CHAPTER 3

Decision Structures and Boolean Logic

TOPICS

3.1 The `if` Statement

CONCEPT: The `if` statement is used to create a decision structure, which allows a program to have more than one path of execution. The `if` statement causes one or more statements to execute only when a Boolean expression is true.

A *control structure* is a logical design that controls the order in which a set of statements execute. So far in this book we have used only the simplest type of control structure: the sequence structure. A *sequence structure* is a set of statements that execute in the order that they appear. For example, the following code is a sequence structure because the statements execute from top to bottom.

VideoNote
The `if` Statement

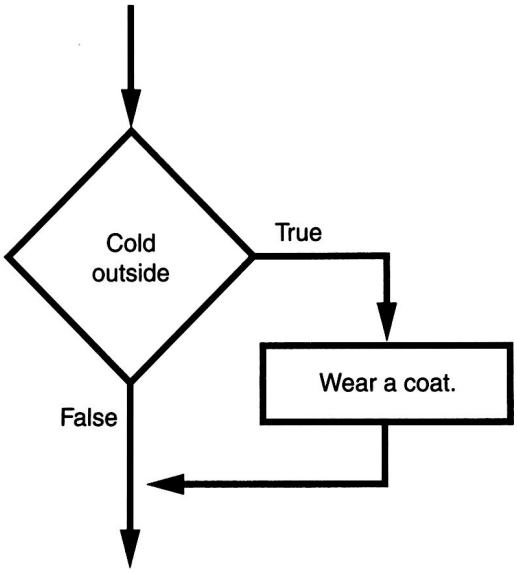
```
name = input('What is your name? ')
age = int(input('What is your age? '))
print('Here is the data you entered:')
print('Name:', name)
print('Age:', age)
```

Although the sequence structure is heavily used in programming, it cannot handle every type of task. This is because some problems simply cannot be solved by performing a set of ordered steps, one after the other. For example, consider a pay calculating program that determines whether an employee has worked overtime. If the employee has worked more than 40 hours, he or she gets paid extra for all the hours over 40. Otherwise, the overtime calculation should be skipped. Programs like this require a different type of control

structure: one that can execute a set of statements only under certain circumstances. This can be accomplished with a *decision structure*. (Decision structures are also known as *selection structures*.)

In a decision structure's simplest form, a specific action is performed only if a certain condition exists. If the condition does not exist, the action is not performed. The flowchart shown in Figure 3-1 shows how the logic of an everyday decision can be diagrammed as a decision structure. The diamond symbol represents a true/false condition. If the condition is true, we follow one path, which leads to an action being performed. If the condition is false, we follow another path, which skips the action.

Figure 3-1 A simple decision structure



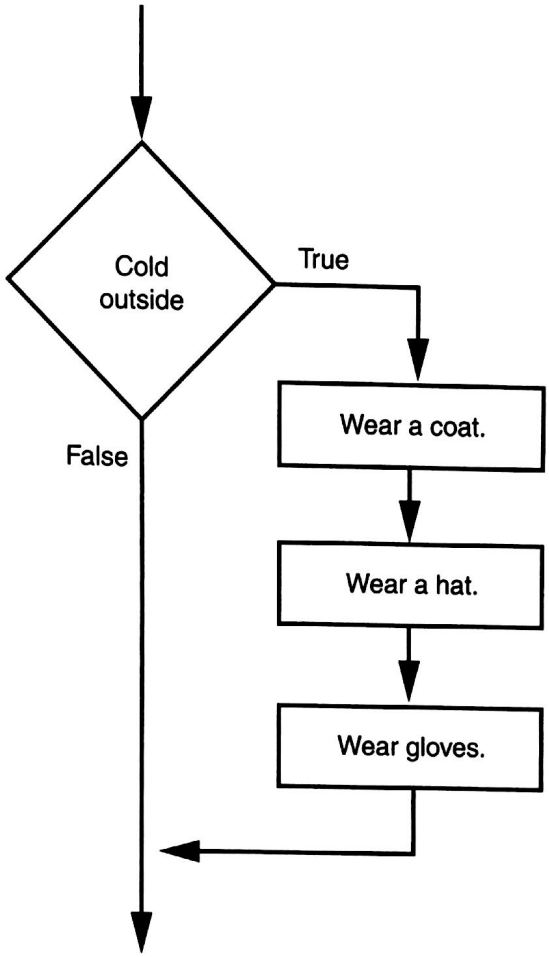
In the flowchart, the diamond symbol indicates some condition that must be tested. In this case, we are determining whether the condition `Cold outside` is true or false. If this condition is true, the action `Wear a coat` is performed. If the condition is false, the action is skipped. The action is *conditionally executed* because it is performed only when a certain condition is true.

Programmers call the type of decision structure shown in Figure 3-1 a *single alternative decision structure*. This is because it provides only one alternative path of execution. If the condition in the diamond symbol is true, we take the alternative path. Otherwise, we exit the structure. Figure 3-2 shows a more elaborate example, where three actions are taken only when it is cold outside. It is still a single alternative decision structure, because there is one alternative path of execution.

In Python we use the `if` statement to write a single alternative decision structure. Here is the general format of the `if` statement:

```
if condition:
    statement
    statement
    etc.
```

Figure 3-2 A decision structure that performs three actions if it is cold outside



For simplicity, we will refer to the first line as the *if clause*. The `if` clause begins with the word `if`, followed by a *`condition`*, which is an expression that will be evaluated as either true or false. A colon appears after the *`condition`*. Beginning at the next line is a block of statements. A *block* is simply a set of statements that belong together as a group. Notice in the general format that all of the statements in the block are indented. This indentation is required because the Python interpreter uses it to tell where the block begins and ends.

When the `if` statement executes, the *`condition`* is tested. If the *`condition`* is true, the statements that appear in the block following the `if` clause are executed. If the condition is false, the statements in the block are skipped.

Boolean Expressions and Relational Operators

As previously mentioned, the `if` statement tests an expression to determine whether it is true or false. The expressions that are tested by the `if` statement are called *Boolean expressions*, named in honor of the English mathematician George Boole. In the 1800s Boole invented a system of mathematics in which the abstract concepts of true and false can be used in computations.

Typically, the Boolean expression that is tested by an `if` statement is formed with a relational operator. A *relational operator* determines whether a specific relationship exists between two values. For example, the greater than operator (>) determines whether one

value is greater than another. The equal to operator (==) determines whether two values are equal. Table 3-1 lists the relational operators that are available in Python.

Table 3-1 Relational operators

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

The following is an example of an expression that uses the greater than (>) operator to compare two variables, `length` and `width`:

```
length > width
```

This expression determines whether the value referenced by `length` is greater than the value referenced by `width`. If `length` is greater than `width`, the value of the expression is true. Otherwise, the value of the expression is false. The following expression uses the less than operator to determine whether `length` is less than `width`:

```
length < width
```

Table 3-2 shows examples of several Boolean expressions that compare the variables `x` and `y`.

Table 3-2 Boolean expressions using relational operators

Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y?
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

You can use the Python interpreter in interactive mode to experiment with these operators. If you type a Boolean expression at the >>> prompt, the interpreter will evaluate the expression and display its value as either `True` or `False`. For example, look at the following interactive session. (We have added line numbers for easier reference.)

```
1  >>> x = 1 Enter
2  >>> y = 0 Enter
3  >>> x > y Enter
```

```
4  True
5  >>> y > x
6  False
7  >>>
```

The statement in line 1 assigns the value 1 to the variable x. The statement in line 2 assigns the value 0 to the variable y. In line 3, we type the Boolean expression x > y. The value of the expression (True) is displayed in line 4. Then, in line 5, we type the Boolean expression y > x. The value of the expression (False) is displayed in line 6.

The following interactive session demonstrates the < operator:

```
1  >>> x = 1 Enter
2  >>> y = 0 Enter
3  >>> y < x Enter
4  True
5  >>> x < y Enter
6  False
7  >>>
```

The statement in line 1 assigns the value 1 to the variable x. The statement in line 2 assigns the value 0 to the variable y. In line 3, we type the Boolean expression y < x. The value of the expression (True) is displayed in line 4. Then, in line 5, we type the Boolean expression x < y. The value of the expression (False) is displayed in line 6.

The >= and <= Operators

Two of the operators, >= and <=, test for more than one relationship. The >= operator determines whether the operand on its left is greater than *or* equal to the operand on its right. The <= operator determines whether the operand on its left is less than *or* equal to the operand on its right.

For example, look at the following interactive session:

```
1  >>> x = 1 Enter
2  >>> y = 0 Enter
3  >>> z = 1 Enter
4  >>> x >= y Enter
5  True
6  >>> x >= z Enter
7  True
8  >>> x <= z Enter
9  True
10 >>> x <= y Enter
11 False
12 >>>
```

In lines 1 through 3, we assign values to the variables x, y, and z. In line 4 we enter the Boolean expression x >= y, which is True. In line 6 we enter the Boolean expression x >= z, which is True. In line 8 we enter the Boolean expression x <= z, which is True. In line 10 we enter the Boolean expression x <= y, which is False.

The == Operator

The == operator determines whether the operand on its left is equal to the operand on its right. If the values referenced by both operands are the same, the expression is true. Assuming that a is 4, the expression a == 4 is true and the expression a == 2 is false.

The following interactive session demonstrates the == operator:

```
>>> x = 1 [Enter]
>>> y = 0 [Enter]
>>> z = 1 [Enter]
>>> x == y [Enter]
False
>>> x == z [Enter]
True
>>>
```

NOTE: The equality operator is two = symbols together. Don't confuse this operator with the assignment operator, which is one = symbol.

