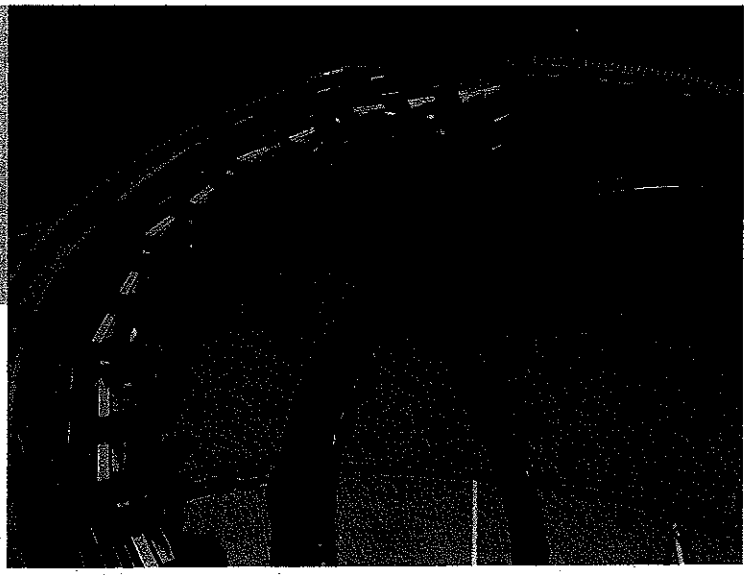
CHAPTER 4

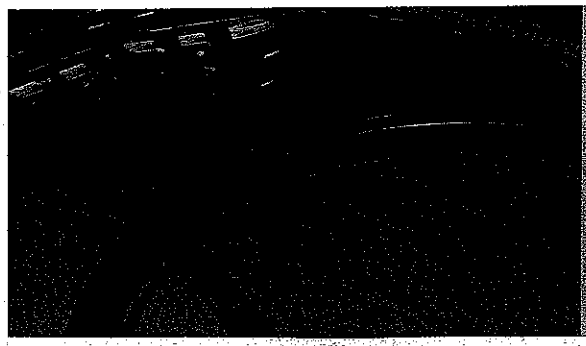
LOOPS



CHAPTER GOALS

- To implement `while` and `for` loops
- To hand-trace the execution of a program
- To become familiar with common loop algorithms
- To understand nested loops
- To process strings
- To use a computer for simulations

CHAPTER CONTENTS



In a loop, a part of a program is repeated over and over, until a specific goal is reached. Loops are important for calculations that require repeated steps and for processing input consisting of many data items. In this chapter, you will learn about loop statements in Python, as well as techniques for writing programs that process input and simulate activities in the real world.

4.1 The `while` Loop

In this section, you will learn about *loop statements* that repeatedly execute instructions until a goal has been reached.

Because the interest earned also earns interest, a bank balance grows exponentially.

Recall the investment problem from Chapter 1. You put $10,000 into a bank account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original investment?

In Chapter 1 we developed the following algorithm for this problem:

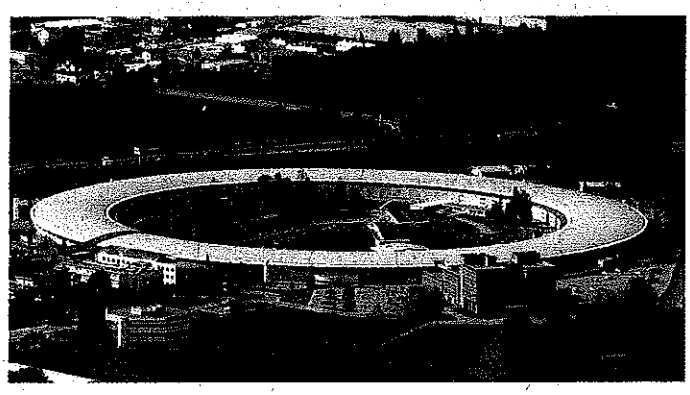
Start with a year value of 0, a column for the interest, and a balance of $10,000.

year	interest	balance
0		$10,000

Repeat the following steps while the balance is less than $20,000.
- **Add 1 to the year value.**
- **Compute the interest as balance x 0.05 (i.e., 5 percent interest).**
- **Add the interest to the balance.**

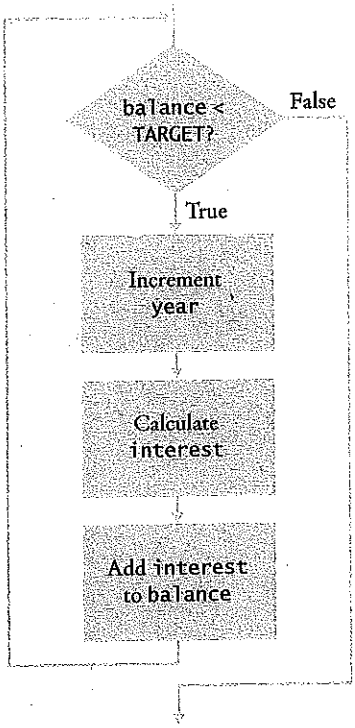
Report the final year value as the answer.

You now know how to create and update the variables in Python. What you don't yet know is how to carry out "Repeat steps while the balance is less than $20,000".



In a particle accelerator, subatomic particles traverse a loop-shaped tunnel multiple times, gaining the speed required for physical experiments. Similarly, in computer science, statements in a loop are executed while a condition is true.

Figure 1 Flowchart of a `while` Loop



A `while` loop executes instructions repeatedly while a condition is true.

In Python, the `while` statement implements such a repetition (see Syntax 4.1). It has the form

```
while condition :
   statements
```

As long as the condition remains true, the statements inside the `while` statement are executed. These statements are called the **body** of the `while` statement.

In our case, we want to increment the year counter and add interest while the balance is less than the target balance of $20,000:

```
while balance < TARGET :
   year = year + 1
   interest = balance * RATE / 100
   balance = balance + interest
```

A `while` statement is an example of a **loop**. If you draw a flowchart, the flow of execution loops again to the point where the condition is tested (see Figure 1).

Syntax 4.1 `while` Statement

Syntax `while condition :`
statements

This variable is initialized outside the loop and updated in the loop.

Beware of "off-by-one" errors in the loop condition. See page 161.

Put a colon here! See page 95.

If the condition never becomes false, an infinite loop occurs. See page 161.

```
balance = 10000.0
.
.
.
while balance < TARGET :
   interest = balance * RATE / 100
   balance = balance + interest
```

These statements are executed while the condition is true.

Statements in the body of a compound statement must be indented to the same column position. See page 95.

