Chapter 4
Repetition Structure

Topics:

1. Introduction to Repetition Structures
2. The `while` Loop: a Condition-Controlled Loop
3. The `for` Loop: a Count-Controlled Loop
4. Calculating a Running Total
5. Sentinels
6. Input Validation Loops
7. Nested Loops

Addison-Wesley
is an imprint of
PEARSON

1. Introduction to Repetition Structures

Often have to write code that performs the same task multiple times. Disadvantages to duplicating code:

- Makes program large
- Time consuming
- May need to be corrected in many places

Repetition structure: causes a statement or set of statements to execute repeatedly. Includes:

- condition-controlled loops → like while statement
- count-controlled loops → like for statement

2. The `while` Loop: a Condition-Controlled Loop

`while` loop: is a condition-controlled loop causes a statement or set of statements to repeat as long as a condition is TRUE.

Python syntax:

```
while condition:
    statements
```

The `while` Loop: a Condition-Controlled Loop (cont'd.)

In flowchart, line goes back to previous part

Figure 4-1 The logic of a `while` loop

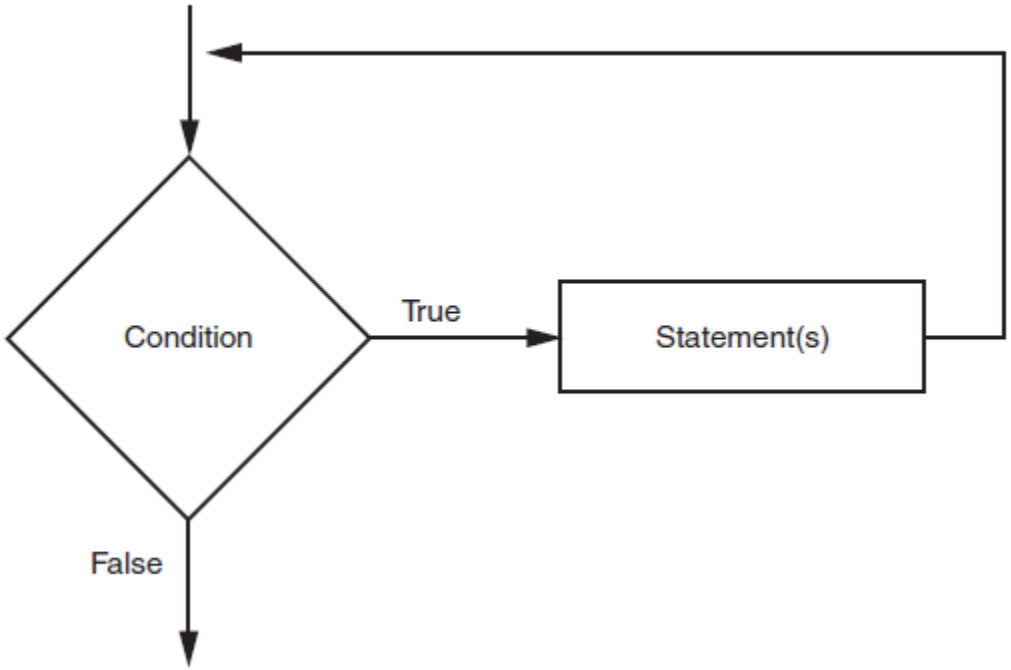


Figure 4-3 Flowchart for Program 4-1

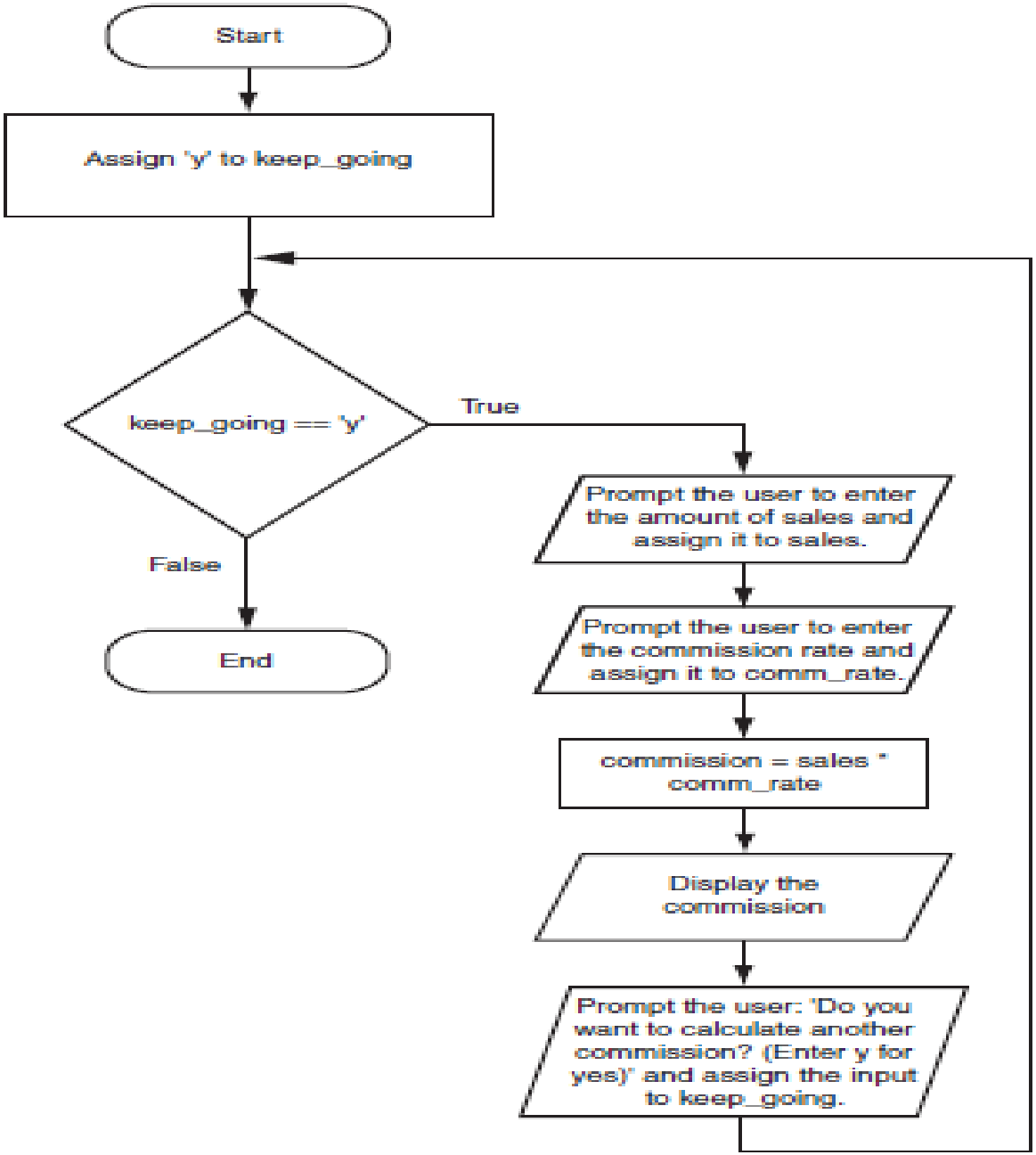
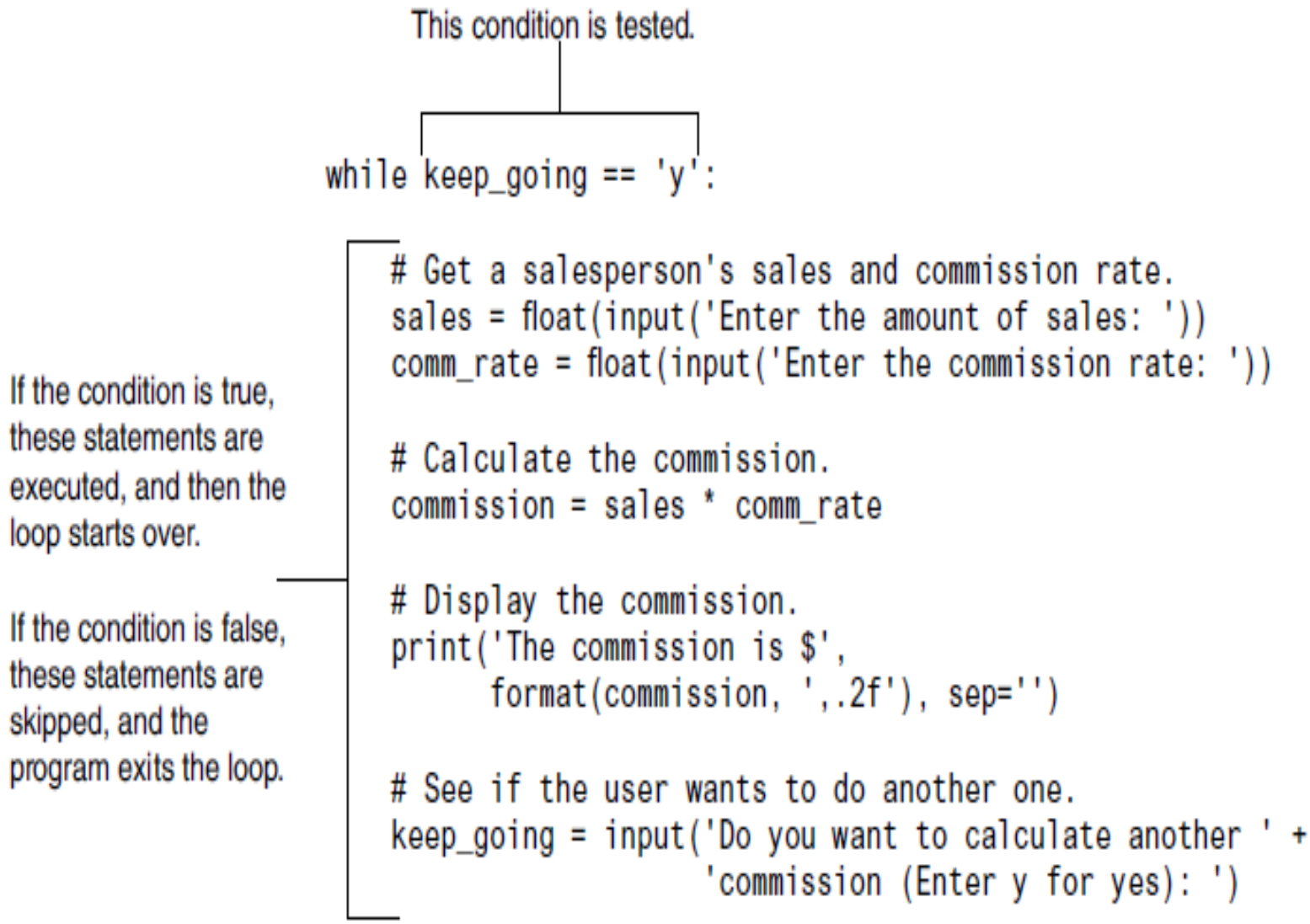