The != Operator

The != operator is the not-equal-to operator. It determines whether the operand on its left is not equal to the operand on its right, which is the opposite of the == operator. As before, assuming a is 4, b is 6, and c is 4, both a != b and b != c are true because a is not equal to b and b is not equal to c. However, a != c is false because a is equal to c.

The following interactive session demonstrates the != operator:

```
>>> x = 1 [Enter]
>>> y = 0 [Enter]
>>> z = 1 [Enter]
>>> x != y [Enter]
True
>>> x != z [Enter]
False
>>>
```

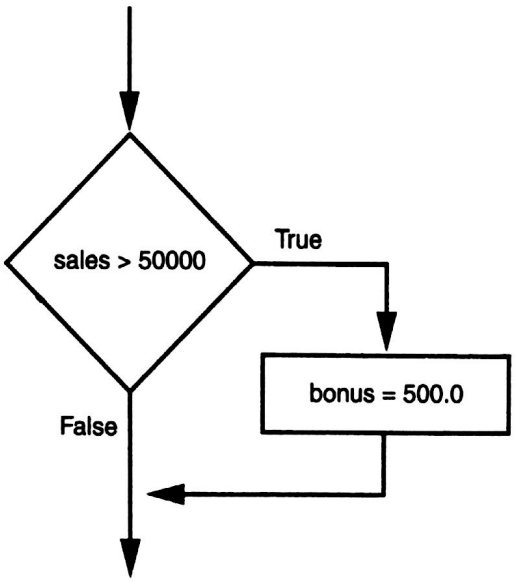
Putting It All Together

Let's look at the following example of the if statement:

```
if sales > 50000:
    bonus = 500.0
```

This statement uses the > operator to determine whether sales is greater than 50,000. If the expression sales > 50000 is true, the variable bonus is assigned 500.0. If the expression is false, however, the assignment statement is skipped. Figure 3-3 shows a flowchart for this section of code.

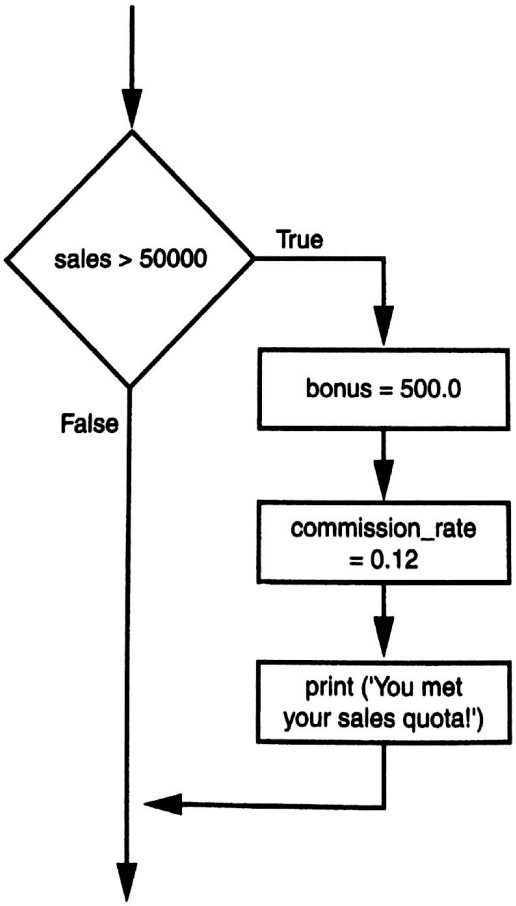
Figure 3-3 Example decision structure



The following example conditionally executes a block containing three statements. Figure 3-4 shows a flowchart for this section of code.

```
if sales > 50000:
    bonus = 500.0
    commission_rate = 0.12
    print('You met your sales quota!')
```

Figure 3-4 Example decision structure



The following code uses the == operator to determine whether two values are equal. The expression `balance == 0` will be true if the `balance` variable is assigned 0. Otherwise the expression will be false.

```
if balance == 0:
    # Statements appearing here will
    # be executed only if balance is
    # equal to 0.
```

The following code uses the != operator to determine whether two values are *not* equal. The expression `choice != 5` will be true if the `choice` variable does not reference the value 5. Otherwise the expression will be false.

```
if choice != 5:
    # Statements appearing here will
    # be executed only if choice is
    # not equal to 5.
```

In the Spotlight:

Using the `if` Statement

Kathryn teaches a science class and her students are required to take three tests. She wants to write a program that her students can use to calculate their average test score. She also wants the program to congratulate the student enthusiastically if the average is greater than 95. Here is the algorithm in pseudocode:

Get the first test score
Get the second test score
Get the third test score
Calculate the average
Display the average
If the average is greater than 95:
  Congratulate the user

Program 3-1 shows the code for the program.

Program 3-1 (test_average.py)

```
1  # This program gets three test scores and displays
2  # their average. It congratulates the user if the
3  # average is a high score.
4
5  # The high score variable holds the value that is
6  # considered a high score.
7  high_score = 95
8
9  # Get the three test scores.
```

```
10  test1 = int(input('Enter the score for test 1:' ))
11  test2 = int(input('Enter the score for test 2:' ))
12  test3 = int(input('Enter the score for test 3:' ))
13
14  # Calculate the average test score.
15  average = (test1 + test2 + test3) / 3
16
17  # Print the average.
18  print('The average score is', average)
19
20  # If the average is a high score,
21  # congratulate the user.
22  if average >= high_score:
23      print('Congratulations!')
24      print('That is a great average!')
```

Program Output (with input shown in bold)

```
Enter the score for test 1: 82 Enter
Enter the score for test 2: 76 Enter
Enter the score for test 3: 91 Enter
The average score is 83.0
```

Program Output (with input shown in bold)

```
Enter the score for test 1: 93 Enter
Enter the score for test 2: 99 Enter
Enter the score for test 3: 96 Enter
The average score is 96.0
Congratulations!
That is a great average!
```

Checkpoint

3.1 What is a control structure?

3.2 What is a decision structure?

3.3 What is a single alternative decision structure?

3.4 What is a Boolean expression?

3.5 What types of relationships between values can you test with relational operators?

3.6 Write an `if` statement that assigns 0 to `x` if `y` is equal to 20.

3.7 Write an `if` statement that assigns 0.2 to `commissionRate` if `sales` is greater than or equal to 10000.

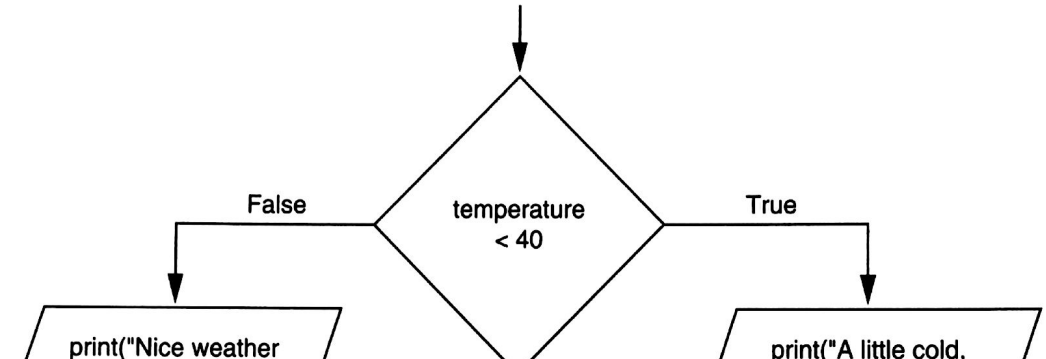
3.2 The `if-else` Statement

CONCEPT: **An `if-else` statement will execute one block of statements if its condition is true, or another block if its condition is false.**


VideoNote
The `if-else` Statement

The previous section introduced the single alternative decision structure (the `if` statement), which has one alternative path of execution. Now we will look at the *dual alternative decision structure*, which has two possible paths of execution—one path is taken if a condition is true, and the other path is taken if the condition is false. Figure 3-5 shows a flowchart for a dual alternative decision structure.

Figure 3-5 A dual alternative decision structure



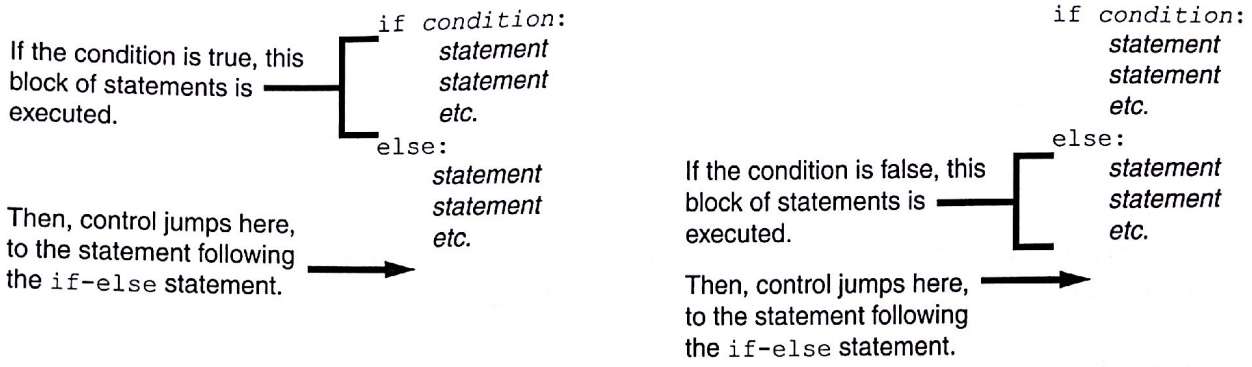
The decision structure in the flowchart tests the condition `temperature < 40`. If this condition is true, the statement `print("A little cold, isn't it?")` is performed. If the condition is false, the statement `print("Nice weather we're having.")` is performed.