It often happens that you want to execute a sequence of statements a given number of times. You can use a `while` loop that is controlled by a counter, as in the following:

```
counter = 1    # Initialize the counter.
while counter <= 10 :    # Check the counter.
   print(counter)
   counter = counter + 1    # Update the loop variable.
```

Some people call this loop *count-controlled*. In contrast, the `while` loop in the `doubleinv.py` program can be called an *event-controlled* loop because it executes until

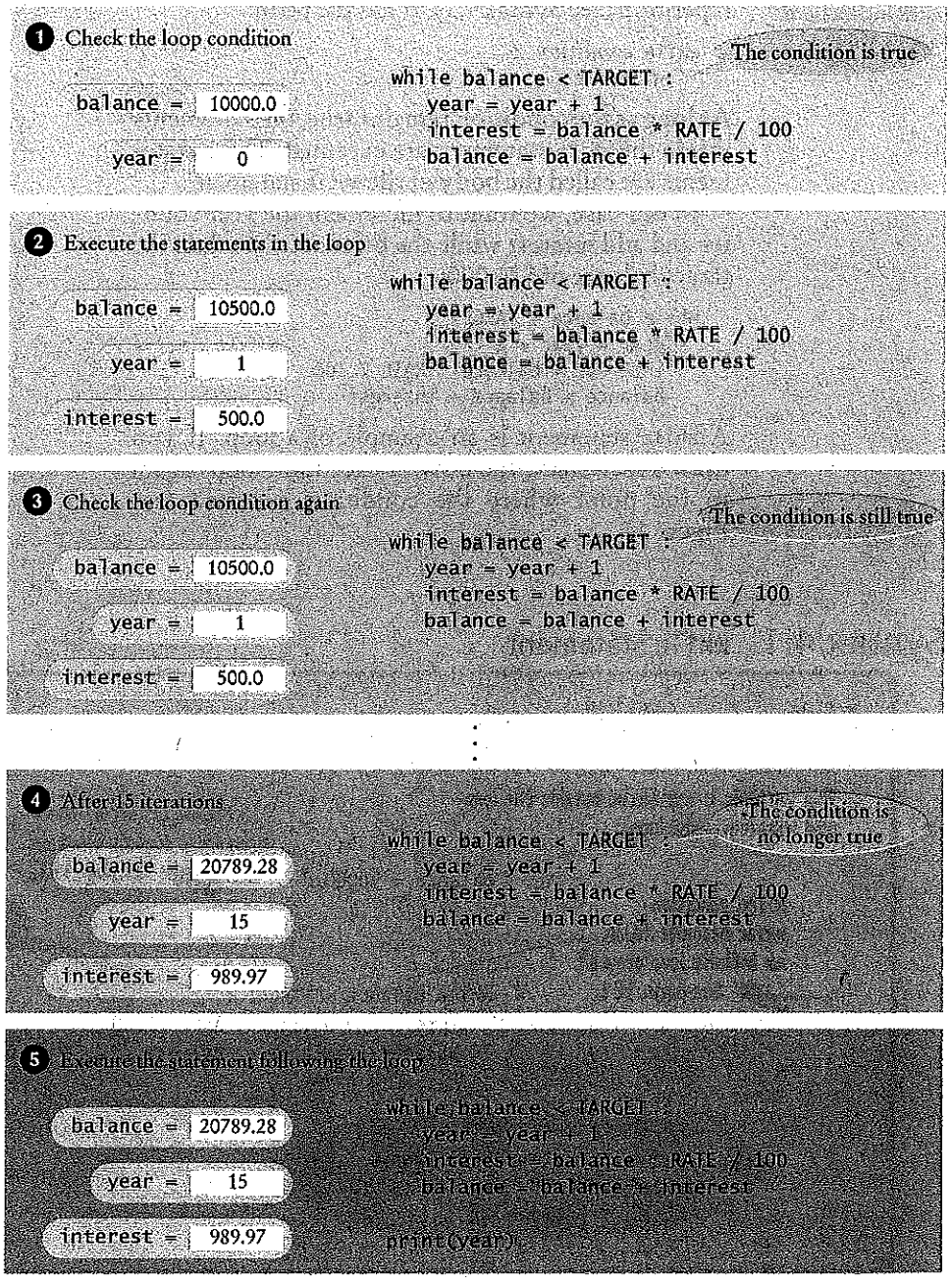


Figure 2
Execution of the `doubleinv.py` Loop

an event occurs; namely that the balance reaches the target. Another commonly used term for a count-controlled loop is *definite*. You know from the outset that the loop body will be executed a definite number of times; in our example ten times. In contrast, you do not know how many iterations it takes to accumulate a target balance. Such a loop is called *indefinite*.

Here is the program that solves the investment problem. Figure 2 illustrates the program's execution.

ch04/doubleinv.py

```
1  ##
2  #  This program computes the time required to double an investment.
3  #
4
5  # Create constant variables.
6  RATE = 5.0
7  INITIAL_BALANCE = 10000.0
8  TARGET = 2 * INITIAL_BALANCE
9
10 # Initialize variables used with the loop.
11 balance = INITIAL_BALANCE
12 year = 0
13
14 # Count the years required for the investment to double.
15 while balance < TARGET :
16    year = year + 1
17    interest = balance * RATE / 100
18    balance = balance + interest
19
20 # Print the results.
21 print("The investment doubled after", year, "years.")
```

Program Run

```
The investment doubled after 15 years.
```

1. How many years does it take for the investment to triple? Modify the program and run it.
2. If the interest rate is 10 percent per year, how many years does it take for the investment to double? Modify the program and run it.
3. Modify the program so that the balance after each year is printed. How did you do that?
4. Suppose we change the program so that the condition of the `while` loop is

   ```
   while balance <= TARGET :
   ```

   What is the effect on the program? Why?
5. What does the following loop print?

   ```
   n = 1
   while n < 100 :
      n = 2 * n
      print(n)
   ```

Practice It Now you can try these exercises at the end of the chapter: R4.1, R4.5, P4.13.

Table 1 while Loop Examples

Loop	Output	Explanation

`i = 0`
`total = 0`
`while total < 10 :`
`  i = i + 1`
`  total = total + i`
`  print(i, total)`
	`1 1`
	`2 3`
	`3 6`
	`4 10`
		When total is 10, the loop condition is false, and the loop ends.

`i = 0`
`total = 0`
`while total < 10 :`
`  i = i + 1`
`  total = total - 1`
`  print(i, total)`
	`1 -1`
	`2 -3`
	`3 -6`
	`4 -10`
	`. . .`
		Because total never reaches 10, this is an "infinite loop" (see Common Error 4.2 on page 161).

`i = 0`
`total = 0`
`while total < 0 :`
`  i = i + 1`
`  total = total - i`
`  print(i, total)`
	(No output)	The statement `total < 0` is false when the condition is first checked, and the loop is never executed.

`i = 0`
`total = 0`
`while total >= 10 :`
`  i = i + 1`
`  total = total + i`
`  print(i, total)`
	(No output)	The programmer probably thought, "Stop when the sum is at least 10." However, the loop condition controls when the loop is executed, not when it ends (see Common Error 4.2 on page 161).

`i = 0`
`total = 0`
`while total >= 0 :`
`  i = i + 1`
`  total = total + i`
`print(i, total)`
	(No output, program does not terminate)	Because total will always be greater than or equal to 0, the loop runs forever. It produces no output because the `print` function is outside the body of the loop, as indicated by the indentation.

Common Error 4.1

Don't Think "Are We There Yet?"