Figure 4-2 The while loop



Infinite Loops

Loops must contain within themselves a way to terminate. Something inside a `while` loop must make the condition FALSE.

Infinite loop: loop that does not have a way of stopping, repeats until program is interrupted, occurs when programmer forgets to include stopping code in the loop.

The only way to stop this program is to press Ctrl+C on the keyboard to interrupt it.

3. The `for` Loop: a Count-Controlled Loop

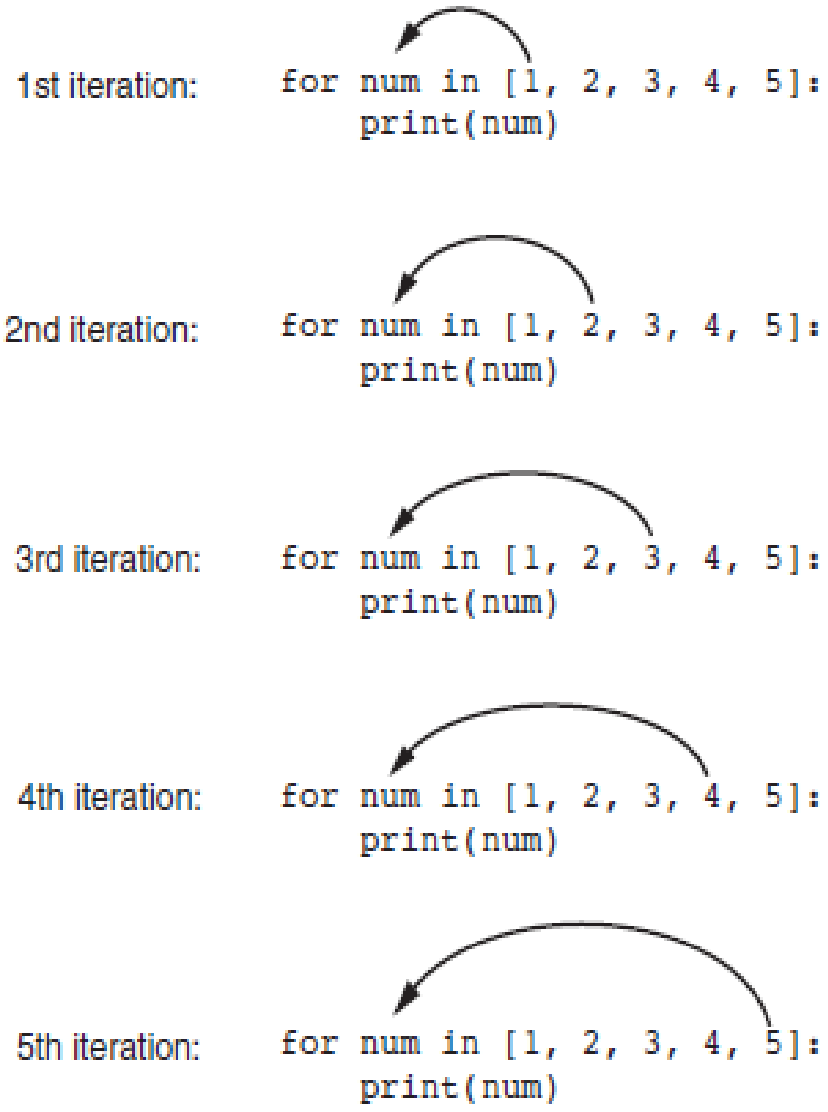
`for` loop: is a count-controlled loop which iterates a specific number of times, designed to work with sequence or list of data items (iterates once for each item in the sequence).