In code we write a dual alternative decision structure as an `if-else` statement. Here is the general format of the `if-else` statement:

```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
```

When this statement executes, the *condition* is tested. If it is true, the block of indented statements following the `if` clause is executed, and then control of the program jumps to the statement that follows the `if-else` statement. If the condition is false, the block of indented statements following the `else` clause is executed, and then control of the program jumps to the statement that follows the `if-else` statement. This action is described in Figure 3-6.

Figure 3-6 Conditional execution in an if-else statement



The following code shows an example of an if-else statement. This code matches the flowchart that was shown in Figure 3-5.

```
if temperature < 40:
    print("A little cold, isn't it?")
else:
    print("Nice weather we're having.")
```

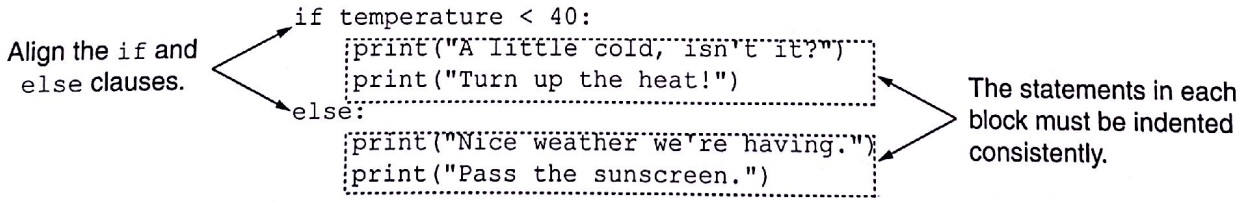
Indentation in the if-else Statement

When you write an if-else statement, follow these guidelines for indentation:

- Make sure the if clause and the else clause are aligned.
- The if clause and the else clause are each followed by a block of statements. Make sure the statements in the blocks are consistently indented.

This is shown in Figure 3-7.

Figure 3-7 Indentation with an if-else statement



In the Spotlight:

Using the if-else Statement

Chris owns an auto repair business and has several employees. If any employee works over 40 hours in a week, he pays them 1.5 times their regular hourly pay rate for all hours over 40. He has asked you to design a simple payroll program that calculates an employee's gross pay, including any overtime wages. You design the following algorithm:

Get the number of hours worked.
Get the hourly pay rate.

If the employee worked more than 40 hours:
  Calculate and display the gross pay with overtime.
Else:
  Calculate and display the gross pay as usual.

The code for the program is shown in Program 3-2. Notice that two variables are created in lines 3 and 4. The `base_hours` variable is assigned 40, which is the number of hours an employee can work in a week without getting paid overtime. The `ot_multiplier` variable is assigned 1.5, which is the pay rate multiplier for overtime hours. This means that the employee's hourly pay rate is multiplied by 1.5 for all overtime hours.

Program 3-2 (auto_repair_payroll.py)

```
1  # Variables to represent the base hours and
2  # the overtime multiplier.
3  base_hours = 40      # Base hours per week
4  ot_multiplier = 1.5 # Overtime multiplier
5
6  # Get the hours worked and the hourly pay rate.
7  hours = float(input('Enter the number of hours worked:'))
8  pay_rate = float(input('Enter the hourly pay rate:'))
9
10 # Calculate and display the gross pay.
11 if hours > base_hours:
12      # Calculate the gross pay with overtime.
13      # First, get the number of overtime hours worked.
14      overtime_hours = hours - base_hours
15
16      # Calculate the amount of overtime pay.
17      overtime_pay = overtime_hours * pay_rate * ot_multiplier
18
19      # Calculate the gross pay.
20      gross_pay = base_hours * pay_rate + overtime_pay
21 else:
22      # Calculate the gross pay without overtime.
23      gross_pay = hours * pay_rate
24
25 # Display the gross pay.
26 print('The gross pay is $', format(gross_pay, ',.2f'), sep='')
```

Program Output (with input shown in bold)

```
Enter the number of hours worked: 40 [Enter]
Enter the hourly pay rate: 20 [Enter]
The gross pay is $800.00.
```

Program Output (with input shown in bold)

```
Enter the number of hours worked: 50 [Enter]
Enter the hourly pay rate: 20 [Enter]
The gross pay is $1,100.00.
```

Checkpoint

3.8 How does a dual alternative decision structure work?

3.9 What statement do you use in Python to write a dual alternative decision structure?

3.10 When you write an `if-else` statement, under what circumstances do the statements that appear after the `else` clause execute?

3.3 Comparing Strings

CONCEPT: Python allows you to compare strings. This allows you to create decision structures that test the value of a string.

You saw in the preceding examples how numbers can be compared in a decision structure. You can also compare strings. For example, look at the following code:

```
name1 = 'Mary'
name2 = 'Mark'
if name1 == name2:
    print('The names are the same.')
else:
    print('The names are NOT the same.')
```

The == operator compares `name1` and `name2` to determine whether they are equal. Because the strings `'Mary'` and `'Mark'` are not equal, the `else` clause will display the message `'The names are NOT the same.'`

Let's look at another example. Assume the `month` variable references a string. The following code uses the `!=` operator to determine whether the value referenced by `month` is not equal to `'October'`.

```
if month != 'October':
    print('This is the wrong time for Octoberfest!')
```

Program 3-3 is a complete program demonstrating how two strings can be compared. The program prompts the user to enter a password and then determines whether the string entered is equal to `'prospero'`.

Program 3-3 (password.py)

```
1  # This program compares two strings.
2  # Get a password from the user.
3  password = input('Enter the password:')
4
5  # Determine whether the correct password
6  # was entered.
7  if password == 'prospero':
8      print('Password accepted.')
9  else:
10     print('Sorry, that is the wrong password.')
```

Program Output (with input shown in bold)

```
Enter the password: ferdinand [Enter]
Sorry, that is the wrong password.
```

Program Output (with input shown in bold)

```
Enter the password: prospero [Enter]
Password accepted.
```

String comparisons are case sensitive. For example, the strings `'saturday'` and `'Saturday'` are not equal because the `"s"` is lowercase in the first string, but uppercase in the second string. The following sample session with Program 4-3 shows what happens when the user enters `Prospero` as the password (with an uppercase `P`).

Program Output (with input shown in bold)

```
Enter the password: Prospero [Enter]
Sorry, that is the wrong password.
```

TIP: In Chapter 8 you will learn how to manipulate strings so that case-insensitive comparisons can be performed.

Other String Comparisons

In addition to determining whether strings are equal or not equal, you can also determine whether one string is greater than or less than another string. This is a useful capability because programmers commonly need to design programs that sort strings in some order.

Recall from Chapter 1 that computers do not actually store characters, such as A, B, C, and so on, in memory. Instead, they store numeric codes that represent the characters. Chapter 1 mentioned that ASCII (the American Standard Code for Information Interchange) is a commonly used character coding system. You can see the set of ASCII codes in Appendix C, but here are some facts about it:

- The uppercase characters A through Z are represented by the numbers 65 through 90.
- The lowercase characters a through z are represented by the numbers 97 through 122.
- When the digits 0 through 9 are stored in memory as characters, they are represented by the numbers 48 through 57. (For example, the string `'abc123'` would be stored in memory as the codes 97, 98, 99, 49, 50, and 51.)
- A blank space is represented by the number 32.

In addition to establishing a set of numeric codes to represent characters in memory, ASCII also establishes an order for characters. The character "A" comes before the character "B", which comes before the character "C", and so on.

When a program compares characters, it actually compares the codes for the characters. For example, look at the following `if` statement:

```
if 'a' < 'b':
    print('The letter a is less than the letter b.')
```

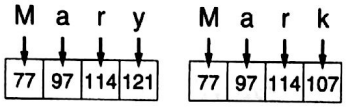
This code determines whether the ASCII code for the character `'a'` is less than the ASCII code for the character `'b'`. The expression `'a' < 'b'` is true because the code for `'a'` is less than the code for `'b'`. So, if this were part of an actual program it would display the message `'The letter a is less than the letter b.'`

Let's look at how strings containing more than one character are typically compared. Suppose a program uses the strings `'Mary'` and `'Mark'` as follows:

```
name1 = 'Mary'
name2 = 'Mark'
```

Figure 3-8 shows how the individual characters in the strings `'Mary'` and `'Mark'` would actually be stored in memory, using ASCII codes.

Figure 3-8 Character codes for the strings `'Mary'` and `'Mark'`

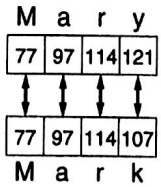


When you use relational operators to compare these strings, the strings are compared character-by-character. For example, look at the following code:

```
name1 = 'Mary'
name2 = 'Mark'
if name1 > name2:
    print('Mary is greater than Mark')
else:
    print('Mary is not greater than Mark')
```

The > operator compares each character in the strings `'Mary'` and `'Mark'`, beginning with the first, or leftmost, characters. This is shown in Figure 3-9.