When doing something repetitive, most of us want to know when we are done. For example, you may think, "I want to get at least $20,000," and set the loop condition to

```
balance >= TARGET
```

But the `while` loop thinks the opposite: How long am I allowed to keep going? The correct loop condition is

```
while balance < TARGET :
```

In other words: "Keep at it while the balance is less than the target."

When writing a loop condition, don't ask, "Are we there yet?" The condition determines how long the loop will keep going.

Common Error 4.2

Infinite Loops

A very annoying loop error is an *infinite loop:* a loop that runs forever and can be stopped only by killing the program or restarting the computer. If there are output statements in the loop, then many lines of output flash by on the screen. Otherwise, the program just sits there and *hangs*, seeming to do nothing. On some systems, you can kill a hanging program by hitting Ctrl + C. On others, you can close the window in which the program runs.

A common reason for infinite loops is forgetting to update the variable that controls the loop:

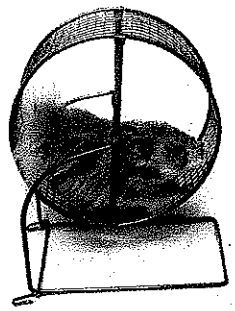
```
year = 1
while year <= 20 :
   interest = balance * RATE / 100
   balance = balance + interest
```

Here the programmer forgot to add a `year = year + 1` command in the loop. As a result, the year always stays at 1, and the loop never comes to an end.

Another common reason for an infinite loop is accidentally incrementing a counter that should be decremented (or vice versa). Consider this example:

```
year = 20
while year > 0 :
   interest = balance * RATE / 100
   balance = balance + interest
   year = year + 1
```

The `year` variable really should have been decremented, not incremented. This is a common error because incrementing counters is so much more common than decrementing that your fingers may type the + on autopilot. As a consequence, `year` is always larger than 0, and the loop never ends.



Like this hamster who can't stop running in the treadmill, an infinite loop never ends.

Common Error 4.3

Off-by-One Errors

Consider our computation of the number of years that are required to double an investment:

```
year = 0
while balance < TARGET :
   year = year + 1
   interest = balance * RATE / 100
   balance = balance + interest
print("The investment doubled after", year, "years.")
```

Should `year` start at 0 or at 1? Should you test for `balance < TARGET` or for `balance <= TARGET`? It is easy to be *off by one* in these expressions.

Some people try to solve **off-by-one errors** by randomly inserting +1 or -1 until the program seems to work, which is a terrible strategy. It can take a long time to test all the various possibilities. Expending a small amount of mental effort is a real time saver.

Fortunately, off-by-one errors are easy to avoid, simply by thinking through a couple of test cases and using the information from the test cases to come up with a rationale for your decisions.

An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.

Should year start at 0 or at 1? Look at a scenario with simple values: an initial balance of $100 and an interest rate of 50 percent. After year 1, the balance is $150, and after year 2 it is $225, or over $200. So the investment doubled after 2 years. The loop executed two times, incrementing year each time. Hence year must start at 0, not at 1.

year	balance
0	$100
1	$150
2	$225

In other words, the balance variable denotes the balance after the end of the year. At the outset, the balance variable contains the balance after year 0 and not after year 1.

Next, should you use a < or <= comparison in the test? This is harder to figure out, because it is rare for the balance to be exactly twice the initial balance. There is one case when this happens, namely when the interest rate is 100 percent. The loop executes once. Now year is 1, and balance is exactly equal to `2 * INITIAL_BALANCE`. Has the investment doubled after one year? It has. Therefore, the loop should not execute again. If the test condition is `balance < TARGET`, the loop stops, as it should. If the test condition had been `balance <= TARGET`, the loop would have executed once more.

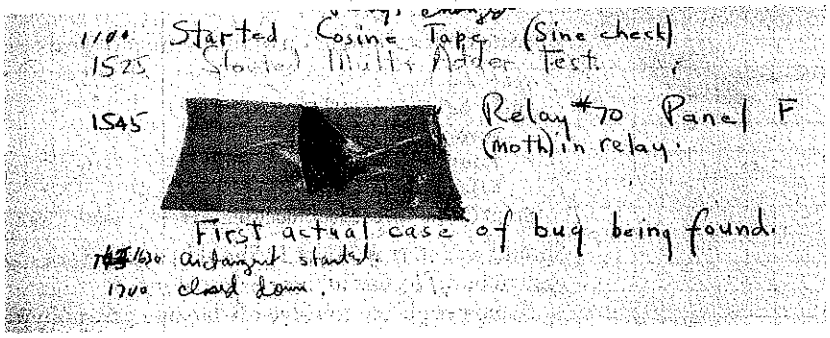
In other words, you keep adding interest while the balance *has not yet doubled.*

Computing & Society 4.1 The First Bug

According to legend, the first bug was found in the Mark II, a huge electromechanical computer at Harvard University. It really was caused by a bug—a moth was trapped in a relay switch.

Actually, from the note that the operator left in the log book next to the moth (see the photo), it appears as if the term "bug" had already been in active use at the time.

The pioneering computer scientist Maurice Wilkes wrote, "Somehow, at the Moore School and afterwards, one had always assumed there would be no particular difficulty in getting programs right. I can remember the exact instant in time at which it dawned on me that a great part of my future life would be spent finding mistakes in my own programs."



The First Bug

4.2 Problem Solving: Hand-Tracing

Hand-tracing is a simulation of code execution in which you step through instructions and track the values of the variables.

In Programming Tip 3.2, you learned about the method of hand-tracing. When you hand-trace code or pseudocode, you write the names of the variables on a sheet of paper, mentally execute each step of the code, and update the variables.

It is best to have the code written or printed on a sheet of paper. Use a marker, such as a paper clip, to mark the current line. Whenever a variable changes, cross out the old value and write the new value below. When a program produces output, also write down the output in another column.

Consider this example. What value is displayed?

```
n = 1729
total = 0
while n > 0 :
   digit = n % 10
   total = total + digit
   n = n // 10

print(total)
```

There are three variables: `n`, `total`, and `digit`.

n	total	digit

The first two variables are initialized with 1729 and 0 before the loop is entered.

```
n = 1729
total = 0
while n > 0 :
   digit = n % 10
   total = total + digit
   n = n // 10

print(total)
```

n	total	digit
1729	0	

Because `n` is greater than zero, enter the loop. The variable `digit` is set to 9 (the remainder of dividing 1729 by 10). The variable `total` is set to 0 + 9 = 9.

```
n = 1729
total = 0
while n > 0 :
   digit = n % 10
   total = total + digit
   n = n // 10

print(total)
```

n	total	digit
1729	~~0~~	
	9	9

Finally, `n` becomes 172. (Recall that the remainder in the division 1729 // 10 is discarded because the // operator performs floor division.)