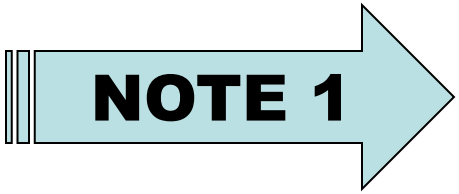
Python syntax:

```
for variable in [val1, val2, etc]:
    statements
```

The *`variable`* : is the name of a variable which is also known as *Target variable* because it is the target of an assignment at the beginning of each loop iteration.

Figure 4-4 The `for` loop

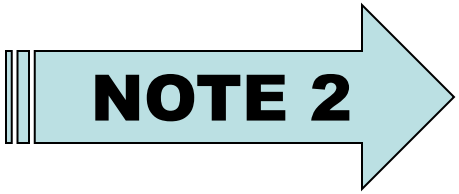




The value that appear in the list do not have to be a consecutively ordered series of numbers. For example:

```
for num in [1, 3, 5 , 7, 9]:
    print ( num )
```

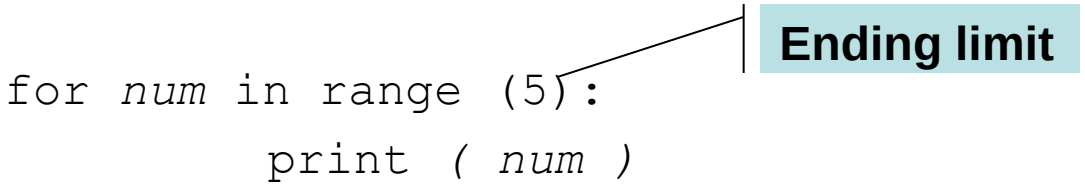
The value that appear in the list may contained a number of strings. For example:




```
for name in ['Ali', 'Ahmed', 'Omar']:
    print ( name )
```

Using the `range` Function with the `for` Loop

The `range` function simplifies the process of writing a count-controlled loop and returns an *iterable object* which is similar to a list which contains a sequence of values that can be iterated over. For example:

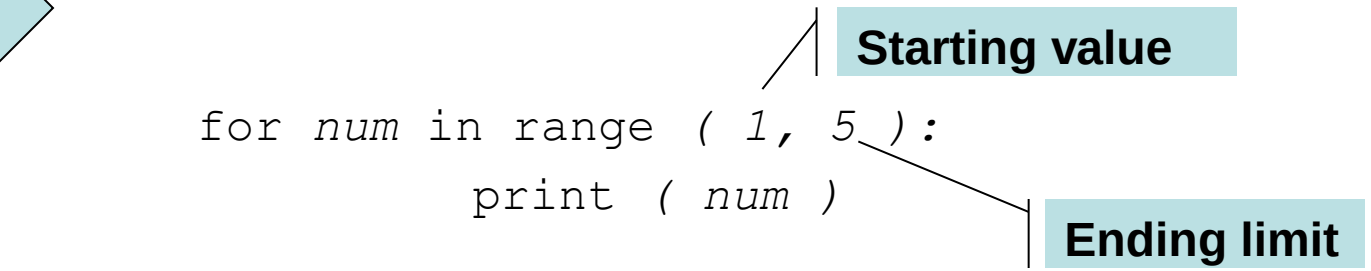


This code works the same as the following:

```
for num in [                    ]:
        print ( num )
```

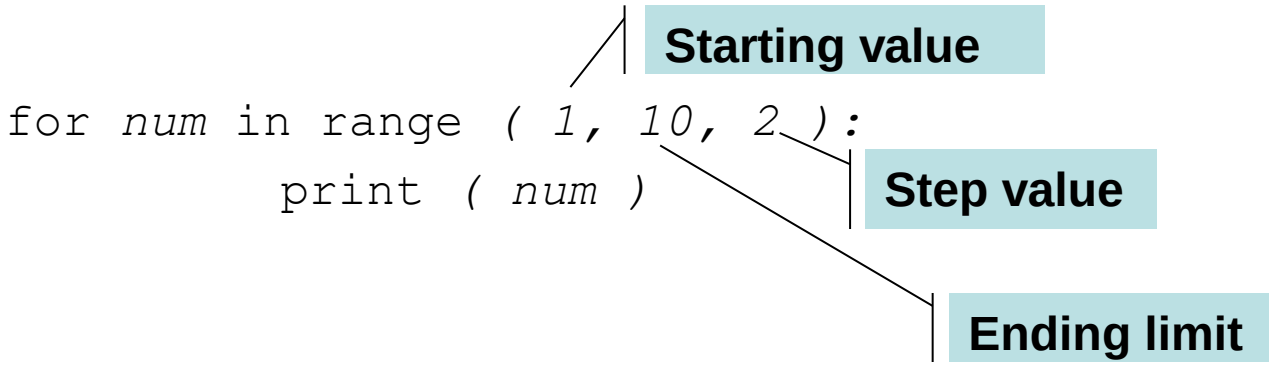
NOTE 3

If you pass two arguments to the `range` function, the first argument is used as the *starting value* of the sequence, and the second argument is used as the *ending limit*. For example:



NOTE 4

If you pass a third argument to the `range` function, that argument is used as *step value*. Instead of increasing by 1, each successive number in the sequence will increase by the step value. For example:



Using the Target Variable Inside the Loop

The purpose of target variable is to reference each item in a sequence as the loop iterates

Target variable can be used in calculations or tasks in the body of the loop. For example:

```
#calculate square root of each number in a range (1 through 10)

print ( 'number\tSquare')
for num in range (1 , 11):
        square = num**2
        print ( num , '\t' , square )
```

Letting the User Control the Loop Iterations

Sometimes, the programmer does not know exactly how many times the loop will execute. He needs to let the user control the number of times that a loop iterates. For example:

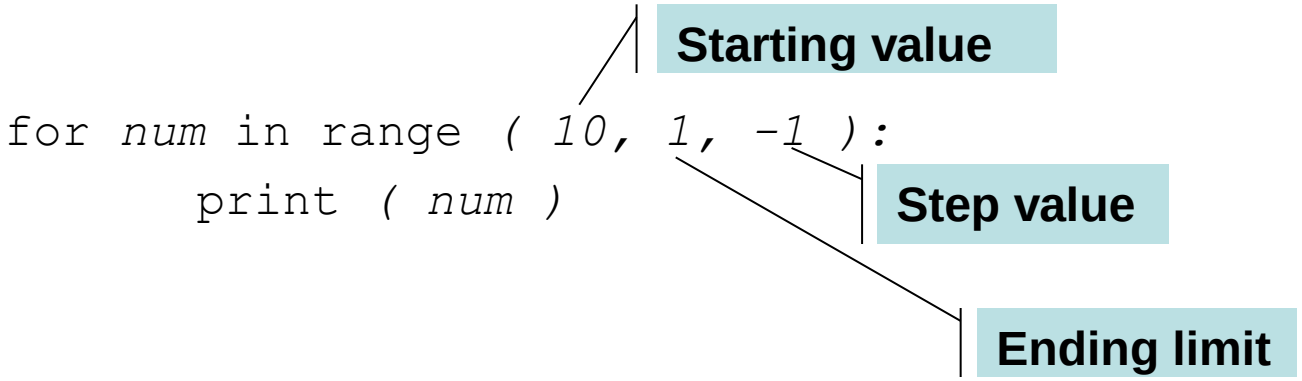
#This program uses a loop to display a table of numbers and their squares.

```
end = int (input ( 'how high should I got? '))
print ( 'number\tSquare')
for num in range (1 , end + 1 ):
        square = num**2
        print ( num , '\t' , square )
```

Ending limit

Generating an Iterable Sequence that Ranges from Highest to Lowest

The `range` function can be used to generate a sequence with numbers in descending order (from highest to lowest number). Make sure *starting number* is larger than *end limit*, and *step value* is negative. For example:



4. Calculating a Running Total

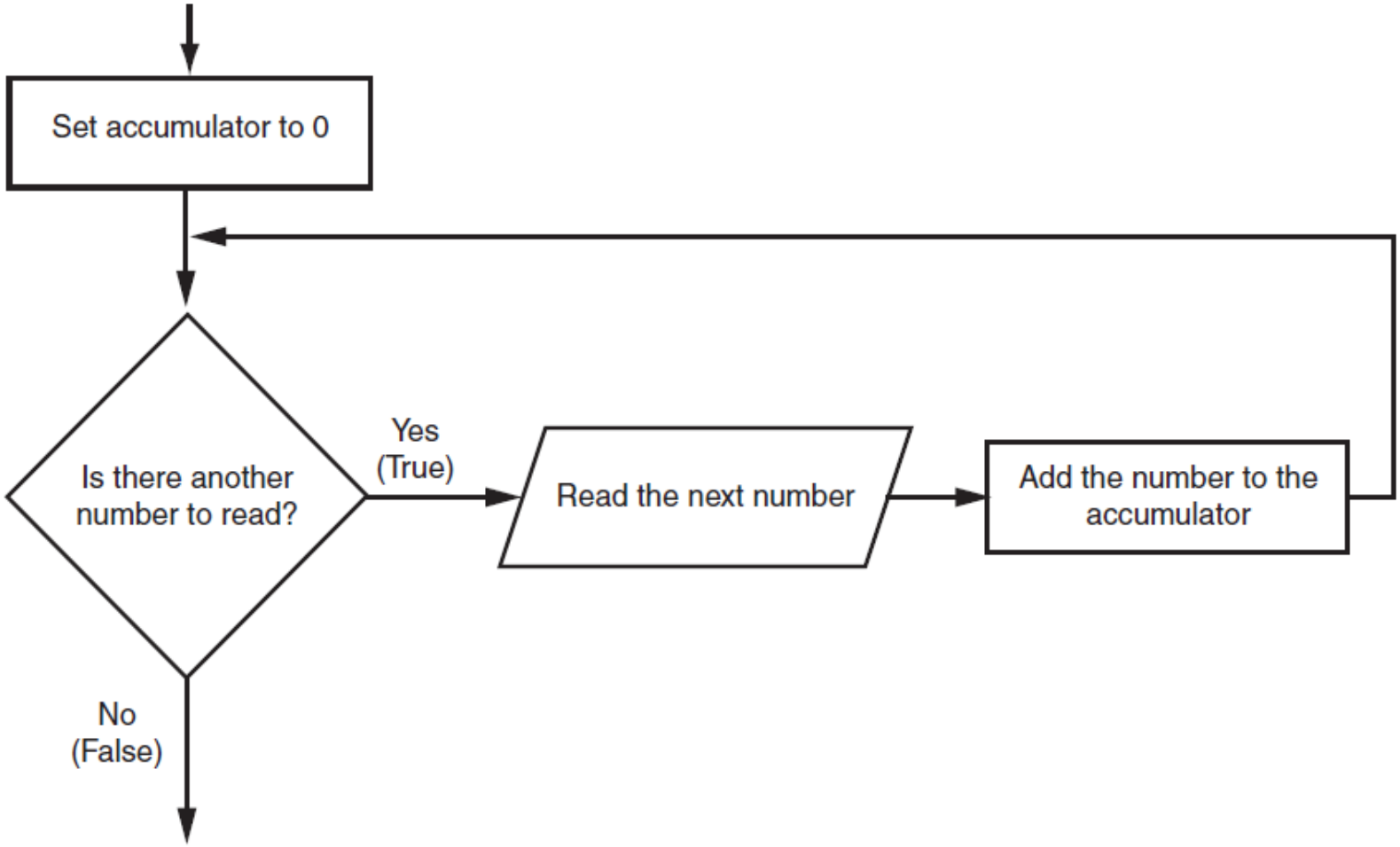
A Running total: is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an *accumulator.*

