are computed using *short-circuit evaluation:* As soon as the truth value is determined, no further conditions are evaluated.

For example, consider the expression

```
quantity > 0 and price / quantity < 10
```

Suppose the value of `quantity` is zero. Then the test `quantity > 0` fails, and the second test is not attempted. That is just as well, because it is illegal to divide by zero.

Similarly, when the first condition of an `or` expression is true, then the remainder is not evaluated because the result must be true.



In a short circuit, electricity travels along the path of least resistance. Similarly, short-circuit evaluation takes the fastest path for computing the result of a Boolean expression.

Special Topic 3.5

De Morgan's Law

Humans generally have a hard time comprehending logical conditions with *not* operators applied to *and*/*or* expressions. De Morgan's Law, named after the logician Augustus De Morgan (1806–1871), can be used to simplify these Boolean expressions.

Suppose we want to charge a higher shipping rate if we don't ship within the continental United States.

```
if not (country == "USA" and state != "AK" and state != "HI") :
   shippingCharge = 20.00
```

This test is a little bit complicated, and you have to think carefully through the logic. When it is *not* true that the country is USA *and* the state is not Alaska *and* the state is not Hawaii, then charge $20.00. Huh? It is not true that some people won't be confused by this code.

The computer doesn't care, but it takes human programmers to write and maintain the code. Therefore, it is useful to know how to simplify such a condition.

De Morgan's Law has two forms: one for the negation of an `and` expression and one for the negation of an `or` expression:

`not (A and B)`	is the same as	`not A or not B`
`not (A or B)`	is the same as	`not A and not B`

De Morgan's law tells you how to negate `and` and `or` conditions.

Pay particular attention to the fact that the `and` and `or` operators are *reversed* by moving the `not` inward. For example, the negation of "the state is Alaska *or* it is Hawaii",

```
not (state == "AK" or state == "HI")
```

is "the state is not Alaska *and* it is not Hawaii":

```
state != "AK" and state != "HI"
```

Now apply the law to our shipping charge computation:

```
not (country == "USA" and state != "AK" and state != "HI")
```

is equivalent to

```
not (country == "USA") or not (state != "AK") or not (state != "HI")
```

Because two negatives cancel each other out, the result is the simpler test

```
country != "USA" or state == "AK" or state == "HI"
```

In other words, higher shipping charges apply when the destination is outside the United States or to Alaska or Hawaii.

To simplify conditions with negations of *and* or *or* expressions, it is usually a good idea to apply De Morgan's Law to move the negations to the innermost level.

3.8 Analyzing Strings

Use the `in` operator to test whether a string occurs in another.

Sometimes it is necessary to determine if a string contains a given substring. That is, one string contains an exact match of another string. Given this code segment,

```
name = "John Wayne"
```

the expression

```
"Way" in name
```

yields `True` because the substring `"Way"` occurs within the string stored in variable `name`. Python also provides the inverse of the `in` operator, `not in`:

```
if "-" not in name :
   print("The name does not contain a hyphen.")
```

Sometimes we need to determine not only if a string contains a given substring, but also if the string begins or ends with that substring. For example, suppose you are given the name of a file and need to ensure that it has the correct extension.

```
if filename.endswith(".html") :
   print("This is an HTML file.")
```

The `endswith` string method is applied to the string stored in `filename` and returns `True` if the string ends with the substring `".html"` and `False` otherwise. Table 6 describes additional string methods available for testing substrings.

Table 6 Operations for Testing Substrings

Operation	Description
substring `in` *s*	Returns `True` if the string *s* contains *substring* and `False` otherwise.
s.`count(`*substring*`)`	Returns the number of non-overlapping occurrences of *substring* in the string *s*.
s.`endswith(`*substring*`)`	Returns `True` if the string *s* ends with the substring and `False` otherwise.
s.`find(`*substring*`)`	Returns the lowest index in the string *s* where *substring* begins, or –1 if *substring* is not found.
s.`startswith(`*substring*`)`	Returns `True` if the string *s* begins with *substring* and `False` otherwise.

Table 7 Methods for Testing String Characteristics

Method	Description
`s.isalnum()`	Returns `True` if string *s* consists of only letters or digits and it contains at least one character. Otherwise it returns `False`.
`s.isalpha()`	Returns `True` if string *s* consists of only letters and contains at least one character. Otherwise it returns `False`.
`s.isdigit()`	Returns `True` if string *s* consists of only digits and contains at least one character. Otherwise, it returns `False`.
`s.islower()`	Returns `True` if string *s* contains at least one letter and all letters in the string are lowercase. Otherwise, it returns `False`.
`s.isspace()`	Returns `True` if string *s* consists of only white space characters (blank, newline, tab) and it contains at least one character. Otherwise, it returns `False`.
`s.isupper()`	Returns `True` if string *s* contains at least one letter and all letters in the string are uppercase. Otherwise, it returns `False`.

We can also examine a string to test for specific characteristics. For example, the `islower` string method examines the string and determines if all letters in the string are lowercase. The code segment

```
line = "Four score and seven years ago"
if line.islower() :
   print("The string contains only lowercase letters.")
else :
   print("The string also contains uppercase letters.")
```

prints

```
The string also contains uppercase letters.
```

because the string in `line` begins with an uppercase letter. If the string contains non-letters, they are ignored and do not affect the Boolean result. But what if we need to determine whether a string contains only letters of the alphabet? There is a string method for that as well.

```
if line.isalpha() :
   print("The string is valid.")
else :
   print("The string must contain only upper and lowercase letters.")
```

Python provides several string methods that test for specific characteristics as described in Table 7. Table 8 summarizes how to compare and examine strings in Python.

Table 8 Comparing and Analyzing Strings

Expression	Value	Comment
`"John" == "John"`	`True`	== is also used to test the equality of two strings.
`"John" == "john"`	`False`	For two strings to be equal, they must be identical. An uppercase "J" does not equal a lowercase "j".
`"john" < "John"`	`False`	Based on lexicographical ordering of strings an uppercase "J" comes before a lowercase "j" so the string `"john"` follows the string `"John"`. See Special Topic 3.2 on page 101.
`"john" in "John Johnson"`	`False`	The substring `"john"` must match exactly.
`name = "John Johnson"` `"ho" not in name`	`True`	The string does not contain the substring `"ho"`.
`name.count("oh")`	`2`	All non-overlapping substrings are included in the count.
`name.find("oh")`	`1`	Finds the position or string index where the first substring occurs.
`name.find("ho")`	`-1`	The string does not contain the substring ho.
`name.startswith("john")`	`False`	The string starts with `"John"` but an uppercase "J" does not match a lowercase "j".
`name.isspace()`	`False`	The string contains non-white space characters.
`name.isalnum()`	`False`	The string also contains blank spaces.
`"1729".isdigit()`	`True`	The string only contains characters that are digits.
`"-1729".isdigit()`	`False`	A negative sign is not a digit.