Cross out the old values and write the new ones under the old ones.

```
n = 1729
total = 0
while n > 0 :
   digit = n % 10
   total = total + digit
   n = n // 10

print(total)
```

n	total	digit
~~1729~~	~~0~~	
172	9	9

Now check the loop condition again.

```
n = 1729
total = 0
while n > 0 :
   digit = n % 10
   total = total + digit
   n = n // 10

print(total)
```

Because n is still greater than zero, repeat the loop. Now digit becomes 2, total is set to 9 + 2 = 11, and n is set to 17.

n	total	digit
~~1729~~	~~0~~	
~~172~~	~~9~~	~~9~~
17	11	2

Repeat the loop once again, setting digit to 7, total to 11 + 7 = 18, and n to 1.

n	total	digit
~~1729~~	~~0~~	
~~172~~	~~9~~	~~9~~
~~17~~	~~11~~	~~2~~
1	18	7

Enter the loop for one last time. Now digit is set to 1, total to 19, and n becomes zero.

n	total	digit
~~1729~~	~~0~~	
~~172~~	~~9~~	~~9~~
~~17~~	~~11~~	~~2~~
~~1~~	~~18~~	~~7~~
0	19	1

```
n = 1729
total = 0
while n > 0 :
   digit = n % 10
   total = total + digit
   n = n // 10

print(total)
```

Because n equals zero, this condition is not true.

The condition `n > 0` is now false. Continue with the statement after the loop.

```
n = 1729
total = 0
while n > 0 :
   digit = n % 10
   total = total + digit
   n = n // 10

print(total)
```

n	total	digit	output
1729	0		
172	9	9	
17	11	2	
1	18	7	
0	19	1	19

This statement is an output statement. The value that is output is the value of `total`, which is 19.

Of course, you can get the same answer by just running the code. However, hand-tracing can give you *insight* that you would not get if you simply ran the code. Consider again what happens in each iteration:

- We extract the last digit of `n`.
- We add that digit to `total`.
- We strip the digit off of `n`.

Hand-tracing can help you understand how an unfamiliar algorithm works.

In other words, the loop computes the sum of the digits in `n`. You now know what the loop does for any value of `n`, not just the one in the example. (Why would anyone want to compute the sum of the digits? Operations of this kind are useful for checking the validity of credit card numbers and other forms of ID numbers—see Exercise P4.33.)

Hand-tracing can show errors in code or pseudocode.

Hand-tracing does not just help you understand code that works correctly. It is a powerful technique for finding errors in your code. When a program behaves in a way that you don't expect, get out a sheet of paper and track the values of the variables as you mentally step through the code.

You don't need a working program to do hand-tracing. You can hand-trace pseudocode. In fact, it is an excellent idea to hand-trace your pseudocode before you go to the trouble of translating it into actual code, to confirm that it works correctly.

6. Hand-trace the following code, showing the value of `n` and the output.

```
n = 5
while n >= 0 :
   n = n - 1
   print(n)
```

7. Hand-trace the following code, showing the value of `n` and the output.

```
n = 1
while n <= 3 :
   print(n)
   n = n + 1
```

8. Hand-trace the following code, assuming that a is 2 and n is 4. Then explain what the code does for arbitrary values of a and n.

```
r = 1
i = 1
while i <= n :
   r = r * a
   i = i + 1
```

9. Hand-trace the following code. What error do you observe?

```
n = 1
while n != 50 :
   print(n)
   n = n + 10
```

10. The following pseudocode is intended to count the number of digits in the number n:

```
count = 1
temp = n
while temp > 10
   Increment count.
   Divide temp by 10.0.
```

Hand-trace the pseudocode for n = 123 and n = 100. What error do you find?

Practice It Now you can try these exercises at the end of the chapter: R4.3, R4.6.

4.3 Application: Processing Sentinel Values

In this section, you will learn how to write loops that read and process a sequence of input values.

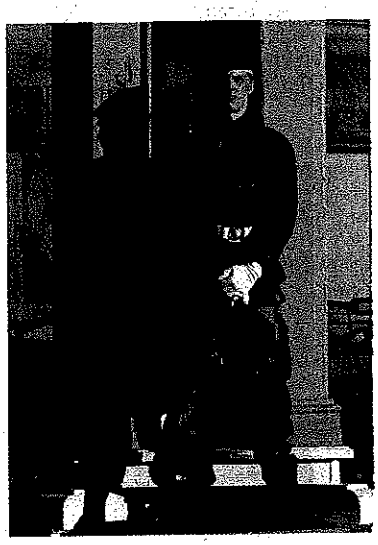
Whenever you read a sequence of inputs, you need to have some method of indicating the end of the sequence. Sometimes you are lucky and no input value can be zero. Then you can prompt the user to keep entering numbers, or 0 to finish the sequence. If zero is allowed but negative numbers are not, you can use –1 to indicate termination.

Such a value, which is not an actual input, but serves as a signal for termination, is called a **sentinel**.

A sentinel value denotes the end of a data set, but it is not part of the data.

Let's put this technique to work in a program that computes the average of a set of salary values. In our sample program, we will use any negative value as the sentinel. An employee would surely not work for a negative salary, but there may be volunteers who work for free.

Inside the loop, we read an input. If the input is non-negative, we process it. In order to compute the average, we need the total sum of all salaries, and the number of inputs.



In the military, a sentinel guards a border or passage. In computer science, a sentinel value denotes the end of an input sequence or the border between input sequences.

```
while . . . :
   salary = float(input("Enter a salary or -1 to finish: "))
   if salary >= 0.0 :
      total = total + salary
      count = count + 1
```

Any negative number can end the loop, but we prompt for a sentinel of –1 so that the user need not ponder which negative number to enter. Note that we stay in the loop while the sentinel value is *not* detected.

```
while salary >= 0.0 :
   . . .
```

There is just one problem: When the loop is entered for the first time, no data value has been read. We must make sure to initialize `salary` with a value that will satisfy the `while` loop condition so that the loop will be executed at least once.

```
salary = 0.0   # Any non-negative value will do.
```

After the loop has finished, we compute and print the average.

Here is the complete program:

ch04/sentinel.py

```
1  ##
2  #  This program prints the average of salary values that are terminated with a sentinel.
3  #
4
5  # Initialize variables to maintain the running total and count.
6  total = 0.0
7  count = 0
8
9  # Initialize salary to any non-sentinel value.
10 salary = 0.0
11
12 # Process data until the sentinel is entered.
13 while salary >= 0.0 :
14    salary = float(input("Enter a salary or -1 to finish: "))
15    if salary >= 0.0 :
16       total = total + salary
17       count = count + 1
18
19 # Compute and print the average salary.
20 if count > 0 :
21    average = total / count
22    print("Average salary is", average)
23 else :
24    print("No data was entered.")
```

Program Run

```
Enter a salary or -1 to finish: 10000
Enter a salary or -1 to finish: 10000
Enter a salary or -1 to finish: 40000
Enter a salary or -1 to finish: -1
Average salary is 20000.0
```

A pair of input operations, known as the priming and modification reads, can be used to read a sentinel-terminated sequence of values.