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

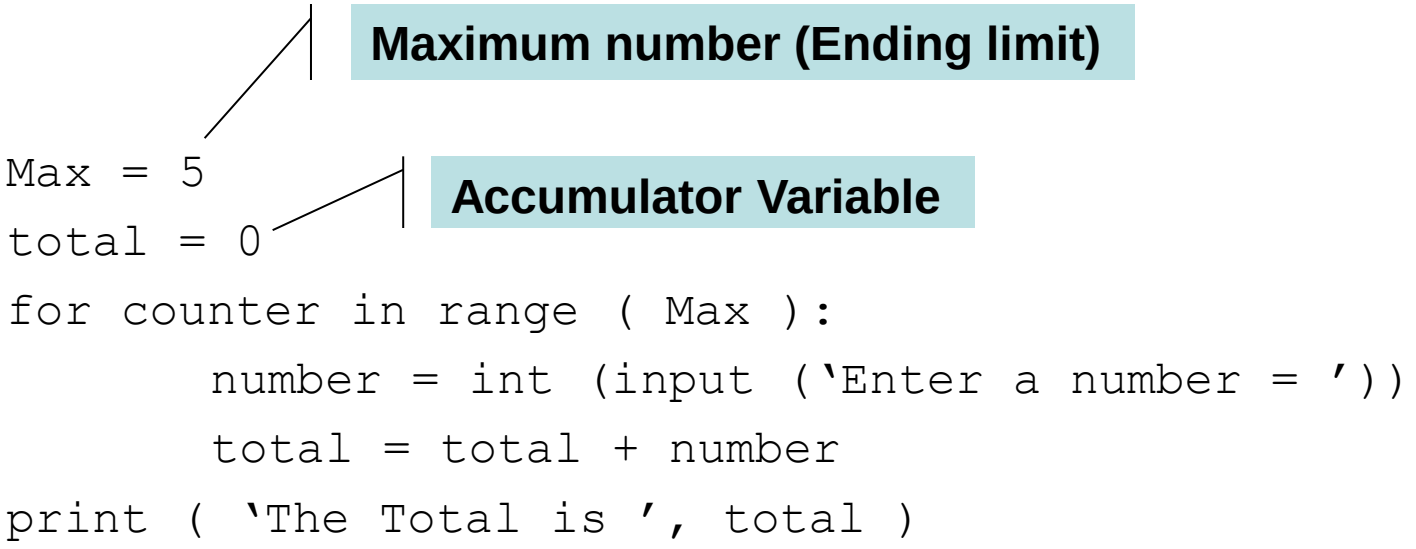
Calculating a Running Total (cont'd.)

Figure 4-6 Logic for calculating a running total



Calculating a Running Total (cont'd.)