The following program demonstrates the use of operators and methods for examining substrings.

ch03/substrings.py

```
1  ##
2  #  This program demonstrates the various string methods that test substrings.
3  #
4
5  # Obtain a string and substring from the user.
6  theString = input("Enter a string: ")
7  theSubString = input("Enter a substring: ")
8
9  if theSubString in theString :
10    print("The string does contain the substring.")
11
12    howMany = theString.count(theSubString)
13    print("   It contains", howMany, "instance(s)")
14
15    where = theString.find(theSubString)
16    print("   The first occurrence starts at position", where)
```

```
17
18    if theString.startswith(theSubString) :
19       print("   The string starts with the substring.")
20    else :
21       print("   The string does not start with the substring.")
22
23    if theString.endswith(theSubString) :
24       print("   The string ends with the substring.")
25    else :
26       print("   The string does not end with the substring.")
27
28 else :
29    print("The string does not contain the substring.")
```

Program Run

```
Enter a string: The itsy bitsy spider went up the water spout
Enter a substring: itsy
The string does contain the substring.
   It contains 2 instance(s)
   The first occurrence starts at position 4
   The string does not start with the substring.
   The string does not end with the substring.
```

36. How do you determine the number of blank spaces contained in a string?

37. How do you test whether the first character of a string is an uppercase letter?

38. Consider the following statements

```
userStr = "A test question."
index = userStr.find("qu")
```

What is the value of `index`?

39. Assuming variable `userStr` contains the string `"Monty Python"`, what is printed after the following code is executed?

```
print(userStr.isalpha())
```

40. Suppose `userStr` contains a string. How do you test whether the string only contains lowercase letters?

41. How do you test whether a filename (given as a string) has the extension ".jpg" or ".jpeg"?

Practice It Now you can try these exercises at the end of the chapter: P3.17, P3.19.

3.9 Application: Input Validation

An important application for the `if` statement is *input validation*. Whenever your program accepts user input, you need to make sure that the user-supplied values are valid before you use them in your computations.

Like a quality control worker, you want to make sure that user input is correct before processing it.

Consider our elevator simulation program on page 94. Assume that the elevator panel has buttons labeled 1 through 20 (but not 13). The following are illegal inputs:

- The number 13
- Zero or a negative number
- A number larger than 20
- An input that is not a sequence of digits, such as `five`

In each of these cases, we will want to give an error message and exit the program.

It is simple to guard against an input of 13:

```
if floor == 13 :
   print("Error: There is no thirteenth floor.")
```

If the user provides an input that is not in the expected range, print an error message and don't process the input.

Here is how you ensure that the user doesn't enter a number outside the valid range:

```
if floor <= 0 or floor > 20 :
   print("Error: The floor must be between 1 and 20.")
```

However, dealing with an input that is not a valid integer is a more serious problem. When the statement

```
floor = int(input("Floor: "))
```

is executed, and the user types in an input that is not an integer (such as `five`), then the variable `floor` is not set. Instead, a run-time exception occurs and the program is terminated. Python's exception mechanism is needed to help verify integer and floating-point values. We will cover more advanced input verifications in Chapter 7, when exceptions are covered in detail.

Here is a revised elevator simulation program with input validation:

ch03/elevatorsim2.py

```
1  ##
2  #  This program simulates an elevator panel that skips the 13th floor,
3  #  checking for input errors.
4  #
5
6  # Obtain the floor number from the user as an integer.
7  floor = int(input("Floor: "))
8
9  # Make sure the user input is valid.
10 if floor == 13 :
11    print("Error: There is no thirteenth floor.")
12 elif floor <= 0 or floor > 20 :
13    print("Error: The floor must be between 1 and 20.")
14 else :
15    # Now we know that the input is valid.
16    actualFloor = floor
```

```
17    if floor > 13 :
18       actualFloor = floor - 1
19
20    print("The elevator will travel to the actual floor", actualFloor)
```

Program Run

```
Floor: 13
Error: There is no thirteenth floor.
```

Programs that prompt the user to enter a character in order to perform some action or to specify a certain condition are also very common. Consider the income tax computation program from Section 3.3. The user is prompted for marital status and asked to enter a single letter

```
maritalStatus = input("Please enter s for single, m for married: ")
```

Note the specification of lowercase letters for the status. It is common, however, for a user to enter an uppercase letter accidentally or because the caps lock key is on. Instead of flagging this as an error, we can allow the user to enter either an upper- or lowercase letter. When validating the user input, we must compare against both cases:

```
if maritalStatus == "s" or maritalStatus == "S" :
   Process the data for single status
elif maritalStatus == "m" or maritalStatus == "M" :
   Process the data for married status
else :
   print("Error: the marital status must be either s or m.")
```

One-letter inputs are easy to validate by simply comparing against both the upper- and lowercase letters. But what if the user is asked to enter a multi-letter code? For example, in the shipping cost program, the user is asked to enter codes for the country and state or province. In the original version of the program, we only checked the user input against uppercase versions of the codes:

```
if country == "USA" :
   if state == "AK" or state == "HI" :
```

It's not uncommon for a user to enter a multi-letter code using lowercase letters or a mix of upper- and lowercase. It would be tedious to compare the input against all possible combinations of upper- and lowercase letters. Instead, we can first convert the user input to either all upper- or lowercase letters and then compare against a single version. This can be done using the `lower` or `upper` string method.

```
state = input("Enter the state or province: ")
state = state.upper()

country = input("Enter the country: ")
country = country.upper()

if country == "USA" :
   if state == "AK" or state == "HI" :
      Compute the shipping cost.
```

42. In the `elevatorsim2.py` program, what is the output when the input is

a. 100?
b. −1?
c. 20?
d. thirteen?

43. Your task is to rewrite lines 10–13 of the `elevatorsim2.py` program so that there is a single `if` statement with a complex condition. What is the condition?

```
if . . . :
   print("Error: Invalid floor number")
```

44. In the Sherlock Holmes story "The Adventure of the Sussex Vampire", the inimitable detective uttered these words: "Matilda Briggs was not the name of a young woman, Watson, ... It was a ship which is associated with the giant rat of Sumatra, a story for which the world is not yet prepared." Over a hundred years later, researchers found giant rats in Western New Guinea, another part of Indonesia.

Suppose you are charged with writing a program that processes rat weights. It contains the statements

```
weightStr = input("Enter weight in kg: ")
weight = float(weightStr)
```

What input checks should you supply?

When processing inputs, you want to reject values that are too large. But how large is too large? These giant rats, found in Western New Guinea, are about five times the size of a city rat.

45. Run the following test program and supply inputs `2` and `three` at the prompts. What happens? Why?

```
intStr = input("Enter an integer: ")
m = int(intStr)
intStr = input("Enter another integer: ")
n = int(intStr)
print(m, n)
```

Practice It Now you can try these exercises at the end of the chapter: R3.3, R3.31, P3.11.