Figure 3-9 Comparing each character in a string



Here is how the comparison takes place:

1. The `'M'` in `'Mary'` is compared with the `'M'` in `'Mark'`. Since these are the same, the next characters are compared.
2. The `'a'` in `'Mary'` is compared with the `'a'` in `'Mark'`. Since these are the same, the next characters are compared.
3. The `'r'` in `'Mary'` is compared with the `'r'` in `'Mark'`. Since these are the same, the next characters are compared.
4. The `'y'` in `'Mary'` is compared with the `'k'` in `'Mark'`. Since these are not the same, the two strings are not equal. The character `'y'` has a higher ASCII code (121) than `'k'` (107), so it is determined that the string `'Mary'` is greater than the string `'Mark'`.

If one of the strings in a comparison is shorter than the other, only the corresponding characters will be compared. If the corresponding characters are identical, then the shorter string is considered less than the longer string. For example, suppose the strings `'High'` and `'Hi'` were being compared. The string `'Hi'` would be considered less than `'High'` because it is shorter.

Program 3-4 shows a simple demonstration of how two strings can be compared with the < operator. The user is prompted to enter two names, and the program displays those two names in alphabetical order.

Program 3-4 (sort_names.py)

```
1  # This program compares strings with the < operator.
2  # Get two names from the user.
3  name1 = input('Enter a name (last name first):')
4  name2 = input('Enter another name (last name first):')
5
6  # Display the names in alphabetical order.
7  print('Here are the names, listed alphabetically.')
8
9  if name1 < name2:
10     print(name1)
11     print(name2)
12 else:
13     print(name2)
14     print(name1)
```

Program Output (with input shown in bold)

```
Enter a name (last name first): Jones, Richard Enter
Enter another name (last name first) Costa, Joan Enter
Here are the names, listed alphabetically:
Costa, Joan
Jones, Richard
```

Checkpoint

3.11 What would the following code display?

```
if 'z' < 'a':
    print('z is less than a.')
else:
    print('z is not less than a.')
```

3.12 What would the following code display?

```
s1 = 'New York'
s2 = 'Boston'
```

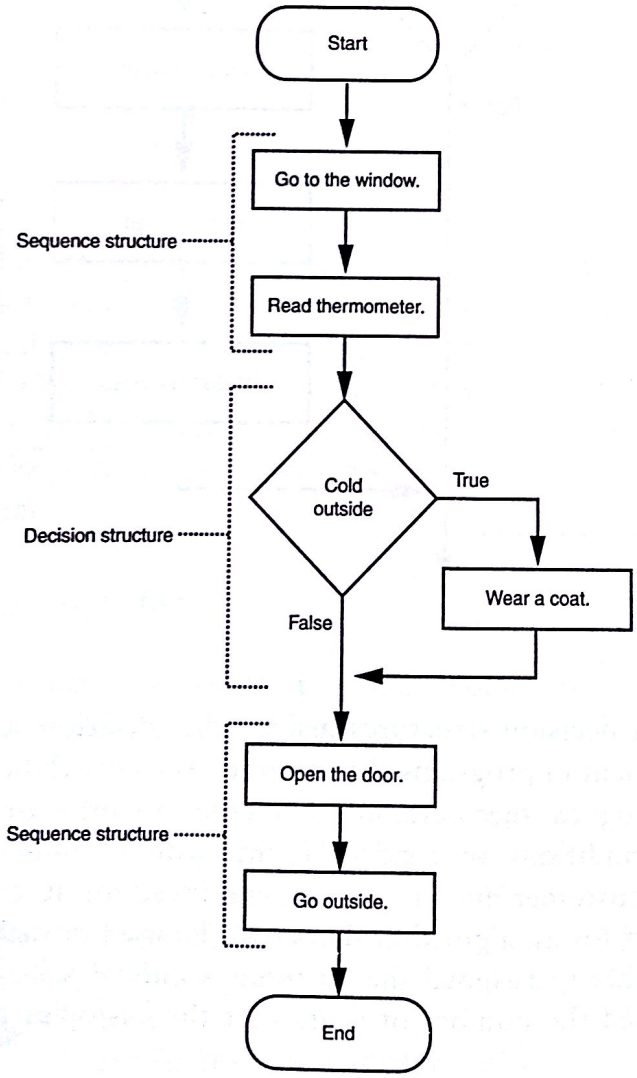
```
if s1 > s2:
    print(s2)
    print(s1)
else:
    print(s1)
    print(s2)
```

3.4 Nested Decision Structures and the if-elif-else Statement

CONCEPT: To test more than one condition, a decision structure can be nested inside another decision structure.

In Section 3.1, we mentioned that a control structure determines the order in which a set of statements execute. Programs are usually designed as combinations of different control structures. For example, Figure 3-10 shows a flowchart that combines a decision structure with two sequence structures.

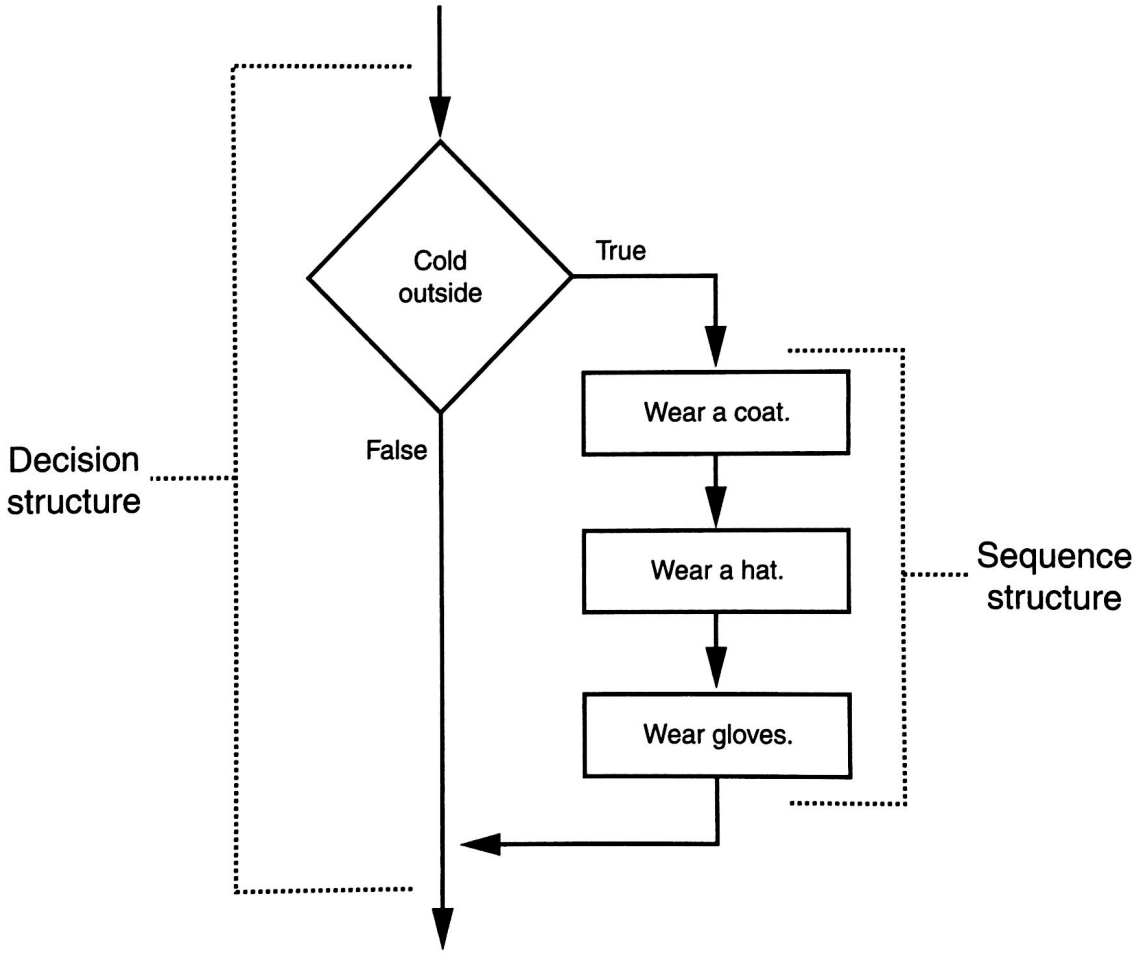
Figure 3-10 Combining sequence structures with a decision structure



The flowchart in the figure starts with a sequence structure. Assuming you have an outdoor thermometer in your window, the first step is `Go to the window`, and the next step is `Read thermometer`. A decision structure appears next, testing the condition `Cold outside`. If this is true, the action `Wear a coat` is performed. Another sequence structure appears next. The step `Open the door` is performed, followed by `Go outside`.