Some programmers don't like the "trick" of initializing the input variable with a value other than a sentinel. Although it solves the problem, it requires the use of an `if` statement in the body of the loop to test for the sentinel value. Another approach is to use two input statements, one before the loop to obtain the first value and another at the bottom of the loop to read additional values:

```
salary = float(input("Enter a salary or -1 to finish: "))
while salary >= 0.0 :
   total = total + salary
   count = count + 1
   salary = float(input("Enter a salary or -1 to finish: "))
```

If the first value entered by the user is the sentinel, then the body of the loop is never executed. Otherwise, the value is processed just as it was in the earlier version of the loop. The input operation before the loop is known as the *priming read*, because it prepares or initializes the loop variable.

The input operation at the bottom of the loop is used to obtain the next input. It is known as the *modification read*, because it modifies the loop variable inside the loop. Note that this is the last statement to be executed before the next iteration of the loop. If the user enters the sentinel value, then the loop terminates. Otherwise, the loop continues, processing the input.

Special Topic 4.1 shows a third approach for processing sentinel values that uses a Boolean variable.

Now consider the case in which any number (positive, negative, or zero) can be an acceptable input. In such a situation, you must use a sentinel that is not a number (such as the letter Q).

Because the `input` function obtains data from the user and returns it as a string, you can examine the string to see if the user entered the letter Q before converting the string to a numeric value for use in the calculations:

```
inputStr = input("Enter a value or Q to quit: ")
while inputStr != "Q" :
   value = float(inputStr)
   Process value.
   inputStr = input("Enter a value or Q to quit: ")
```

Note that the conversion to a floating-point value is performed as the first statement within the loop. By including it as the first statement, it handles the input string for both the priming read and the modification read.

Finally, consider the case where you prompt for multiple strings, for example, a sequence of names. We still need a sentinel to flag the end of the data extraction. Using a string such as Q is not such a good idea because that might be a valid input. You can use the empty string instead. When a user presses the Enter key without pressing any other keys, the `input` function returns the empty string:

```
name = input("Enter a name or press the Enter key to quit: ")
while name != "" :
   Process name.
   inputStr = input("Enter a name or press the Enter key to quit: ")
```

11. What does the `sentinel.py` program print when the user immediately types –1 when prompted for a value?

12. Why does the `sentinel.py` program have *two* checks of the form

```
salary >= 0
```

13. What would happen if the initialization of the `salary` variable in `sentinel.py` was changed to

```
salary = -1
```

14. In the second example of this section, we prompt the user "Enter a value or Q to quit." What happens when the user enters a different letter?

Practice It Now you can try these exercises at the end of the chapter: R4.14, P4.28, P4.29.

Special Topic 4.1

Processing Sentinel Values with a Boolean Variable

Sentinel values can also be processed using a Boolean variable for the loop termination:

```
done = False
while not done :
   value = float(input("Enter a salary or -1 to finish: "))
   if value < 0.0 :
      done = True
   else :
      Process value.
```

The actual test for loop termination is in the middle of the loop, not at the top. This is called a **loop and a half** because one must go halfway into the loop before knowing whether one needs to terminate. As an alternative, you can use the `break` statement:

```
while True :
   value = float(input("Enter a salary or -1 to finish: "))
   if value < 0.0 :
      break
   Process value.
```

The `break` statement breaks out of the enclosing loop, independent of the loop condition. When the `break` statement is encountered, the loop is terminated, and the statement following the loop is executed.

In the loop-and-a-half case, `break` statements can be beneficial. But it is difficult to lay down clear rules as to when they are safe and when they should be avoided. We do not use the `break` statement in this book.

Special Topic 4.2

Redirection of Input and Output

Use input redirection to read input from a file. Use output redirection to capture program output in a file.

Consider the `sentinel.py` program that computes the average value of an input sequence. If you use such a program, then it is quite likely that you already have the values in a file, and it seems a shame that you have to type them all in again. The command line interface of your operating system provides a way to link a file to the input of a program, as if all the characters in the file had actually been typed by a user. If you type

```
python sentinel.py < numbers.txt
```

the program is executed, but it no longer expects input from the keyboard. All input commands get their input from the file `numbers.txt`. This process is called *input redirection*.

Input redirection is an excellent tool for testing programs. When you develop a program and fix its bugs, it is boring to keep entering the same input every time you run the program. Spend a few minutes putting the inputs into a file, and use redirection.

You can also redirect output. In this program, that is not terribly useful. If you run

```
python sentinel.py < numbers.txt > output.txt
```

the file `output.txt` contains the input prompts and the output, such as

```
Enter a salary or -1 to finish:
Enter a salary or -1 to finish:
Enter a salary or -1 to finish:
Enter a salary or -1 to finish:
Average salary is 15
```

However, redirecting output is obviously useful for programs that produce lots of output. You can format or print the file containing the output.

4.4 Problem Solving: Storyboards

When you design a program that interacts with a user, you need to make a plan for that interaction. What information does the user provide, and in which order? What information will your program display, and in which format? What should happen when there is an error? When does the program quit?

A storyboard consists of annotated sketches for each step in an action sequence.

This planning is similar to the development of a movie or a computer game, where *storyboards* are used to plan action sequences. A storyboard is made up of panels that show a sketch of each step. Annotations explain what is happening and note any special situations. Storyboards are also used to develop software—see Figure 3.

Making a storyboard is very helpful when you begin designing a program. You need to ask yourself which information you need in order to compute the answers that the program user wants. You need to decide how to present those answers. These

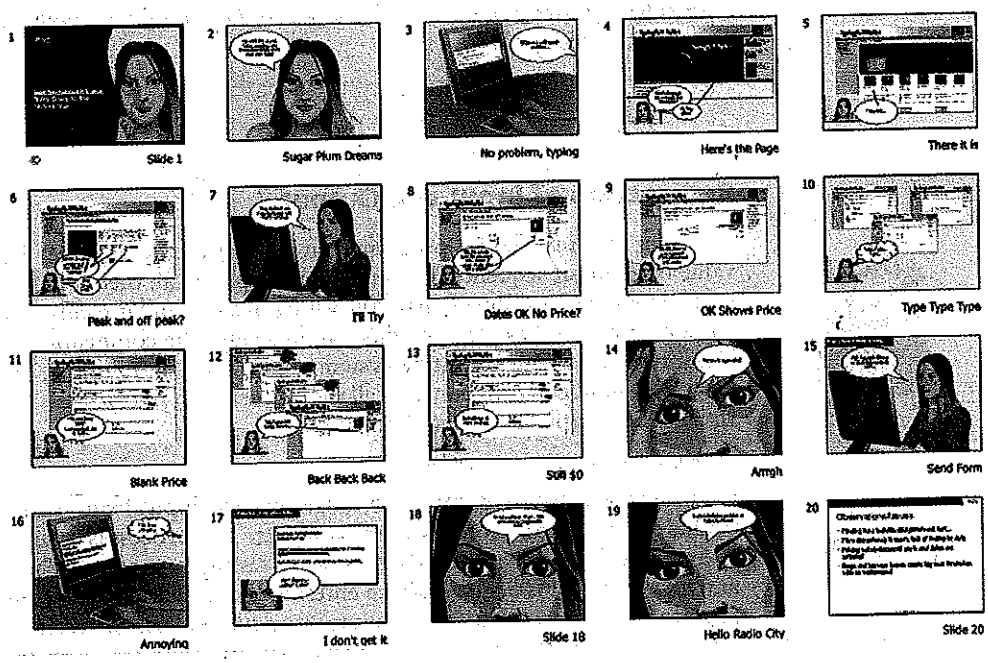


Figure 3 Storyboard for the Design of a Web Application

Developing a storyboard helps you understand the inputs and outputs that are required for a program.

are important considerations that you want to settle before you design an algorithm for computing the answers.