Special Topic 3.6

Terminating a Program

In text-based programs (those without a graphical user interface) it is common to abort the program if the user enters invalid input. As we saw in the main text, we check the user input and process the data only if valid input was provided. This requires the use of an `if/elif/else` statement to process the data only if the input is valid. This works fine with small programs where the input value is examined only once. But in larger programs, we may need to examine the input value in multiple locations. Instead of having to validate and display an error message each time the input value is used, we can validate the input once and immediately abort the program when invalid data is entered.

The `exit` function defined in the `sys` standard library module immediately aborts the program when executed. An optional message can be displayed to the terminal before the program aborts.

```
from sys import exit

if not (userResponse == "n" or userResponse == "y") :
   exit("Error: you must enter either n or y.")
```

This function, when used as part of the input validation process, can be used to abort the program when an error occurs and to construct cleaner and more readable code.

Special Topic 3.7

Text Input in Graphical Programs

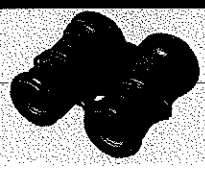
In a program that uses the `graphics` module, you can read and validate user input in the same way as in any other Python program. Simply put calls to the `input` function before the call to the `wait` method. For example,

```
from graphics import GraphicsWindow
from sys import exit
win = GraphicsWindow()
canvas = win.canvas()
x = int(input("Please enter the x coordinate: "))
y = int(input("Please enter the y coordinate: "))
if x < 0 or y < 0 :
   exit("Error: x and y must be >= 0".)
canvas.drawOval(x - 5, y - 5, 10, 10)
win.wait()
```

Worked Example 3.2 shows a more complex graphical application with input validation.

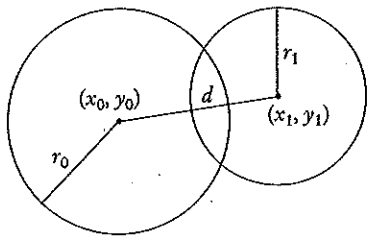
WORKED EXAMPLE 3.2

Intersecting Circles



Problem Statement Develop a graphics program that draws two circles, each defined by its center and radius, and determines whether the two circles intersect.

Given two circles, each defined by a center point and radius, we can determine whether they intersect.



Two circles may intersect at a single point, at two points, or at an unlimited number of points when the two circles are coincident. If the circles do not intersect, one circle may be contained entirely within the other, or the two circles may be completely separate.

Your task is to write a graphics program that obtains the parameters for two circles from the user and draws each circle in the graphics window with a message that reports whether the circles intersect. Each circle should be drawn immediately after its parameters have been input by the user and validated by the program. The result message is to be displayed horizontally centered at the bottom of the window, and should be one of the following:

```
The circles are completely separate.
One circle is contained within the other.
The circles intersect at a single point.
The circles are coincident.
The circles intersect at two points.
```

The center of each circle should be inside the graphics window and the radius should be at least 5 pixels.

Step 1 Determine the data to be extracted from the user and the appropriate input validation tests.

In order to define and draw a circle, the user must enter the x- and y-coordinates of the center point and the radius. Because the circle will be drawn in a graphics window using the `graphics` module, these parameters must be integers.

The data extracted from the user must be validated to ensure that the circles will be visible in the window and large enough to see. The size of the graphics window can be specified at the time it is created.

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

The constant variables used to create the window can also be used to validate the center coordinates. The validation tests required for each set of inputs include

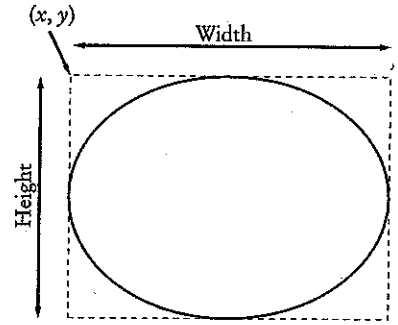
```
If x < 0 or x >= WIN_WIDTH or y < 0 or y >= WIN_HEIGHT
    Exit the program indicating a bad center coordinate.
If radius < MIN_RADIUS
    Exit the program indicating a bad radius size.
```

Step 2 Drawing a circle.

The `graphics` module does not define a method for drawing a circle. But it does define the `drawOval` method:

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

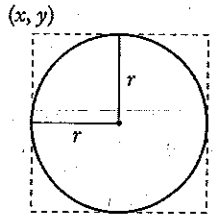
This method requires the coordinates of the upper-left corner and the dimensions (width and height) of the bounding box that encloses the oval.



To draw a circle, we use the same value for the width and height parameters. This will be the diameter of the circle. As a reminder, the diameter of a circle is twice its radius:

```
diameter = 2 x radius
```

Because the user enters the x- and y-coordinates for the center of a circle, we need to compute the coordinates for the upper-left corner of the bounding box.



This is simple because the distance between the center of the circle and the top, or the center and the left side, of the bounding box is equal to the radius of the circle.

```
left side = centerX - radius
top side = centerY - radius
```

Step 3 Determine whether the two circles intersect.

To determine whether the two circles intersect, we must compute the Euclidean distance between the two center points

$$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

and compare it with the radiuses of the two circles as follows:

- if $d > r_0 + r_1$, the two circles do not intersect and are completely separate.
- if $d < |r_0 - r_1|$, the two circles do not intersect and one is contained within the other.
- if $d = r_0 + r_1$, the two circles intersect at a single point.
- if $d = 0$ and $r_0 = r_1$, the two circles are coincident.
- otherwise, the two circles intersect at two points.

With this explanation, the mathematical conditions can be easily converted into algorithmic form for selecting the appropriate message.

```
Set dist to the Euclidean distance between the two center points.
If dist > r0 + r1
    Set message to "The circles are completely separate."
Else If dist < abs(r0 - r1)
    Set message to "One circle is contained within the other."
Else If dist == r0 + r1
    Set message to "The circles intersect at a single point."
Else If dist == 0 and r0 == r1
    Set message to "The circles are coincident."
Else
    Set message to "The circles intersect at two points."
```

Step 4 Determine where the message is to be drawn within the graphical window.

The message has to be displayed horizontally centered at the bottom of the graphical window. The `drawText()` method draws text centered around a given point. For the y-coordinate, a good position is 15 pixels from the bottom of the window. For the x-coordinate, we need the coordinate that is at the horizontal center of the window. Having defined constant variables earlier for the size of the window, specifying the position of the text is rather simple:

```
canvas.drawText(WIN_WIDTH // 2, MIN_HEIGHT - 15)
```

Step 5 Implement your solution in Python.