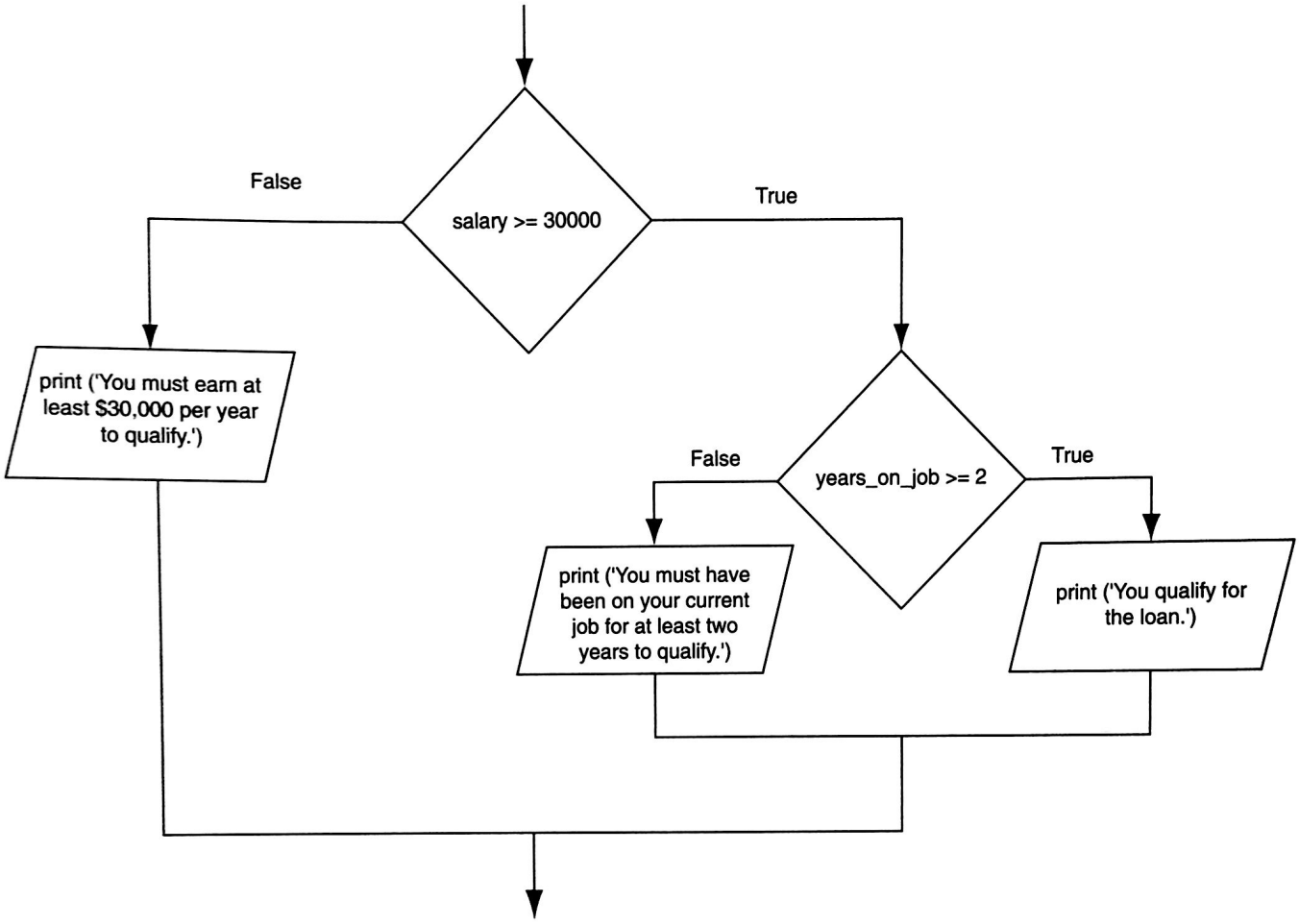
Quite often, structures must be nested inside other structures. For example, look at the partial flowchart in Figure 3-11. It shows a decision structure with a sequence structure nested inside it. The decision structure tests the condition `Cold outside`. If that condition is true, the steps in the sequence structure are executed.

Figure 3-11 A sequence structure nested inside a decision structure



You can also nest decision structures inside other decision structures. In fact, this is a common requirement in programs that need to test more than one condition. For example, consider a program that determines whether a bank customer qualifies for a loan. To qualify, two conditions must exist: (1) the customer must earn at least $30,000 per year, and (2) the customer must have been employed for at least two years. Figure 3-12 shows a flowchart for an algorithm that could be used in such a program. Assume that the `salary` variable is assigned the customer's annual salary, and the `years_on_job` variable is assigned the number of years that the customer has worked on his or her current job.

Figure 3-12 A nested decision structure



If we follow the flow of execution, we see that the condition `salary >= 30000` is tested. If this condition is false, there is no need to perform further tests; we know that the customer does not qualify for the loan. If the condition is true, however, we need to test the second condition. This is done with a nested decision structure that tests the condition `years_on_job >= 2`. If this condition is true, then the customer qualifies for the loan. If this condition is false, then the customer does not qualify. Program 3-5 shows the code for the complete program.

Program 3-5 (loan_qualifier.py)

```
1  # This program determines whether a bank customer
2  # qualifies for a loan.
3
4  min_salary = 30000.0 # The minimum annual salary
5  min_years = 2        # The minimum years on the job
6
7  # Get the customer's annual salary.
8  salary = float(input('Enter your annual salary:'))
9
10 # Get the number of years on the current job.
```

(program continues)

Program 3-5 *(continued)*

```
11  years_on_job = int(input('Enter the number of' +
12                           'years employed:'))
13
14  # Determine whether the customer qualifies.
15  if salary >= min_salary:
16      if years_on_job >= min_years:
17          print('You qualify for the loan.')
18      else:
19          print('You must have been employed', \
20                'for at least', min_years, \
21                'years to qualify.')
22  else:
23      print('You must earn at least $', \
24            format(min_salary, ',.2f'), \
25            ' per year to qualify.', sep='')
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 Enter
Enter the number of years employed: 1 Enter
You must have been employed for at least 2 years to qualify.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 25000 Enter
Enter the number of years employed: 5 Enter
You must earn at least $30,000.00 per year to qualify.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 Enter
Enter the number of years employed: 5 Enter
You qualify for the loan.
```

Look at the `if-else` statement that begins in line 15. It tests the condition `salary >= min_salary`. If this condition is true, the `if-else` statement that begins in line 16 is executed. Otherwise the program jumps to the `else` clause in line 22 and executes the statement in lines 23 through 25.

It's important to use proper indentation in a nested decision structure. Not only is proper indentation required by the Python interpreter, but it also makes it easier for you, the human reader of your code, to see which actions are performed by each part of the structure. Follow these rules when writing nested `if` statements:

- Make sure each `else` clause is aligned with its matching `if` clause. This is shown in Figure 3-13.
- Make sure the statements in each block are consistently indented. The shaded parts of Figure 3-14 show the nested blocks in the decision structure. Notice that each statement in each block is indented the same amount.

Figure 3-13 Alignment of `if` and `else` clauses

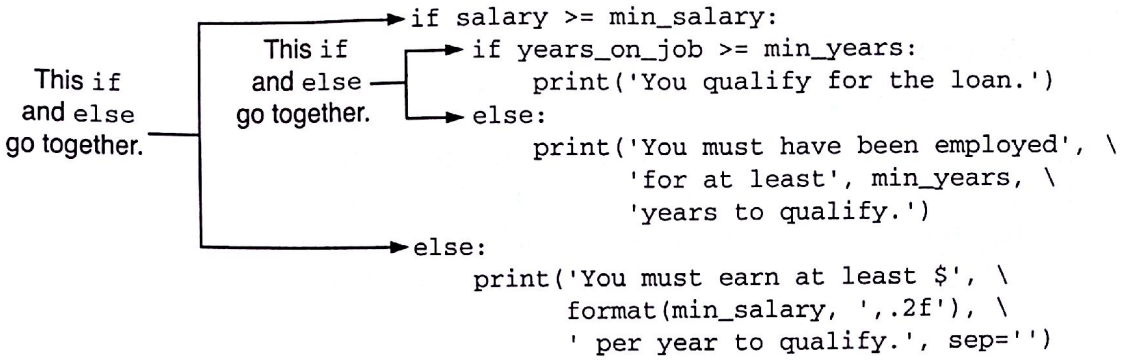


Figure 3-14 Nested blocks

```
if salary >= min_salary:
    if years_on_job >= min_years:
        print('You qualify for the loan.')
    else:
        print('You must have been employed', \
              'for at least', min_years, \
              'years to qualify.')
else:
    print('You must earn at least $', \
          format(min_salary, ',.2f'), \
          ' per year to qualify.', sep='')
```

Testing a Series of Conditions

In the previous example you saw how a program can use nested decision structures to test more than one condition. It is not uncommon for a program to have a series of conditions to test and then perform an action depending on which condition is true. One way to accomplish this is to have a decision structure with numerous other decision structures nested inside it. For example, consider the program presented in the following *In the Spotlight* section.