Let's look at a simple example. We want to write a program that helps users with questions such as "How many tablespoons are in a pint?" or "How many inches are in 30 centimeters?"

What information does the user provide?

- The quantity and unit to convert from
- The unit to convert to

What if there is more than one quantity? A user may have a whole table of centimeter values that should be converted into inches.

What if the user enters units that our program doesn't know how to handle, such as ångström?

What if the user asks for impossible conversions, such as inches to gallons?

Let's get started with a storyboard panel. It is a good idea to write the user inputs in a different color. (Underline them if you don't have a color pen handy.)

Converting a Sequence of Values

```
What unit do you want to convert from? cm
What unit do you want to convert to? in
Enter values, terminated by zero
30
30 cm = 11.81 in
100
100 cm = 39.37 in
0
What unit do you want to convert from?
```

Allows conversion of multiple values

Format makes clear what got converted

The storyboard shows how we deal with a potential confusion. A user who wants to know how many inches are 30 centimeters may not read the first prompt carefully and specify inches. But then the output is "30 in = 76.2 cm", alerting the user to the problem.

The storyboard also raises an issue. How is the user supposed to know that "cm" and "in" are valid units? Would "centimeter" and "inches" also work? What happens when the user enters a wrong unit? Let's make another storyboard to demonstrate error handling.

Handling Unknown Units (needs improvement)

```
What unit do you want to convert from? cm
What unit do you want to convert to? inches
Sorry, unknown unit.
What unit do you want to convert to? inch
Sorry, unknown unit.
What unit do you want to convert to? grrr
```

To eliminate frustration, it is better to list the units that the user can supply.

```
From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal): cm
To unit: in
```

No need to list the units again

We switched to a shorter prompt to make room for all the unit names. Exercise R4.20 explores an alternative approach.

There is another issue that we haven't addressed yet. How does the user quit the program? The first storyboard suggests that the program will go on forever.

We can ask the user after seeing the sentinel that terminates an input sequence.

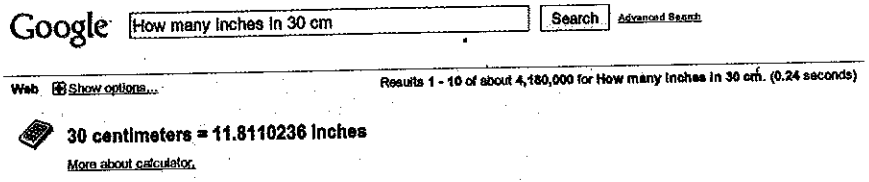
Exiting the Program

```
From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal): cm
To unit: in
Enter values, terminated by zero
30
30 cm = 11.81 in
0
More conversions (y, n)? n
(Program exits)
```

Sentinel triggers the prompt to exit

As you can see from this case study, a storyboard is essential for developing a working program. You need to know the flow of the user interaction in order to structure your program.

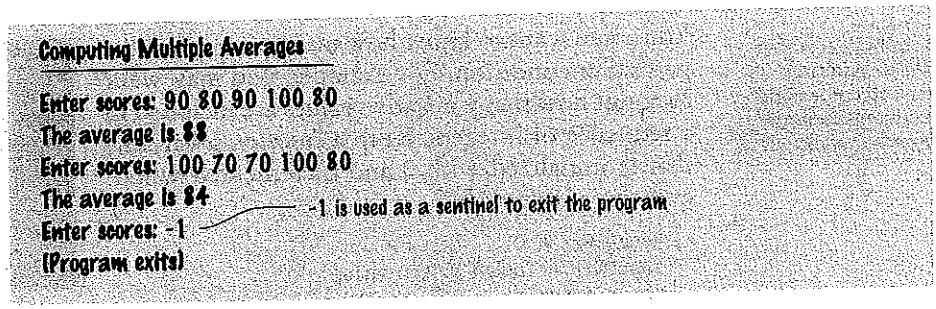
15. Provide a storyboard panel for a program that reads a number of test scores and prints the average score. The program only needs to process one set of scores. Don't worry about error handling.
16. Google has a simple interface for converting units. You just type the question, and you get the answer.



Make storyboards for an equivalent interface in a Python program. Show a scenario in which all goes well, and show the handling of two kinds of errors.

17. Consider a modification of the program in Self Check 15. Suppose we want to drop the lowest score before computing the average. Provide a storyboard for the situation in which a user only provides one score.

18. What is the problem with implementing the following storyboard in Python?



19. Produce a storyboard for a program that compares the growth of a $10,000 investment for a given number of years under two interest rates.

Practice It Now you can try these exercises at the end of the chapter: R4.20, R4.21, R4.22.

4.5 Common Loop Algorithms

In the following sections, we discuss some of the most common algorithms that are implemented as loops. You can use them as starting points for your loop designs.

4.5.1 Sum and Average Value

To compute an average, keep a total and a count of all values.

Computing the sum of a number of inputs is a very common task. Keep a *running total*, a variable to which you add each input value. Of course, the total should be initialized with 0.

```
total = 0.0
inputStr = input("Enter value: ")
while inputStr != "" :
   value = float(inputStr)
   total = total + value
   inputStr = input("Enter value: ")
```

Note that the total variable is created and initialized outside the loop. We want the loop to add each value entered by the user to the variable.

To compute an average, count how many values you have, and divide by the count. Be sure to check that the count is not zero.

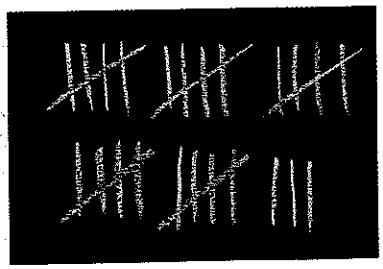
```
total = 0.0
count = 0
inputStr = input("Enter value: ")
while inputStr != "" :
   value = float(inputStr)
   total = total + value
   count = count + 1
   inputStr = input("Enter value: ")

if count > 0 :
   average = total / count
else :
   average = 0.0
```

4.5.2 Counting Matches

To count values that fulfill a condition, check all values and increment a counter for each match.

You often want to know how many values fulfill a particular condition. For example, you may want to count how many negative values are included in a sequence of integers. Keep a *counter*, a variable that is initialized with 0 and incremented whenever there is a match.



In a loop that counts matches, a counter is incremented whenever a match is found.

```
negatives = 0
inputStr = input("Enter value: ")
while inputStr != "" :
   value = int(inputStr)
   if value < 0 :
      negatives = negatives + 1
   inputStr = input("Enter value: ")

print("There were", negatives, "negative values.")
```

Note that the `negatives` variable is created and initialized outside the loop. We want the loop to increment `negatives` by 1 for each negative value entered by the user.