The complete program is provided below:

ch03/circles.py

```
1  ##
2  #  Draws and determines if two circles intersect. The parameters of both
3  #  circles are obtained from the user.
4  #
5
6  from graphics import GraphicsWindow
7  from math import sqrt
8  from sys import exit
9
10 # Define constant variables.
11 MIN_RADIUS = 5
12 WIN_WIDTH = 500
```

```
WIN_HEIGHT = 500

# Create the graphics window and get the canvas.
win = GraphicsWindow(WIN_WIDTH, WIN_HEIGHT)
canvas = win.canvas()

# Obtain the parameters of the first circle.
print("Enter parameters for the first circle:")
x0 = int(input("  x-coord: "))
y0 = int(input("  y-coord: "))
if x0 < 0 or x0 >= WIN_WIDTH or y0 < 0 or y0 >= WIN_HEIGHT :
   exit("Error: the center of the circle must be within the area of the window.")

r0 = int(input("  radius: "))
if r0 <= MIN_RADIUS :
   exit("Error: the radius must be >", MIN_RADIUS)

# Draw the first circle.
canvas.setOutline("blue")
canvas.drawOval(x0 - r0, y0 - r0, 2 * r0, 2 * r0)

# Obtain the parameters of the second circle.
print("Enter parameters for the second circle:")
x1 = int(input("  x-coord: "))
y1 = int(input("  y-coord: "))
if x1 < 0 or x1 >= WIN_WIDTH or y1 < 0 or y1 >= WIN_HEIGHT :
   exit("Error: the center of the circle must be within the area of the window.")

r1 = int(input("  radius: "))
if r1 <= MIN_RADIUS :
   exit("Error: the radius must be >", MIN_RADIUS)

# Draw the second circle.
canvas.setOutline("red")
canvas.drawOval(x1 - r1, y1 - r1, 2 * r1, 2 * r1)

# Determine if the two circles intersect and select appropriate message.
dist = sqrt((x1 - x0) ** 2 + (y1 - y0) ** 2)

if dist > r0 + r1 :
   message = "The circles are completely separate."
elif dist < abs(r0 - r1) :
   message = "One circle is contained within the other."
elif dist == r0 + r1 :
   message = "The circles intersect at a single point."
elif dist == 0 and r0 == r1 :
   message = "The circles are coincident."
else :
   message = "The circles intersect at two points."

# Display the result at the bottom of the graphics window.
canvas.setOutline("black")
canvas.drawText(WIN_WIDTH // 2, WIN_HEIGHT - 15, message)

# Wait until the user closes the window.
win.wait()
```

Computing & Society 3.2 Artificial Intelligence

When one uses a sophisticated computer program such as a tax preparation package, one is bound to attribute some intelligence to the computer. The computer asks sensible questions and makes computations that we find a mental challenge. After all, if doing one's taxes were easy, we wouldn't need a computer to do it for us.

As programmers, however, we know that all this apparent intelligence is an illusion. Human programmers have carefully "coached" the software in all possible scenarios, and it simply replays the actions and decisions that were programmed into it.

Would it be possible to write computer programs that are genuinely intelligent in some sense? From the earliest days of computing, there was a sense that the human brain might be nothing but an immense computer, and that it might well be feasible to program computers to imitate some processes of human thought. Serious research into *artificial intelligence* began in the mid-1950s, and the first twenty years brought some impressive successes. Programs that play chess—surely an activity that appears to require remarkable intellectual powers—have become so good that they now routinely beat all but the best human players. As far back as 1975, an *expert-system* program called Mycin gained fame for being better at diagnosing meningitis in patients than the average physician.

However, there were serious setbacks as well. From 1982 to 1992, the Japanese government embarked on a massive research project, funded at over 40 billion Japanese yen. It was known as the *Fifth-Generation Project.* Its goal was to develop new hardware and software to greatly improve the performance of expert system software. At its outset, the project created fear in other countries that the Japanese computer industry was about to become the undisputed leader in the field. However, the end results were disappointing and did little to bring artificial intelligence applications to market.

From the very outset, one of the stated goals of the AI community was to produce software that could translate text from one language to another, for example from English to Russian. That undertaking proved to be enormously complicated. Human language appears to be much more subtle and interwoven with the human experience than had originally been thought. Even the grammar-checking tools that come with word-processing programs today are more of a gimmick than a useful tool, and analyzing grammar is just the first step in translating sentences.

The CYC (from encyclopedia) project, started by Douglas Lenat in 1984, tries to codify the implicit assumptions that underlie human speech and writing. The team members started out analyzing news articles and asked themselves what unmentioned facts are necessary to actually understand the sentences. For example, consider the sentence, "Last fall she enrolled in Michigan State". The reader automatically realizes that "fall" is not related to falling down in this context, but refers to the season. While there is a state of Michigan, here Michigan State denotes the university. A priori, a computer program has none of this knowledge. The goal of the CYC project is to extract and store the requisite facts—that is, (1) people enroll in universities; (2) Michigan is a state; (3) many states have universities named X State University, often abbreviated as X State; (4) most people enroll in a university in the fall. By 1995, the project had codified about 100,000 common-sense concepts and about a million facts of knowledge relating them. Even this massive amount of data has not proven sufficient for useful applications.

In recent years, artificial intelligence technology has seen substantial advances. One of the most astounding examples is the outcome of a series of "grand challenges" for autonomous vehicles posed by the Defense Advanced Research Projects Agency (DARPA). Competitors were invited to submit a computer-controlled vehicle that had to complete an obstacle course without a human driver or remote control. The first event, in 2004, was a disappointment, with none of the entrants finishing the route. In 2005, five vehicles completed a grueling 212 km course in the Mojave desert. Stanford's Stanley came in first, with an average speed of 30 km/h. In 2007, DARPA moved the competition to an "urban" environment, an abandoned air force base. Vehicles had to be able to interact with each other, following California traffic laws. As Stanford's Sebastian Thrun explained: "In the last Grand Challenge, it didn't really matter whether an obstacle was a rock or a bush, because either way you'd just drive around it. The current challenge is to move from just sensing the environment to understanding it."

Winner of the 2007 DARPA Urban Challenge

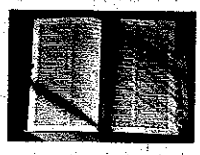
CHAPTER SUMMARY

Use the `if` statement to implement a decision.

- The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.
- Compound statements consist of a header and a statement block.

Implement comparisons of numbers and strings.

- Use relational operators (< <= > >= == !=) to compare numbers and strings.
- The relational operators compare strings in lexicographic order.



Implement decisions whose branches require further decisions.

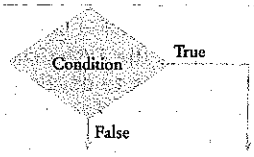
- When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- Nested decisions are required for problems that have multiple levels of decision making.

Implement complex decisions that require multiple `if` statements.