In the Spotlight:

Multiple Nested Decision Structures

Dr. Suarez teaches a literature class and uses the following 10-point grading scale for all of his exams:

Test Score	Grade
90 and above	A
80–89	B
70–79	C
60–69	D
Below 60	F

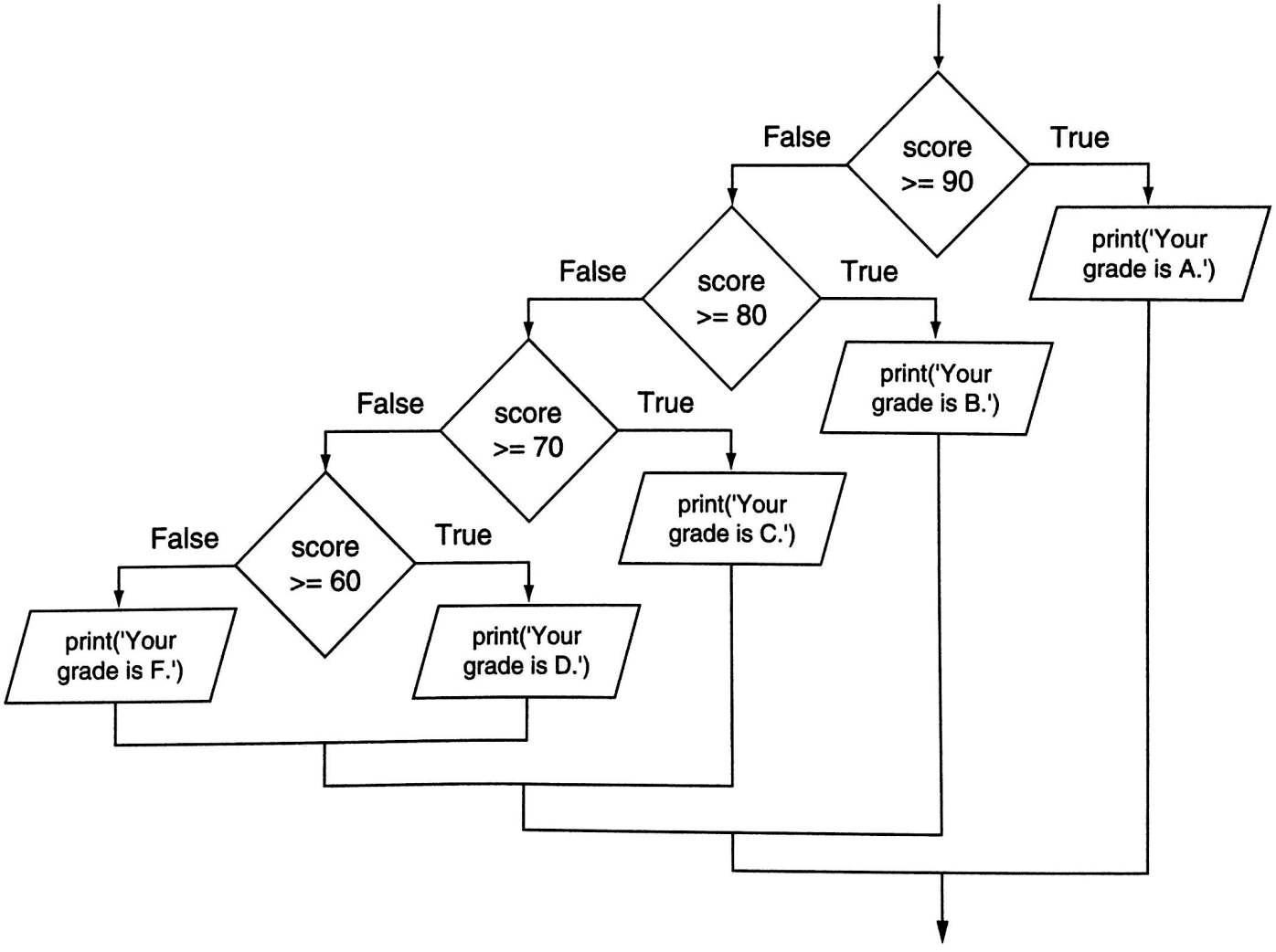
He has asked you to write a program that will allow a student to enter a test score and then display the grade for that score. Here is the algorithm that you will use:

1. Ask the user to enter a test score.
2. Determine the grade in the following manner:

If the score is greater than or equal to 90, then the grade is A.
Else, if the score is greater than or equal to 80, then the grade is B.
Else, if the score is greater than or equal to 70, then the grade is C.
Else, if the score is greater than or equal to 60, then the grade is D.
Else, the grade is F.

You decide that the process of determining the grade will require several nested decision structures, as shown in Figure 3-15. Program 3-6 shows the code for the program. The code for the nested decision structures is in lines 14 through 26.

Figure 3-15 Nested decision structure to determine a grade



Program 3-6 (grader.py)

```
1  # This program gets a numeric test score from the
2  # user and displays the corresponding letter grade.
3
```

```
4  # Variables to represent the grade thresholds
5  A_score = 90
6  B_score = 80
7  C_score = 70
8  D_score = 60
9
10 # Get a test score from the user.
11 score = int(input('Enter your test score:'))
12
13 # Determine the grade.
14 if score >= A_score:
15     print('Your grade is A.')
16 else:
17     if score >= B_score:
18         print('Your grade is B.')
19     else:
20         if score >= C_score:
21             print('Your grade is C.')
22         else:
23             if score >= D_score:
24                 print('Your grade is D.')
25             else:
26                 print('Your grade is F.')
```

Program Output (with input shown in bold)

```
Enter your test score: 78 Enter
Your grade is C.
```

Program Output (with input shown in bold)

```
Enter your test score: 84 Enter
Your grade is B.
```

The `if-elif-else` Statement

Even though Program 3-6 is a simple example, the logic of the nested decision structure is fairly complex. Python provides a special version of the decision structure known as the `if-elif-else` statement, which makes this type of logic simpler to write. Here is the general format of the `if-elif-else` statement:

```
if condition_1:
    statement
    statement
    etc.
elif condition_2:
    statement
    statement
    etc.
```

Insert as many `elif` *clauses as necessary . . .*

```
else:
    statement
    statement
    etc.
```

When the statement executes, *condition_1* is tested. If *condition_1* is true, the block of statements that immediately follow is executed, up to the `elif` clause. The rest of the structure is ignored. If *condition_1* is false, however, the program jumps to the very next `elif` clause and tests *condition_2*. If it is true, the block of statements that immediately follow is executed, up to the next `elif` clause. The rest of the structure is then ignored. This process continues until a condition is found to be true, or no more `elif` clauses are left. If no condition is true, the block of statements following the `else` clause is executed.

The following is an example of the `if-elif-else` statement. This code works the same as the nested decision structure in lines 14 through 26 of Program 3-6.

```
if score >= A_score:
    print('Your grade is A.')
elif score >= B_score:
    print('Your grade is B.')
elif score >= C_score:
    print('Your grade is C.')
elif score >= D_score:
    print('Your grade is D.')
else:
    print('Your grade is F.')
```

Notice the alignment and indentation that is used with the `if-elif-else` statement: The `if`, `elif`, and `else` clauses are all aligned, and the conditionally executed blocks are indented.

The `if-elif-else` statement is never required because its logic can be coded with nested `if-else` statements. However, a long series of nested `if-else` statements has two particular disadvantages when you are debugging code:

- The code can grow complex and become difficult to understand.
- Because of the required indentation, a long series of nested `if-else` statements can become too long to be displayed on the computer screen without horizontal scrolling. Also, long statements tend to "wrap around" when printed on paper, making the code even more difficult to read.

The logic of an `if-elif-else` statement is usually easier to follow than a long series of nested `if-else` statements. And, because all of the clauses are aligned in an `if-elif-else` statement, the lengths of the lines in the statement tend to be shorter.

Checkpoint

3.13 Convert the following code to an `if-elif-else` statement:

```
if number == 1:
    print('One')
```

```
else:
    if number == 2:
        print('Two')
    else:
        if number == 3:
            print('Three')
        else:
            print('Unknown')
```

3.5 Logical Operators

CONCEPT: The logical **and** operator and the logical **or** operator allow you to connect multiple Boolean expressions to create a compound expression. The logical **not** operator reverses the truth of a Boolean expression.

Python provides a set of operators known as *logical operators*, which you can use to create complex Boolean expressions. Table 3-3 describes these operators.

Table 3-3 Logical operators

Operator	Meaning
`and`	The `and` operator connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.
`or`	The `or` operator connects two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which.
`not`	The `not` operator is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The `not` operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

Table 3-4 shows examples of several compound Boolean expressions that use logical operators.

Table 3-4 Compound Boolean expressions using logical operators

Expression	Meaning
`x > y and a < b`	Is `x` greater than `y` AND is `a` less than `b`?
`x == y or x == z`	Is `x` equal to `y` OR is `x` equal to `z`?
`not (x > y)`	Is the expression `x > y` NOT true?

The and Operator

The and operator takes two Boolean expressions as operands and creates a compound Boolean expression that is true only when both subexpressions are true. The following is an example of an if statement that uses the and operator:

```
if temperature < 20 and minutes > 12:
    print('The temperature is in the danger zone.')
```

In this statement, the two Boolean expressions `temperature < 20` and `minutes > 12` are combined into a compound expression. The `print` function will be called only if `temperature` is less than 20 and `minutes` is greater than 12. If either of the Boolean subexpressions is false, the compound expression is false and the message is not displayed.

Table 3-5 shows a truth table for the and operator. The truth table lists expressions showing all the possible combinations of true and false connected with the and operator. The resulting values of the expressions are also shown.

Table 3-5 Truth table for the and operator

Expression	Value of the Expression
true and false	false
false and true	false
false and false	false
true and true	true

As the table shows, both sides of the and operator must be true for the operator to return a true value.

The or Operator

The or operator takes two Boolean expressions as operands and creates a compound Boolean expression that is true when either of the subexpressions is true. The following is an example of an if statement that uses the or operator:

```
if temperature < 20 or temperature > 100:
    print('The temperature is too extreme')
```

The `print` function will be called only if `temperature` is less than 20 or `temperature` is greater than 100. If either subexpression is true, the compound expression is true. Table 3-6 shows a truth table for the or operator.

Table 3-6 Truth table for the or operator

Expression	Value of the Expression
true or false	true
false or true	true
false or false	false
true or true	true

All it takes for an or expression to be true is for one side of the or operator to be true. It doesn't matter if the other side is false or true.

Short-Circuit Evaluation

Both the and and or operators perform *short-circuit evaluation*. Here's how it works with the and operator: If the expression on the left side of the and operator is false, the expression on the right side will not be checked. Because the compound expression will be false if only one of the subexpressions is false, it would waste CPU time to check the remaining expression. So, when the and operator finds that the expression on its left is false, it short-circuits and does not evaluate the expression on its right.