4.5.3 Prompting Until a Match is Found

In Chapter 3, we checked to be sure the user-supplied values were valid before they were used in a computation. If invalid data was entered, we printed an error message and ended the program. Instead of ending the program, however, you should keep asking the user to enter the data until a correct value is provided. For example, suppose you are asking the user to enter a positive value < 100:

```
valid = False
while not valid :
   value = int(input("Please enter a positive value < 100: "))
   if value > 0 and value < 100 :
      valid = True
   else :
      print("Invalid input.")
```

4.5.4 Maximum and Minimum

To find the largest value, update the largest value seen so far whenever you see a larger one.

To compute the largest value in a sequence, keep a variable that stores the largest element that you have encountered, and update it when you find a larger one:

```
largest = int(input("Enter a value: "))
inputStr = input("Enter a value: ")
while inputStr != "" :
   value = int(inputStr)
   if value > largest :
      largest = value
   inputStr = input("Enter a value: ")
```

This algorithm requires that there is at least one input, which is used to initialize the `largest` variable. The second input operation acts as the priming read for the loop.

To compute the smallest value, simply reverse the comparison:

```
smallest = int(input("Enter a value: "))
inputStr = input("Enter a value: ")
while inputStr != "" :
   value = int(inputStr)
   if value < smallest :
      smallest = value
   inputStr = input("Enter a value: ")
```

To find the height of the tallest bus rider, remember the largest value so far, and update it whenever you see a taller one.

4.5.5 Comparing Adjacent Values

To compare adjacent inputs, store the preceding input in a variable.

When processing a sequence of values in a loop, you sometimes need to compare a value with the value that just preceded it. For example, suppose you want to check whether a sequence of inputs such as `1 7 2 9 9 4 9` contains adjacent duplicates.

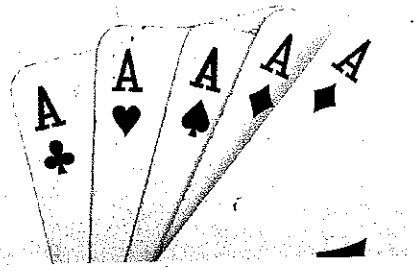
Now you face a challenge. Consider the typical loop for reading a value:

```
inputStr = input("Enter a value: ")
while inputStr != "" :
   value = int(inputStr)
   . . .
   inputStr = input("Enter a value: ")
```

How can you compare the current input with the preceding one? At any time, `value` contains the current input, overwriting the previous one.

The answer is to store the previous input, like this:

```
inputStr = input("Enter a value: ")
while inputStr != "" :
   previous = value
   value = int(inputStr)
   if value == previous :
      print("Duplicate input")
   inputStr = input("Enter a value: ")
```



When comparing adjacent values, store the previous value in a variable.

One problem remains. When the loop is entered for the first time, `value` has not yet been assigned a value. You can solve this problem with an initial input operation outside the loop:

```
value = int(input("Enter a value: "))
inputStr = input("Enter a value: ")
while inputStr != "" :
   previous = value
   value = int(inputStr)
   if value == previous :
      print("Duplicate input")
   inputStr = input("Enter a value: ")
```

Here is a sample program that illustrates some of the common loop algorithms:

ch04/grades.py

```
##
#  This program computes information related to a sequence of grades obtained
#  from the user. It computes the number of passing and failing grades,
#  computes the average grade and finds the highest and lowest grade.
#

# Initialize the counter variables.
numPassing = 0
numFailing = 0

# Initialize the variables used to compute the average.
total = 0
count = 0

# Initialize the min and max variables.
minGrade = 100.0    # Assuming 100 is the highest grade possible.
maxGrade = 0.0

# Use an event-controlled loop with a priming read to obtain the grades.
grade = float(input("Enter a grade or -1 to finish: "))
while grade >= 0.0 :
   # Increment the passing or failing counter.
   if grade >= 60.0 :
      numPassing = numPassing + 1
   else :
      numFailing = numFailing + 1

   # Determine if the grade is the min or max grade.
   if grade < minGrade :
      minGrade = grade
   if grade > maxGrade :
      maxGrade = grade

   # Add the grade to the running total.
   total = total + grade
   count = count + 1

   # Read the next grade.
   grade = float(input("Enter a grade or -1 to finish: "))

# Print the results.
if count > 0 :
   average = total / count
   print("The average grade is %.2f" % average)
   print("Number of passing grades is", numPassing)
   print("Number of failing grades is", numFailing)
   print("The maximum grade is %.2f" % maxGrade)
   print("The minimum grade is %.2f" % minGrade)
```

20. What total is computed when no user input is provided in the algorithm in Section 4.5.1?
21. How do you compute the total of all positive inputs?

22. Why is the input string in the algorithm in Section 4.5.2 converted to an integer inside the loop instead of immediately when the value is read from the user?
23. What is wrong with the following loop for finding the smallest input value?

```
smallest = 0
inputStr = input("Enter a value: ")
while inputStr != "" :
   value = int(inputStr)
   if value < smallest :
      smallest = value
   inputStr = input("Enter a value: ")
```

24. What happens with the algorithm in Section 4.5.4 when no input is provided at all? How can you overcome that problem?

Practice It Now you can try these exercises at the end of the chapter: P4.2, P4.5.

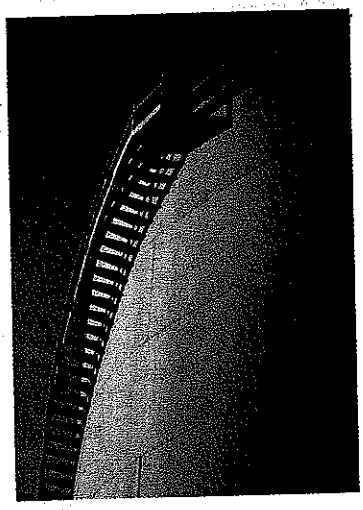
4.6 The for Loop

Often, you will need to visit each character in a string. The `for` loop (see Syntax 4.2) makes this process particularly easy to program. For example, suppose we want to print a string, with one character per line. We cannot simply print the string using the `print` function. Instead, we need to iterate over the characters in the string and print each character individually. Here is how you use the `for` loop to accomplish this task:

```
stateName = "Virginia"
for letter in stateName :
   print(letter)
```

which results in the output

```
V
i
r
g
i
n
i
a
```



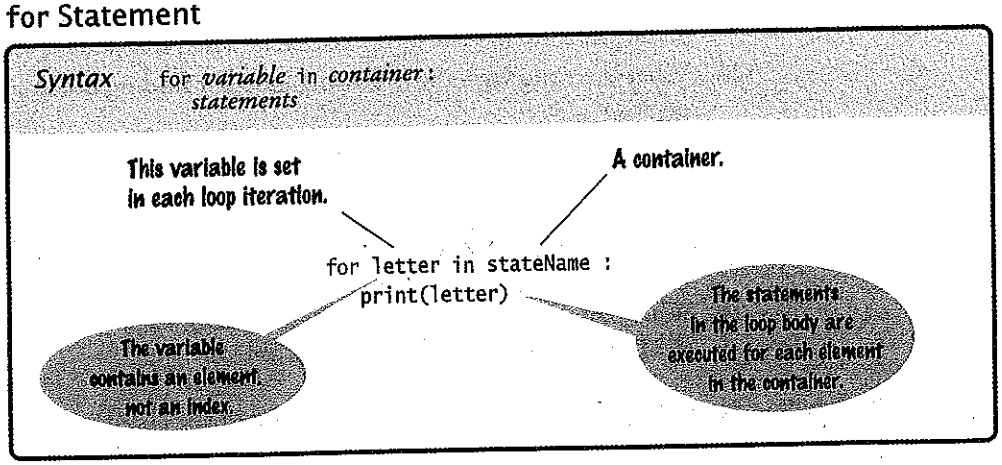
You can visualize the `for` *loop as an orderly sequence of steps.*