- Multiple `if` statements can be combined to evaluate complex decisions.
- When using multiple `if` statements, test general conditions after more specific conditions.

Draw flowcharts for visualizing the control flow of a program.

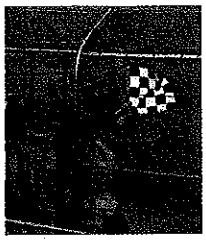
- Flow charts are made up of elements for tasks, input/output, and decisions.
- Each branch of a decision can contain tasks and further decisions.
- Never point an arrow inside another branch.



Design test cases for your programs.

- Each branch of your program should be covered by a test case.
- It is a good idea to design test cases before implementing a program.

Use the Boolean data type to store and combine conditions that can be true or false.



- The Boolean type `bool` has two values, `False` and `True`.
- Python has two Boolean operators that combine conditions: `and` and `or`.
- To invert a condition, use the `not` operator.
- The `and` and `or` operators are computed using *short-circuit evaluation:* As soon as the truth value is determined, no further conditions are evaluated.
- De Morgan's law tells you how to negate `and` and `or` conditions.

Examine strings for specific characteristics.

- Use the `in` operator to test whether a string occurs in another.

Apply `if` statements to detect whether user input is valid.

- If the user provides an input that is not in the expected range, print an error message and don't process the input.

REVIEW QUESTIONS

■ **R3.1** What is the value of each variable after the `if` statement?

a.
```
n = 1
k = 2
r = n
if k < n :
   r = k
```

b.
```
n = 1
k = 2
if n < k :
   r = k
else :
   r = k + n
```

c.
```
n = 1
k = 2
r = k
if r < k :
   n = r
else :
   k = n
```

d.
```
n = 1
k = 2
r = 3
if r < n + k :
   r = 2 * n
else :
   k = 2 * r
```

■■ **R3.2** Explain the difference between

```
s = 0
if x > 0 :
   s = s + 1
if y > 0 :
   s = s + 1
```

and

```
s = 0
if x > 0 :
   s = s + 1
elif y > 0 :
   s = s + 1
```

■■ **R3.3** Find the errors in the following `if` statements.

a.
```
if x > 0 then
   print(x)
```

b.
```
if 1 + x > x ** sqrt(2) :
   y = y + x
```

c.
```
if x = 1 :
   y += 1
```

d.
```
xStr = input("Enter an integer value")
x = int(xStr)
if xStr.isdigit() :
   sum = sum + x
else :
   print("Bad input for x")
```

e.
```
letterGrade = "F"
if grade >= 90 :
   letterGrade = "A"
if grade >= 80 :
   letterGrade = "B"
if grade >= 70 :
   letterGrade = "C"
if grade >= 60 :
   letterGrade = "d"
```

■ **R3.4** What do these code fragments print?

a.
```
n = 1
m = -1
if n < -m :
   print(n)
else :
   print(m)
```

b.
```
n = 1
m = -1
if -n >= m :
   print(n)
else :
   print(m)
```

c.
```
x = 0.0
y = 1.0
if abs(x - y) < 1 :
   print(x)
else :
   print(y)
```

d.
```
x = sqrt(2.0)
y = 2.0
if x * x == y :
   print(x)
else :
   print(y)
```

■■ **R3.5** Suppose x and y are variables each of which contains a number. Write a code fragment that sets y to x if x is positive and to 0 otherwise.

■■ **R3.6** Suppose x and y are variables each of which contains a number. Write a code fragment that sets y to the absolute value of x without calling the `abs` function. Use an `if` statement.

■■ **R3.7** Explain why it is more difficult to compare floating-point numbers than integers. Write Python code to test whether an integer `n` equals 10 and whether a floating-point number `x` is approximately equal to 10.

■ **R3.8** It is easy to confuse the = and == operators. Write a test program containing the statement

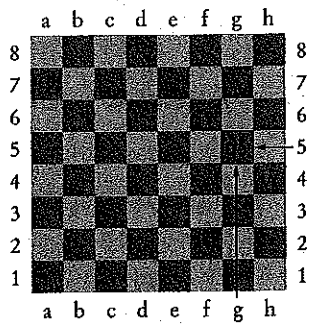
```
if floor = 13
```

What error message do you get? Write another test program containing the statement

```
count == 0
```

What happens when you run the program?

■■ **R3.9** Each square on a chess board can be described by a letter and number, such as `g5` in this example:



The following pseudocode describes an algorithm that determines whether a square with a given letter and number is dark (black) or light (white).

```
If the letter is an a, c, e, or g
   If the number is odd
      color = "black"
   Else
      color = "white"
Else
   If the number is even
      color = "black"
   Else
      color = "white"
```

Using the procedure in Programming Tip 3.2, trace this pseudocode with input `g5`.

■■ **R3.10** Give a set of four test cases for the algorithm of Exercise R3.9 that covers all branches.

■■ **R3.11** In a scheduling program, we want to check whether two appointments overlap. For simplicity, appointments start at a full hour, and we use military time (with

hours 0–24). The following pseudocode describes an algorithm that determines whether the appointment with start time **start1** and end time **end1** overlaps with the appointment with start time **start2** and end time **end2**.

```
If start1 > start2
   s = start1
Else
   s = start2
If end1 < end2
   e = end1
Else
   e = end2
If s < e
   The appointments overlap.
Else
   The appointments don't overlap.
```

Trace this algorithm with an appointment from 10–12 and one from 11–13, then with an appointment from 10–11 and one from 12–13.

■ **R3.12** Draw a flow chart for the algorithm in Exercise R3.11.

■ **R3.13** Draw a flow chart for the algorithm in Exercise P3.18.

■ **R3.14** Draw a flow chart for the algorithm in Exercise P3.20.

■■ **R3.15** Develop a set of test cases for the algorithm in Exercise R3.11.

■■ **R3.16** Develop a set of test cases for the algorithm in Exercise P3.20.

■■ **R3.17** Write pseudocode for a program that prompts the user for a month and day and prints out whether it is one of the following four holidays:

- New Year's Day (January 1)
- Independence Day (July 4)
- Veterans Day (November 11)
- Christmas Day (December 25)

■■ **R3.18** Write pseudocode for a program that assigns letter grades for a quiz, according to the following table:

Score	Grade
90–100	A
80–89	B
70–79	C
60–69	D
< 60	F

■■ **R3.19** Explain how the lexicographic ordering of strings in Python differs from the ordering of words in a dictionary or telephone book. *Hint:* Consider strings such as `IBM`, `wiley.com`, `Century 21`, and `While-U-Wait`.

■■ **R3.20** Of the following pairs of strings, which comes first in lexicographic order?

a. `"Tom", "Jerry"`

b. `"Tom", "Tomato"`

c. `"church", "Churchill"`

d. `"car manufacturer", "carburetor"`
e. `"Harry", "hairy"`
f. `"Python", " Car"`
g. `"Tom", "Tom"`
h. `"Car", "Carl"`
i. `"car", "bar"`

■ **R3.21** Explain the difference between an `if/elif/else` sequence and nested `if` statements. Give an example of each.