#This program allows the user to enter five numbers, and displays the total of the numbers entered.




```
Max = 5
total = 0
for counter in range ( Max ):
        number = int (input ('Enter a number = '))
        total = total + number
print ( 'The Total is ', total )
```

The Augmented Assignment Operators

Table 4-2 Augmented assignment operators

Operator	Example Usage	Equivalent To
+=	x += 5	x = x + 5
-=	y -= 2	y = y - 2
*=	z *= 10	z = z * 10
/=	a /= b	a = a / b
%=	c %= 3	c = c % 3

5. Sentinels

Sentinel: is a special value that marks the end of a sequence of items

- When program reaches a sentinel, it knows that the end of the sequence of items was reached, and the loop terminates
- Must be distinctive enough so as not to be mistaken for a regular value in the sequence
- Example: when reading an input file, empty line can be used as a sentinel

6. Input Validation Loops

Computer cannot tell the difference between good data and bad data

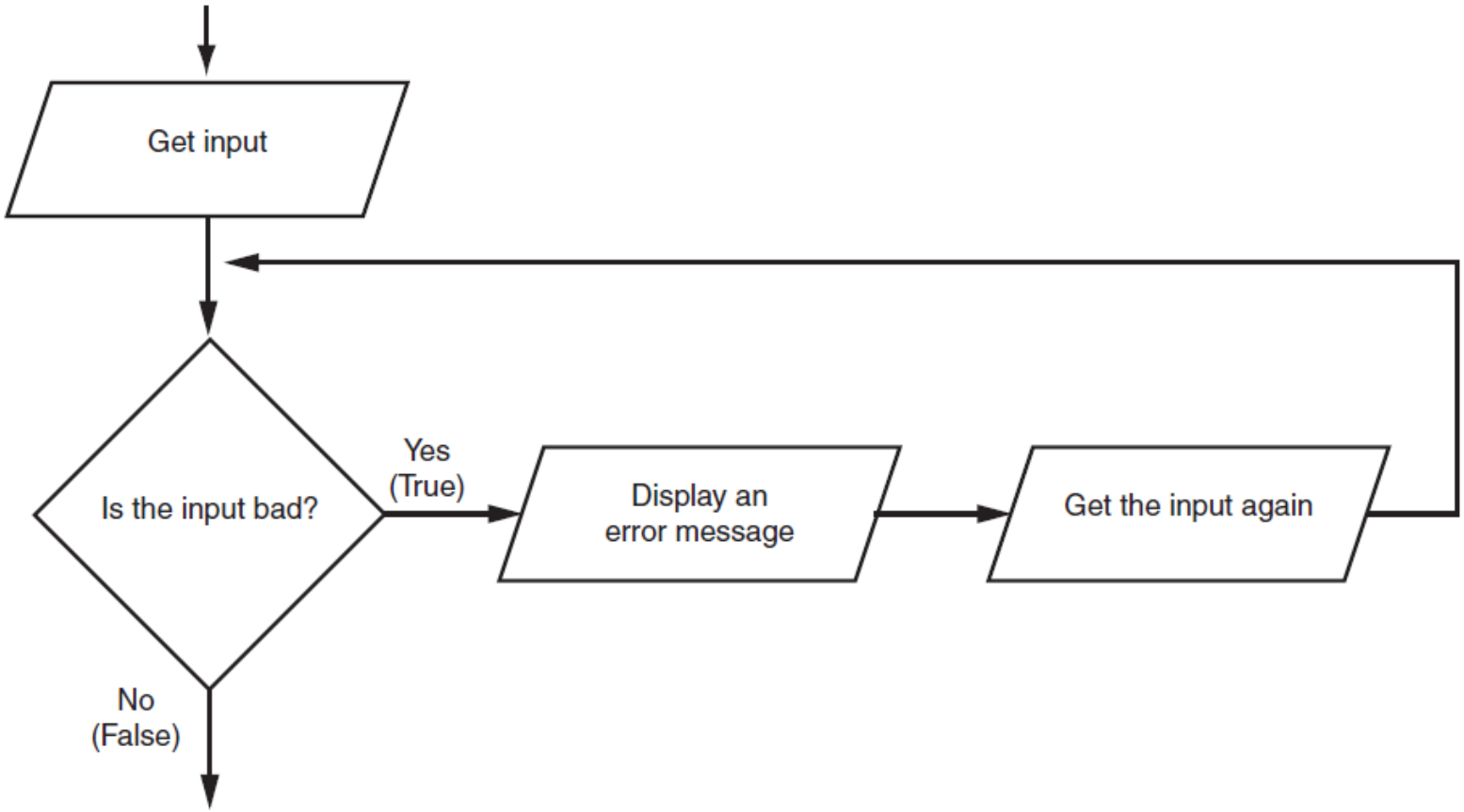
- If user provides bad input, program will produce bad output
- GIGO: garbage in, garbage out
- It is important to design program such that bad input is never accepted

Input Validation Loops (cont'd.)

- **Input validation: inspecting input before it is processed by the program**
  - If input is invalid, prompt user to enter correct data
  - Commonly accomplished using a `while` loop which repeats as long as the input is bad
    - If input is bad, display error message and receive another set of data
    - If input is good, continue to process the input

Input Validation Loops (cont'd.)