Here's how short-circuit evaluation works with the or operator: If the expression on the left side of the or operator is true, the expression on the right side will not be checked. Because it is only necessary for one of the expressions to be true, it would waste CPU time to check the remaining expression.

The not Operator

The not operator is a unary operator that takes a Boolean expression as its operand and reverses its logical value. In other words, if the expression is true, the not operator returns false, and if the expression is false, the not operator returns true. The following is an if statement using the not operator:

```
if not(temperature > 100):
    print('This is below the maximum temperature.')
```

First, the expression (temperature > 100) is tested and a value of either true or false is the result. Then the not operator is applied to that value. If the expression (temperature > 100) is true, the not operator returns false. If the expression (temperature > 100) is false, the not operator returns true. The previous code is equivalent to asking: "Is the temperature not greater than 100?"

NOTE: In this example, we have put parentheses around the expression temperature > 100. This is to make it clear that we are applying the not operator to the value of the expression temperature > 100, not just to the temperature variable.

Table 3-7 shows a truth table for the not operator.

Table 3-7 Truth table for the not operator

Expression	Value of the Expression
not true	false
not false	true

The Loan Qualifier Program Revisited

In some situations the and operator can be used to simplify nested decision structures. For example, recall that the loan qualifier program in Program 3-5 uses the following nested `if-else` statements:

```
if salary >= min_salary:
    if years_on_job >= min_years:
        print('You qualify for the loan.')
    else:
        print('You must have been employed', \
              'for at least', min_years, \
              'years to qualify.')
else:
    print('You must earn at least $', \
          format(min_salary, ',.2f'), \
          ' per year to qualify.', sep='')
```

The purpose of this decision structure is to determine that a person's salary is at least $30,000 and that he or she has been at their current job for at least two years. Program 3-7 shows a way to perform a similar task with simpler code.

Program 3-7 (loan_qualifier2.py)

```
1  # This program determines whether a bank customer
2  # qualifies for a loan.
3
4  min_salary = 30000.0 # The minimum annual salary
5  min_years = 2        # The minimum years on the job
6
7  # Get the customer's annual salary.
8  salary = float(input('Enter your annual salary:'))
9
10 # Get the number of years on the current job.
11 years_on_job = int(input('Enter the number of' +
12                           'years employed:'))
13
14 # Determine whether the customer qualifies.
15 if salary >= min_salary and years_on_job >= min_years:
16     print('You qualify for the loan.')
17 else:
18     print('You do not qualify for this loan.')
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 Enter
Enter the number of years employed: 1 Enter
You do not qualify for this loan.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 25000 Enter
Enter the number of years employed: 5 Enter
You do not qualify for this loan.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 Enter
Enter the number of years employed: 5 Enter
You qualify for the loan.
```

The `if-else` statement in lines 15 through 18 tests the compound expression `salary >= min_salary and years_on_job >= min_years`. If both subexpressions are true, the compound expression is true and the message "You qualify for the loan" is displayed. If either of the subexpressions is false, the compound expression is false and the message "You do not qualify for this loan" is displayed.

NOTE: A careful observer will realize that Program 3-7 is similar to Program 3-5, but it is not equivalent. If the user does not qualify for the loan, Program 3-7 displays only the message "You do not qualify for this loan" whereas Program 3-5 displays one of two possible messages explaining why the user did not qualify.

Yet Another Loan Qualifier Program

Suppose the bank is losing customers to a competing bank that isn't as strict about whom it loans money to. In response, the bank decides to change its loan requirements. Now, customers have to meet only one of the previous conditions, not both. Program 3-8 shows the code for the new loan qualifier program. The compound expression that is tested by the `if-else` statement in line 15 now uses the `or` operator.

Program 3-8 (loan_qualifier3.py)

```
1  # This program determines whether a bank customer
2  # qualifies for a loan.
3
4  min_salary = 30000.0 # The minimum annual salary
5  min_years = 2        # The minimum years on the job
6
7  # Get the customer's annual salary.
8  salary = float(input('Enter your annual salary:'))
9
10 # Get the number of years on the current job.
11 years_on_job = int(input('Enter the number of' +
12                           'years employed:'))
13
```

(program continues)

Program 3-8 *(continued)*

```
14  # Determine whether the customer qualifies.
15  if salary >= min_salary or years_on_job >= min_years:
16      print('You qualify for the loan.')
17  else:
18      print('You do not qualify for this loan.')
```

Program Output (with input shown in bold)

```
Enter your annual salary: 35000 Enter
Enter the number of years employed: 1 Enter
You qualify for the loan.
```

Program Output (with input shown in bold)

```
Enter your annual salary: 25000 Enter
Enter the number of years employed: 5 Enter
You qualify for the loan.
```

Program Output (with input shown in bold)

```
Enter your annual salary 12000 Enter
Enter the number of years employed: 1 Enter
You do not qualify for this loan.
```

Checking Numeric Ranges with Logical Operators

Sometimes you will need to design an algorithm that determines whether a numeric value is within a specific range of values or outside a specific range of values. When determining whether a number is inside a range, it is best to use the and operator. For example, the following `if` statement checks the value in `x` to determine whether it is in the range of 20 through 40:

```
if x >= 20 and x <= 40:
    print('The value is in the acceptable range.')
```

The compound Boolean expression being tested by this statement will be true only when `x` is greater than or equal to 20 and less than or equal to 40. The value in `x` must be within the range of 20 through 40 for this compound expression to be true.

When determining whether a number is outside a range, it is best to use the `or` operator. The following statement determines whether `x` is outside the range of 20 through 40:

```
if x < 20 or x > 40:
    print('The value is outside the acceptable range.')
```

It is important not to get the logic of the logical operators confused when testing for a range of numbers. For example, the compound Boolean expression in the following code would never test true:

```
# This is an error!
if x < 20 and x > 40:
    print('The value is outside the acceptable range.')
```

Obviously, `x` cannot be less than 20 and at the same time be greater than 40.

Checkpoint

3.14 What is a compound Boolean expression?

3.15 The following truth table shows various combinations of the values true and false connected by a logical operator. Complete the table by circling T or F to indicate whether the result of such a combination is true or false.

Logical Expression	Result (circle T or F)	
True and False	T	F
True and True	T	F
False and True	T	F
False and False	T	F
True or False	T	F
True or True	T	F
False or True	T	F
False or False	T	F
not True	T	F
not False	T	F

3.16 Assume the variables `a = 2`, `b = 4`, and `c = 6`. Circle the T or F for each of the following conditions to indicate whether its value is true or false.

```
a == 4 or b > 2          T  F
6 <= c and a > 3         T  F
1 != b and c != 3        T  F
a >= -1 or a <= b        T  F
not (a > 2)              T  F
```

3.17 Explain how short-circuit evaluation works with the `and` and `or` operators.

3.18 Write an `if` statement that displays the message "The number is valid" if the value referenced by `speed` is within the range 0 through 200.

3.19 Write an `if` statement that displays the message "The number is not valid" if the value referenced by `speed` is outside the range 0 through 200.

3.6 Boolean Variables

CONCEPT: **A Boolean variable can reference one of two values: `True` or `False`. Boolean variables are commonly used as flags, which indicate whether specific conditions exist.**

So far in this book we have worked with `int`, `float`, and `str` (string) variables. In addition to these data types, Python also provides a `bool` data type. The `bool` data type allows you to create variables that may reference one of two possible values: `True` or `False`. Here are examples of how we assign values to a `bool` variable:

```
hungry = True
sleepy = False
```

Boolean variables are most commonly used as flags. A *flag* is a variable that signals when some condition exists in the program. When the flag variable is set to `False`, it indicates the condition does not exist. When the flag variable is set to `True`, it means the condition does exist.

For example, suppose a salesperson has a quota of $50,000. Assuming `sales` references the amount that the salesperson has sold, the following code determines whether the quota has been met:

```
if sales >= 50000.0:
    sales_quota_met = True
else:
    sales_quota_met = False
```

As a result of this code, the `sales_quota_met` variable can be used as a flag to indicate whether the sales quota has been met. Later in the program we might test the flag in the following way:

```
if sales_quota_met:
    print('You have met your sales quota!')
```

This code displays `'You have met your sales quota!'` if the `bool` variable `sales_quota_met` is `True`. Notice that we did not have to use the == operator to explicitly compare the `sales_quota_met` variable with the value `True`. This code is equivalent to the following:

```
if sales_quota_met == True:
    print('You have met your sales quota!')
```

Checkpoint

3.20 What values can you assign to a `bool` variable?

3.21 What is a flag variable?

Review Questions

Multiple Choice

1. A _____ structure can execute a set of statements only under certain circumstances.
   a. sequence
   b. circumstantial
   c. decision
   d. Boolean

2. A _____ structure provides one alternative path of execution.
   a. sequence
   b. single alternative decision
   c. one path alternative
   d. single execution decision

3. A(n) _____ expression has a value of either true or false.
   a. binary
   b. decision
   c. unconditional
   d. Boolean

4. The symbols >, <, and == are all _____ operators.
   a. relational
   b. logical
   c. conditional
   d. ternary

5. A(n) _____ structure tests a condition and then takes one path if the condition is true, or another path if the condition is false.
   a. `if` statement
   b. single alternative decision
   c. dual alternative decision
   d. sequence

6. You use a(n) _____ statement to write a single alternative decision structure.
   a. `test-jump`
   b. `if`
   c. `if-else`
   d. `if-call`

7. You use a(n) _____ statement to write a dual alternative decision structure.
   a. `test-jump`
   b. `if`
   c. `if-else`
   d. `if-call`

8. `and`, `or`, and `not` are _____ operators.
   a. relational
   b. logical
   c. conditional
   d. ternary

9. A compound Boolean expression created with the _____ operator is true only if both of its subexpressions are true.
   a. `and`
   b. `or`
   c. `not`
   d. `both`

10. A compound Boolean expression created with the _____ operator is true if either of its subexpressions is true.
    a. `and`
    b. `or`
    c. `not`
    d. `either`

11. The _____ operator takes a Boolean expression as its operand and reverses its logical value.
    a. `and`
    b. `or`
    c. `not`
    d. `either`
12. A _____ is a Boolean variable that signals when some condition exists in the program.
    a. flag
    b. signal
    c. sentinel
    d. siren

True or False

1. You can write any program using only sequence structures.
2. A program can be made of only one type of control structure. You cannot combine structures.
3. A single alternative decision structure tests a condition and then takes one path if the condition is true, or another path if the condition is false.
4. A decision structure can be nested inside another decision structure.
5. A compound Boolean expression created with the `and` operator is true only when both subexpressions are true.