■■ **R3.22** Give an example of an `if/elif/else` sequence where the order of the tests does not matter. Give an example where the order of the tests matters.

■ **R3.23** Rewrite the condition in Section 3.4 to use < operators instead of >= operators. What is the impact on the order of the comparisons?

■■ **R3.24** Give a set of test cases for the tax program in Exercise P3.25. Manually compute the expected results.

■■■ **R3.25** Complete the following truth table by finding the truth values of the Boolean expressions for all combinations of the Boolean inputs p, q, and r.

p	q	r	(p and q) or not r	not (p and (q or not r))
False	False	False		
False	False	True		
False	True	False		
. . .				
5 more combinations				
. . .				

■■■ **R3.26** True or false? *A* and *B* is the same as *B* and *A* for any Boolean conditions *A* and *B*.

■ **R3.27** The "advanced search" feature of many search engines allows you to use Boolean operators for complex queries, such as "(cats OR dogs) AND NOT pets". Contrast these search operators with the Boolean operators in Python.

■■ **R3.28** Suppose the value of `b` is `False` and the value of `x` is `0`. What is the value of each of the following expressions?

a. `b and x == 0`
b. `b or x == 0`
c. `not b and x == 0`
d. `not b or x == 0`
e. `b and x != 0`
f. `b or x != 0`
g. `not b and x != 0`
h. `not b or x != 0`

■■ **R3.29** Simplify the following expressions. Here, `b` is a variable of type `bool`.

a. `b == True`

b. `b == False`

c. `b != True`

d. `b != False`

■■■ **R3.30** Simplify the following statements. Here, `b` is a variable that contains a Boolean value and `n` is a variable that contains an integer value.

a.
```
if n == 0 :
   b = True
else :
   b = False
```

b.
```
if n == 0 :
   b = False
else :
   b = True
```

c.
```
b = False
if n > 1 :
   if n < 2 :
      b = True
```

d.
```
if n < 1 :
   b = True
else :
   b = n > 2
```

■ **R3.31** What is wrong with the following program?

```
inputStr = input("Enter the number of quarters: ")
quarters = int(inputStr)
if inputStr.isdigit() :
   total = total + quarters * 0.25
   print("Total: ", total)
else :
   print("Input error.")
```

PROGRAMMING EXERCISES

■ **P3.1** Write a program that reads an integer and prints whether it is negative, zero, or positive.

■■ **P3.2** Write a program that reads a floating-point number and prints "zero" if the number is zero. Otherwise, print "positive" or "negative". Add "small" if the absolute value of the number is less than 1, or "large" if it exceeds 1,000,000.

■■ **P3.3** Write a program that reads an integer and prints how many digits the number has, by checking whether the number is ≥ 10, ≥ 100, and so on. (Assume that all integers are less than ten billion.) If the number is negative, first multiply it by –1.

■■ **P3.4** Write a program that reads three numbers and prints "all the same" if they are all the same, "all different" if they are all different, and "neither" otherwise.

■■ **P3.5** Write a program that reads three numbers and prints "increasing" if they are in increasing order, "decreasing" if they are in decreasing order, and "neither"

otherwise. Here, "increasing" means "strictly increasing", with each value larger than its predecessor. The sequence 3 4 4 would not be considered increasing.

■■ **P3.6** Repeat Exercise P3.5, but before reading the numbers, ask the user whether increasing/decreasing should be "strict" or "lenient". In lenient mode, the sequence 3 4 4 is increasing and the sequence 4 4 4 is both increasing and decreasing.

■■ **P3.7** Write a program that reads in three integers and prints "in order" if they are sorted in ascending *or* descending order, or "not in order" otherwise. For example,

```
1 2 5   in order
1 5 2   not in order
5 2 1   in order
1 2 2   in order
```

■■ **P3.8** Write a program that reads four integers and prints "two pairs" if the input consists of two matching pairs (in some order) and "not two pairs" otherwise. For example,

```
1 2 2 1   two pairs
1 2 2 3   not two pairs
2 2 2 2   two pairs
```

■ **P3.9** Write a program that reads a temperature value and the letter C for Celsius or F for Fahrenheit. Print whether water is liquid, solid, or gaseous at the given temperature at sea level.

■ **P3.10** The boiling point of water drops by about one degree Celsius for every 300 meters (or 1,000 feet) of altitude. Improve the program of Exercise P3.9 to allow the user to supply the altitude in meters or feet.

■ **P3.11** Add error handling to Exercise P3.10. If the user provides an invalid unit for the altitude, print an error message and end the program.

■■ **P3.12** Write a program that translates a letter grade into a number grade. Letter grades are A, B, C, D, and F, possibly followed by + or –. Their numeric values are 4, 3, 2, 1, and 0. There is no F+ or F–. A + increases the numeric value by 0.3, a – decreases it by 0.3. However, an A+ has value 4.0.

```
Enter a letter grade: B-
The numeric value is 2.7.
```

■■ **P3.13** Write a program that translates a number between 0 and 4 into the closest letter grade. For example, the number 2.8 (which might have been the average of several grades) would be converted to B–. Break ties in favor of the better grade; for example 2.85 should be a B.

■■ **P3.14** Write a program that takes user input describing a playing card in the following shorthand notation:

`A`	Ace
`2 ... 10`	Card values
`J`	Jack
`Q`	Queen
`K`	King
`D`	Diamonds
`H`	Hearts
`S`	Spades
`C`	Clubs

Your program should print the full description of the card. For example,

```
Enter the card notation: QS
Queen of Spades
```

■■ **P3.15** Write a program that reads in three floating-point numbers and prints the largest of the three inputs without using the max function. For example:

```
Enter a number: 4
Enter a number: 9
Enter a number: 2.5
The largest number is 9.0
```

■■ **P3.16** Write a program that reads in three strings and sorts them lexicographically.

```
Enter a string: Charlie
Enter a string: Able
Enter a string: Baker
Able
Baker
Charlie
```

■■ **P3.17** Write a program that reads in a string and prints whether it

- contains only letters.
- contains only uppercase letters.
- contains only lowercase letters.
- contains only digits.
- contains only letters and digits.
- starts with an uppercase letter.
- ends with a period.

■■ **P3.18** When two points in time are compared, each given as hours (in military time, ranging from 0 to 23) and minutes, the following pseudocode determines which comes first.

```
If hour1 < hour2
   time1 comes first.
Else if hour1 and hour2 are the same
   If minute1 < minute2
      time1 comes first.
   Else if minute1 and minute2 are the same
      time1 and time2 are the same.
   Else
      time2 comes first.
Else
   time2 comes first.
```

Write a program that prompts the user for two points in time and prints the time that comes first, then the other time.