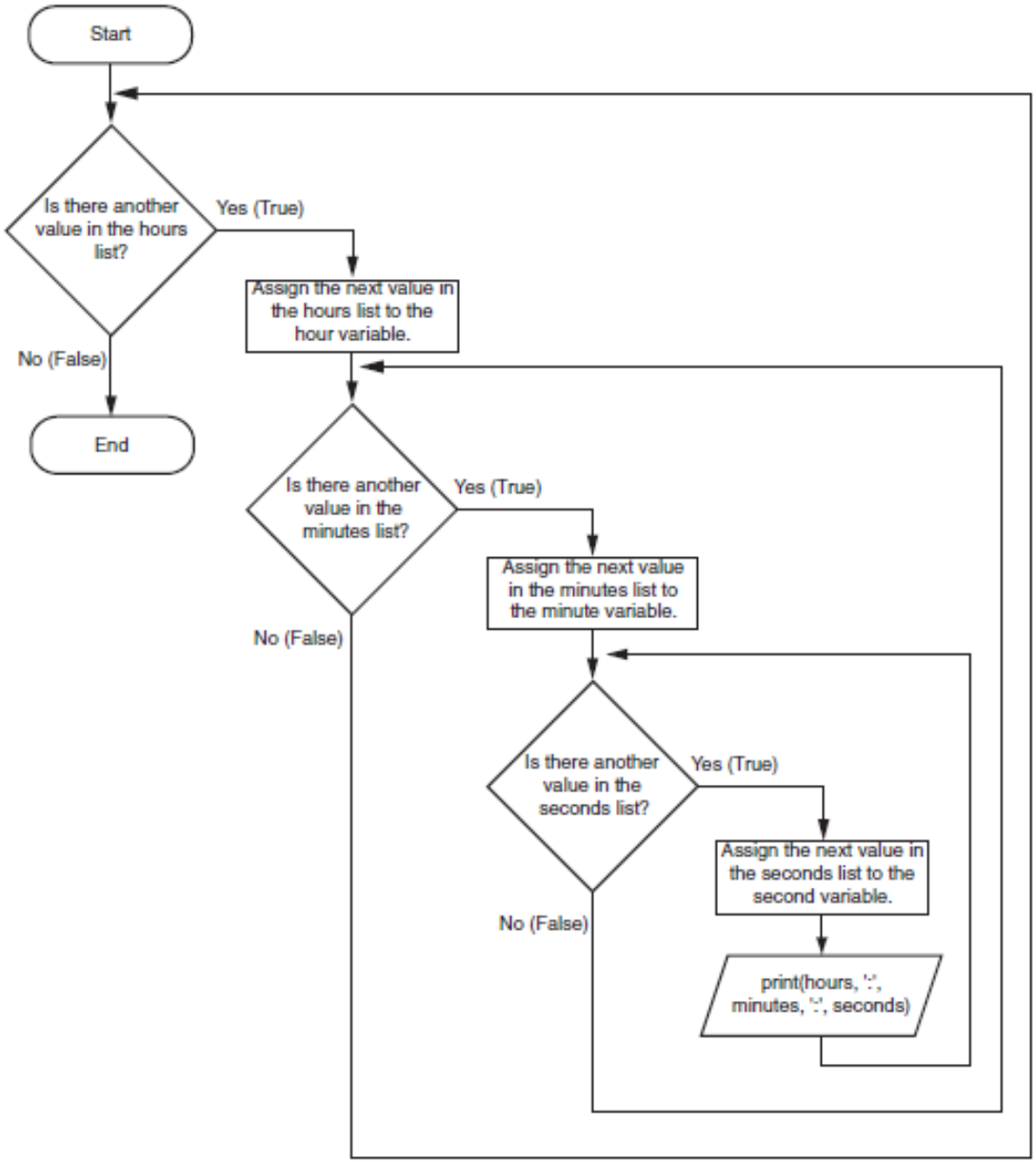
Figure 4-7 Logic containing an input validation loop



Nested Loops

- **Nested loop: loop that is contained inside another loop**
  - Example: analog clock works like a nested loop
    - Hours hand moves once for every twelve movements of the minutes hand: for each iteration of the “hours,” do twelve iterations of “minutes”
    - Seconds hand moves 60 times for each movement of the minutes hand: for each iteration of “minutes,” do 60 iterations of “seconds”

Figure 4-8 Flowchart for a clock simulator



Nested Loops (cont'd.)

- **Key points about nested loops:**
  - Inner loop goes through all of its iterations for each iteration of outer loop
  - Inner loops complete their iterations faster than outer loops
    - Total number of iterations in nested loop:

```
number_iterations_inner x number_iterations_outer
```

Exercises

#How many times will 'Hello World' be printed in the following program?

```
count = 1
while count > 10:
  print('Hello World')
```

#Rewrite the following code so it calls the range function instead of using the list [0, 1, 2, 3, 4, 5]:

```
for x in [0, 1, 2, 3, 4, 5]:
      print('I love programming!')
```

#What will the following codes display?

```
for number in range(6):
                    print(number)
```

```
for number in range(2, 6):
                    print(number)
```

```
for number in range(10, 5, -1):
                    print(number)
```

```
total = 0
for count in range(1, 6):
          total = total + count
print(total)
```