Short Answer

1. Explain what is meant by the term "conditionally executed."
2. You need to test a condition and then execute one set of statements if the condition is true. If the condition is false, you need to execute a different set of statements. What structure will you use?
3. Briefly describe how the `and` operator works.
4. Briefly describe how the `or` operator works.
5. When determining whether a number is inside a range, which logical operator is it best to use?
6. What is a flag and how does it work?

Algorithm Workbench

1. Write an `if` statement that assigns 20 to the variable `y` and assigns 40 to the variable `z` if the variable `x` is greater than 100.
2. Write an `if` statement that assigns 0 to the variable `b` and assigns 1 to the variable `c` if the variable `a` is less than 10.
3. Write an `if-else` statement that assigns 0 to the variable `b` if the variable `a` is less than 10. Otherwise, it should assign 99 to the variable `b`.

4. The following code contains several nested `if-else` statements. Unfortunately, it was written without proper alignment and indentation. Rewrite the code and use the proper conventions of alignment and indentation.

```
if score >= A_score:
print('Your grade is A.')
else:
if score >= B_score:
print('Your grade is B.')
else:
if score >= C_score:
print('Your grade is C.')
else:
if score >= D_score:
print('Your grade is D.')
else:
print('Your grade is F.')
```

5. Write nested decision structures that perform the following: If `amount1` is greater than 10 and `amount2` is less than 100, display the greater of `amount1` and `amount2`.
6. Write an `if-else` statement that displays `'Speed is normal'` if the `speed variable` is within the range of 24 to 56. If the `speed` variable's value is outside this range, display `'Speed is abnormal'`.
7. Write an `if-else` statement that determines whether the `points` variable is outside the range of 9 to 51. If the variable's value is outside this range it should display "Invalid points." Otherwise, it should display "Valid points."

Programming Exercises

1. Day of the Week

Write a program that asks the user for a number in the range of 1 through 7. The program should display the corresponding day of the week, where 1 = Monday, 2 = Tuesday, 3 = Wednesday, 4 = Thursday, 5 = Friday, 6 = Saturday, and 7 = Sunday. The program should display an error message if the user enters a number that is outside the range of 1 through 7.

2. Areas of Rectangles

eoNote
e Areas of
tangles Problem

The area of a rectangle is the rectangle's length times its width. Write a program that asks for the length and width of two rectangles. The program should tell the user which rectangle has the greater area, or if the areas are the same.

3. Age Classifier

Write a program that asks the user to enter a person's age. The program should display a message indicating whether the person is an infant, a child, a teenager, or an adult. Following are the guidelines:

- If the person is 1 year old or less, he or she is an infant.
- If the person is older than 1 year, but younger than 13 years, he or she is a child.
- If the person is at least 13 years old, but less than 20 years old, he or she is a teenager.
- If the person is at least 20 years old, he or she is an adult.

4. Roman Numerals

Write a program that prompts the user to enter a number within the range of 1 through 10. The program should display the Roman numeral version of that number. If the number is outside the range of 1 through 10, the program should display an error message. The following table shows the Roman numerals for the numbers 1 through 10:

Number	Roman Numeral
1	I
2	II
3	III
4	IV
5	V
6	VI
7	VII
8	VIII
9	IX
10	X

5. Mass and Weight

Scientists measure an object's mass in kilograms and its weight in newtons. If you know the amount of mass of an object in kilograms, you can calculate its weight in newtons with the following formula:

$$weight = mass \times 9.8$$

Write a program that asks the user to enter an object's mass, and then calculates its weight. If the object weighs more than 500 newtons, display a message indicating that it is too heavy. If the object weighs less than 100 newtons, display a message indicating that it is too light.

6. Magic Dates

The date June 10, 1960, is special because when it is written in the following format, the month times the day equals the year:

6/10/60

Design a program that asks the user to enter a month (in numeric form), a day, and a two-digit year. The program should then determine whether the month times the day equals the year. If so, it should display a message saying the date is magic. Otherwise, it should display a message saying the date is not magic.

7. Color Mixer

The colors red, blue, and yellow are known as the primary colors because they cannot be made by mixing other colors. When you mix two primary colors, you get a secondary color, as shown here:

When you mix red and blue, you get purple.
When you mix red and yellow, you get orange.
When you mix blue and yellow, you get green.

Design a program that prompts the user to enter the names of two primary colors to mix. If the user enters anything other than "red," "blue," or "yellow," the program should display an error message. Otherwise, the program should display the name of the secondary color that results.

8. Hot Dog Cookout Calculator

Assume that hot dogs come in packages of 10, and hot dog buns come in packages of 8. Write a program that calculates the number of packages of hot dogs and the number of packages of hot dog buns needed for a cookout, with the minimum amount of leftovers. The program should ask the user for the number of people attending the cookout and the number of hot dogs each person will be given. The program should display the following details:

- The minimum number of packages of hot dogs required
- The minimum number of packages of hot dog buns required
- The number of hot dogs that will be left over
- The number of hot dog buns that will be left over

9. Roulette Wheel Colors

On a roulette wheel, the pockets are numbered from 0 to 36. The colors of the pockets are as follows:

- Pocket 0 is green.
- For pockets 1 through 10, the odd-numbered pockets are red and the even-numbered pockets are black.
- For pockets 11 through 18, the odd-numbered pockets are black and the even-numbered pockets are red.
- For pockets 19 through 28, the odd-numbered pockets are red and the even-numbered pockets are black.
- For pockets 29 through 36, the odd-numbered pockets are black and the even-numbered pockets are red.

Write a program that asks the user to enter a pocket number and displays whether the pocket is green, red, or black. The program should display an error message if the user enters a number that is outside the range of 0 through 36.

10. Money Counting Game

Create a change-counting game that gets the user to enter the number of coins required to make exactly one dollar. The program should prompt the user to enter the number of pennies, nickels, dimes, and quarters. If the total value of the coins entered is equal to one dollar, the program should congratulate the user for winning the game. Otherwise, the program should display a message indicating whether the amount entered was more than or less than one dollar.

11. Book Club Points

Serendipity Booksellers has a book club that awards points to its customers based on the number of books purchased each month. The points are awarded as follows:

- If a customer purchases 0 books, he or she earns 0 points.
- If a customer purchases 2 books, he or she earns 5 points.

- If a customer purchases 4 books, he or she earns 15 points.
- If a customer purchases 6 books, he or she earns 30 points.
- If a customer purchases 8 or more books, he or she earns 60 points.

Write a program that asks the user to enter the number of books that he or she has purchased this month and displays the number of points awarded.

12. Software Sales

A software company sells a package that retails for $99. Quantity discounts are given according to the following table:

Quantity	Discount
10–19	10%
20–49	20%
50–99	30%
100 or more	40%

Write a program that asks the user to enter the number of packages purchased. The program should then display the amount of the discount (if any) and the total amount of the purchase after the discount.

13. Shipping Charges

The Fast Freight Shipping Company charges the following rates:

Weight of Package	Rate per Pound
2 pounds or less	$1.50
Over 2 pounds but not more than 6 pounds	$3.00
Over 6 pounds but not more than 10 pounds	$4.00
Over 10 pounds	$4.75

Write a program that asks the user to enter the weight of a package and then displays the shipping charges.

14. Body Mass Index

Write a program that calculates and displays a person's body mass index (BMI). The BMI is often used to determine whether a person is overweight or underweight for his or her height. A person's BMI is calculated with the following formula:

$$BMI = weight \times 703/height^2$$

where *weight* is measured in pounds and *height* is measured in inches. The program should ask the user to enter his or her weight and height and then display the user's BMI. The program should also display a message indicating whether the person has optimal weight, is underweight, or is overweight. A person's weight is considered to be optimal if his or her BMI is between 18.5 and 25. If the BMI is less than 18.5, the person is considered to be underweight. If the BMI value is greater than 25, the person is considered to be overweight.

15. Time Calculator

Write a program that asks the user to enter a number of seconds and works as follows:

- There are 60 seconds in a minute. If the number of seconds entered by the user is greater than or equal to 60, the program should display the number of minutes in that many seconds.
- There are 3,600 seconds in an hour. If the number of seconds entered by the user is greater than or equal to 3,600, the program should display the number of hours in that many seconds.
- There are 86,400 seconds in a day. If the number of seconds entered by the user is greater than or equal to 86,400, the program should display the number of days in that many seconds.