■ **P3.19** Write a program that prompts the user to provide a single character from the alphabet. Print Vowel or Consonant, depending on the user input. If the user input is not a letter (between a and z or A and Z), or is a string of length > 1, print an error message.

■■ **P3.20** The following algorithm yields the season (Spring, Summer, Fall, or Winter) for a given month and day.

```
If month is 1, 2, or 3, season = "Winter"
Else if month is 4, 5, or 6, season = "Spring"
Else if month is 7, 8, or 9, season = "Summer"
Else if month is 10, 11, or 12, season = "Fall"
If month is divisible by 3 and day >= 21
   If season is "Winter", season = "Spring"
   Else if season is "Spring", season = "Summer"
   Else if season is "Summer", season = "Fall"
   Else season = "Winter"
```

Write a program that prompts the user for a month and day and then prints the season, as determined by this algorithm.

■■ **P3.21** Write a program that reads in two floating-point numbers and tests whether they are the same up to two decimal places. Here are two sample runs.

```
Enter a floating-point number: 2.0
Enter a floating-point number: 1.99998
They are the same up to two decimal places.
Enter a floating-point number: 2.0
Enter a floating-point number: 1.98999
They are different.
```

■■■ **P3.22** Write a program that prompts for the day and month of the user's birthday and then prints a horoscope. Make up fortunes for programmers, like this:

```
Please enter your birthday.
   month: 6
   day: 16
Gemini are experts at figuring out the behavior of complicated programs.
You feel where bugs are coming from and then stay one step ahead. Tonight,
your style wins approval from a tough critic.
```

Each fortune should contain the name of the astrological sign. (You will find the names and date ranges of the signs at a distressingly large number of sites on the Internet.)

■■ **P3.23** The original U.S. income tax of 1913 was quite simple. The tax was

- 1 percent on the first $50,000.
- 2 percent on the amount over $50,000 up to $75,000.
- 3 percent on the amount over $75,000 up to $100,000.
- 4 percent on the amount over $100,000 up to $250,000.
- 5 percent on the amount over $250,000 up to $500,000.
- 6 percent on the amount over $500,000.

There was no separate schedule for single or married taxpayers. Write a program that computes the income tax according to this schedule.

■■■ **P3.24** The `taxes.py` program uses a simplified version of the 2008 U.S. income tax schedule. Look up the tax brackets and rates for the current year, for both single and married filers, and implement a program that computes the actual income tax.

■■■ **P3.25** Write a program that computes taxes for the following schedule.

If your status is Single and if the taxable income is over	but not over	the tax is	of the amount over
$0	$8,000	10%	$0
$8,000	$32,000	$800 + 15%	$8,000
$32,000		$4,400 + 25%	$32,000
If your status is Married and if the taxable income is over	**but not over**	**the tax is**	**of the amount over**
$0	$16,000	10%	$0
$16,000	$64,000	$1,600 + 15%	$16,000
$64,000		$8,800 + 25%	$64,000

■■■ **P3.26** *Unit conversion.* Write a unit conversion program that asks the users from which unit they want to convert (fl. oz, gal, oz, lb, in, ft, mi) and to which unit they want to convert (ml, l, g, kg, mm, cm, m, km). Reject incompatible conversions (such as gal → km). Ask for the value to be converted, then display the result:

```
Convert from? gal
Convert to? ml
Value? 2.5
2.5 gal = 9463.5 ml
```

■■■ **P3.27** A year with 366 days is called a leap year. Leap years are necessary to keep the calendar synchronized with the sun because the earth revolves around the sun once every 365.25 days. Actually, that figure is not entirely precise, and for all dates after 1582 the *Gregorian correction* applies. Usually years that are divisible by 4 are leap years, for example 1996. However, years that are divisible by 100 (for example, 1900) are not leap years, but years that are divisible by 400 are leap years (for example, 2000). Write a program that asks the user for a year and computes whether that year is a leap year. Use a single `if` statement and Boolean operators.

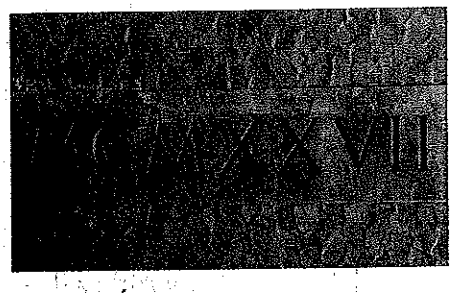
■■■ **P3.28** *Roman numbers.* Write a program that converts a positive integer into the Roman number system. The Roman number system has digits

I	1
V	5
X	10
L	50
C	100
D	500
M	1,000

Numbers are formed according to the following rules:

a. Only numbers up to 3,999 are represented.

b. As in the decimal system, the thousands, hundreds, tens, and ones are expressed separately.



c. The numbers 1 to 9 are expressed as

I	1
II	2
III	3
IV	4
V	5
VI	6
VII	7
VIII	8
IX	9

As you can see, an I preceding a V or X is subtracted from the value, and you can never have more than three I's in a row.

d. Tens and hundreds are done the same way, except that the letters X, L, C and C, D, M are used instead of I, V, X, respectively.

Your program should take an input, such as 1978, and convert it to Roman numerals, MCMLXXVIII.

▪▪ **P3.29** Write a program that asks the user to enter a month (1 for January, 2 for February, and so on) and then prints the number of days in the month. For February, print "28 or 29 days".

```
Enter a month: 5
30 days
```

Do not use a separate `if/else` branch for each month. Use Boolean operators.

▪▪▪ **P3.30** French country names are feminine when they end with the letter e, masculine otherwise, except for the following which are masculine even though they end with e:

- le Belize
- le Cambodge
- le Mexique
- le Mozambique
- le Zaïre
- le Zimbabwe

Write a program that reads the French name of a country and adds the article: le for masculine or la for feminine, such as le Canada or la Belgique.

However, if the country name starts with a vowel, use l'; for example, l'Afghanistan.

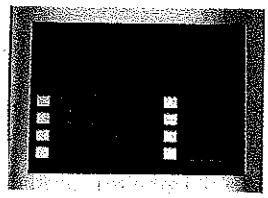
For the following plural country names, use les:

- les Etats-Unis
- les Pays-Bas

▪▪▪ **Business P3.31** Write a program to simulate a bank transaction. There are two bank accounts: checking and savings. First, ask for the initial balances of the bank accounts; reject negative balances. Then ask for the transaction; options are deposit, withdrawal, and transfer. Then ask for the account; options are checking and savings. Then ask for the amount; reject transactions that overdraw an account. At the end, print the balances of both accounts.

Business P3.32 Write a program that reads in the name and salary of an employee. Here the salary will denote an *hourly* wage, such as $9.25. Then ask how many hours the employee worked in the past week. Be sure to accept fractional hours. Compute the pay. Any overtime work (over 40 hours per week) is paid at 150 percent of the regular wage. Print a paycheck for the employee.

Business P3.33 When you use an automated teller machine (ATM) with your bank card, you need to use a personal identification number (PIN) to access your account. If a user fails more than three times when entering the PIN, the machine will block the card. Assume that the user's PIN is "1234" and write a program that asks the user for the PIN no more than three times, and does the following:



- If the user enters the right number, print a message saying, "Your PIN is correct", and end the program.
- If the user enters a wrong number, print a message saying, "Your PIN is incorrect" and, if you have asked for the PIN less than three times, ask for it again.
- If the user enters a wrong number three times, print a message saying "Your bank card is blocked" and end the program.

Business P3.34 A supermarket awards coupons depending on how much a customer spends on groceries. For example, if you spend $50, you will get a coupon worth eight percent of that amount. The following table shows the percent used to calculate the coupon awarded for different amounts spent. Write a program that calculates and prints the value of the coupon a person can receive based on groceries purchased.

Here is a sample run:

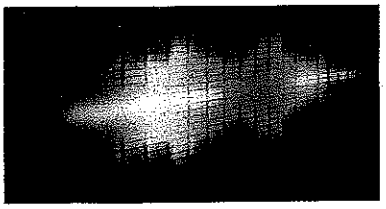
```
Please enter the cost of your groceries: 14
You win a discount coupon of $ 1.12. (8% of your purchase)
```

Money Spent	Coupon Percentage
Less than $10	No coupon
From $10 to $60	8%
More than $60 to $150	10%
More than $150 to $210	12%
More than $210	14%

Business P3.35 Calculating the tip when you go to a restaurant is not difficult, but your restaurant wants to suggest a tip according to the service diners receive. Write a program that calculates a tip according to the diner's satisfaction as follows:

- Ask for the diners' satisfaction level using these ratings: 1 = Totally satisfied, 2 = Satisfied, 3 = Dissatisfied.
- If the diner is totally satisfied, calculate a 20 percent tip.
- If the diner is satisfied, calculate a 15 percent tip.
- If the diner is dissatisfied, calculate a 10 percent tip.
- Report the satisfaction level and tip in dollars and cents.

▪ **Science P3.36** Write a program that prompts the user for a wavelength value and prints a description of the corresponding part of the electromagnetic spectrum, as given in the following table.



Electromagnetic Spectrum

Type	Wavelength (m)	Frequency (Hz)
Radio Waves	$> 10^{-1}$	$< 3 \times 10^{9}$
Microwaves	10^{-3} to 10^{-1}	3×10^{9} to 3×10^{11}
Infrared	7×10^{-7} to 10^{-3}	3×10^{11} to 4×10^{14}
Visible light	4×10^{-7} to 7×10^{-7}	4×10^{14} to 7.5×10^{14}
Ultraviolet	10^{-8} to 4×10^{-7}	7.5×10^{14} to 3×10^{16}
X-rays	10^{-11} to 10^{-8}	3×10^{16} to 3×10^{19}
Gamma rays	$< 10^{-11}$	$> 3 \times 10^{19}$

▪ **Science P3.37** Repeat Exercise P3.36, modifying the program so that it prompts for the frequency instead.

▪▪ **Science P3.38** Repeat Exercise P3.36, modifying the program so that it first asks the user whether the input will be a wavelength or a frequency.

▪▪▪ **Science P3.39** A minivan has two sliding doors. Each door can be opened by either a dashboard switch, its inside handle, or its outside handle. However, the inside handles do not work if a child lock switch is activated. In order for the sliding doors to open, the gear shift must be in park, *and* the master unlock switch must be activated. (This book's author is the long-suffering owner of just such a vehicle.)

Your task is to simulate a portion of the control software for the vehicle. The input is a sequence of values for the switches and the gear shift, in the following order:

- Dashboard switches for left and right sliding door, child lock, and master unlock (0 for off or 1 for activated)
- Inside and outside handles on the left and right sliding doors (0 or 1)
- The gear shift setting (one of P N D 1 2 3 R).

A typical input would be `0 0 0 1 0 1 0 0 P`.

Print "left door opens" and/or "right door opens" as appropriate. If neither door opens, print "both doors stay closed".

▪ **Science P3.40** Sound level L in units of decibel (dB) is determined by

$$L = 20 \log_{10}(p/p_0)$$

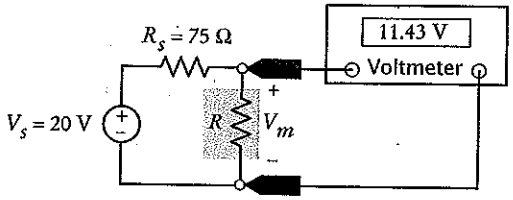
where p is the sound pressure of the sound (in Pascals, abbreviated Pa), and p_0 is a reference sound pressure equal to 20×10^{-6} Pa (where L is 0 dB).

The following table gives descriptions for certain sound levels.

Threshold of pain	130 dB
Possible hearing damage	120 dB
Jack hammer at 1 m	100 dB
Traffic on a busy roadway at 10 m	90 dB
Normal conversation	60 dB
Calm library	30 dB
Light leaf rustling	0 dB

Write a program that reads a value and a unit, either dB or Pa, and then prints the closest description from the list above.

■■ Science P3.41 The electric circuit shown below is designed to measure the temperature of the gas in a chamber.



The resistor R represents a temperature sensor enclosed in the chamber. The resistance R, in Ω, is related to the temperature T, in °C, by the equation

$$R = R_0 + kT$$

In this device, assume $R_0 = 100\ \Omega$ and $k = 0.5$. The voltmeter displays the value of the voltage, V_m, across the sensor. This voltage V_m indicates the temperature, T, of the gas according to the equation

$$T = \frac{R}{k} - \frac{R_0}{k} = \frac{R_s}{k}\frac{V_m}{V_s - V_m} - \frac{R_0}{k}$$

Suppose the voltmeter voltage is constrained to the range $V_{\min} = 12$ volts $\le V_m \le V_{\max} = 18$ volts. Write a program that accepts a value of V_m and checks that it's between 12 and 18. The program should return the gas temperature in degrees Celsius when V_m is between 12 and 18 and an error message when it isn't.

■■■ Science P3.42 Crop damage due to frost is one of the many risks confronting farmers. The figure below shows a simple alarm circuit designed to warn of frost. The alarm circuit uses a device called a thermistor to sound a buzzer when the temperature drops below freezing. Thermistors are semiconductor devices that exhibit a temperature dependent resistance described by the equation

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

where R is the resistance, in Ω, at the temperature T, in °K, and R_0 is the resistance, in Ω, at the temperature T_0, in°K. β is a constant that depends on the material used to make the thermistor.