The loop body is executed for each character in the string `stateName`, starting with the first character. At the beginning of each loop iteration, the next character is assigned to the variable `letter`. Then the loop body is executed. You should read this loop as "for each `letter` in `stateName`". This loop is equivalent to the following `while` loop that uses an explicit index variable:

```
i = 0
while i < len(stateName) :
   letter = stateName[i]
   print(letter)
   i = i + 1
```

Note an important difference between the `for` loop and the `while` loop. In the `for` loop, the *element variable* is assigned `stateName[0]`, `stateName[1]`, and so on. In the `while` loop, the *index variable* `i` is assigned 0, 1, and so on.

Syntax 4.2 for Statement



The for loop is used to iterate over the elements of a container.

The for loop can be used to iterate over the contents of any **container**, which is an object that contains or stores a collection of elements. Thus, a string is a container that stores the collection of characters in the string. In later chapters, we will explore other types of containers available in Python.

As you have seen in prior sections, count-controlled loops that iterate over a range of integer values are very common. To simplify the creation of such loops, Python provides the range function for generating a sequence of integers that can be used with the for loop. The loop

```
for i in range(1, 10) :   # i = 1, 2, 3, ..., 9
   print(i)
```

prints the sequential values from 1 to 9. The range function generates a sequence of values based on its arguments. The first argument of the range function is the first value in the sequence. Values are included in the sequence while they are less than the second argument. This loop is equivalent to the following while loop:

```
i = 1
while i < 10 :
   print(i)
   i = i + 1
```

Note that the ending value (the second argument to the range function) is not included in the sequence, so the equivalent while loop stops before reaching that value, too.

By default, the range function creates the sequence in steps of 1. This can be changed by including a step value as the third argument to the function:

```
for i in range(1, 10, 2) :   # i = 1, 3, 5, ..., 9
   print(i)
```

Now, only the odd values from 1 to 9 are printed. We can also have the for loop count down instead of up:

```
for i in range(10, 0, -1) :   # i = 10, 9, 8, ..., 1
   print(i)
```

Finally, you can use the range function with a single argument. When you do, the range of values starts at zero.

```
for i in range(10) :   # i = 0, 1, 2, ..., 9
   print("Hello")   # Prints Hello ten times
```

Syntax 4.3 for Statement with range Function

Syntax `for` *variable* `in range(...) :`
statements

This variable is set, at the beginning of each iteration, to the next integer in the sequence generated by the range function.

The range function generates a sequence of integers over which the loop iterates.

```
for i in range(5) :
   print(i)  # Prints 0, 1, 2, 3, 4
```

With one argument, the sequence starts at 0. The argument is the first value NOT included in the sequence.

```
for i in range(1, 5) :
   print(i)  # Prints 1, 2, 3, 4
```

With two arguments, the sequence starts with the first argument.

```
for i in range(1, 11, 2) :
   print(i)  # Prints 1, 3, 5, 7, 9
```

With three arguments, the third argument is the step value.

In this form, the sequence is the values from 0 to one less than the argument, in steps of 1. This form is very useful when we need to simply execute the body of a loop a given number of times, as in the preceding example. See Table 2 for additional examples.

Here is a typical use of the for loop. We want to print the balance of our savings account over a period of years, as shown in this table:

Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

The for loop pattern applies because the variable year starts at 1 and then moves in constant increments until it reaches the target:

```
for year in range(1, numYears + 1) :
   Update balance.
   Print year and balance.
```

Following is the complete program. Figure 4 shows the corresponding flowchart.

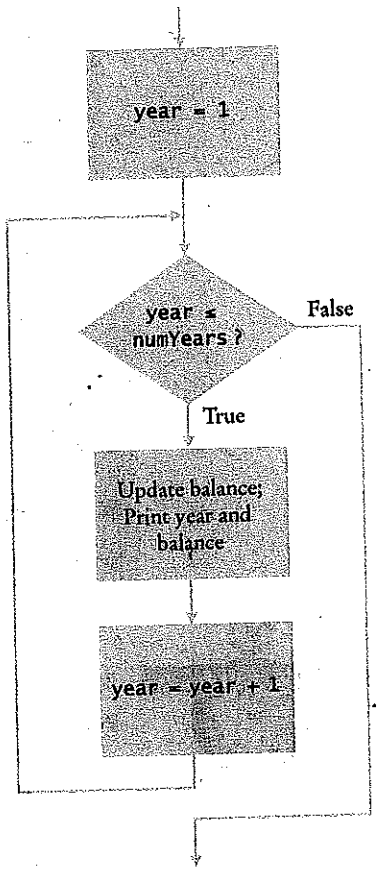


Figure 4 Flowchart of a for Loop

ch04/investment.py

```
1  ##
2  #  This program prints a table showing the growth of an investment.
3  #
4
5  # Define constant variables.
6  RATE = 5.0
7  INITIAL_BALANCE = 10000.0
8
9  # Obtain the number of years for the computation.
10 numYears = int(input("Enter number of years: "))
11
12 # Print the table of balances for each year.
13 balance = INITIAL_BALANCE
14 for year in range(1, numYears + 1) :
15    interest = balance * RATE / 100
16    balance = balance + interest
17    print("%4d %10.2f" % (year, balance))
```

Program Run

```
Enter number of years: 10
   1  10500.00
   2  11025.00
   3  11576.25
   4  12155.06
   5  12762.82
   6  13400.96
   7  14071.00
   8  14774.55
   9  15513.28
  10  16288.95
```

Table 2 for Loop Examples

Loop	Values of i	Comment
`for i in range(6) :`	0, 1, 2, 3, 4, 5	Note that the loop executes 6 times.
`for i in range(10, 16) :`	10, 11, 12, 13, 14 15	The ending value is never included in the sequence.
`for i in range(0, 9, 2) :`	0, 2, 4, 6, 8	The third argument is the step value.
`for i in range(5, 0, -1) :`	5, 4, 3, 2, 1	Use a negative step value to count down.

25. Write the `for` loop of the `investment.py` program as a `while` loop.

26. How many numbers does this loop print?

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

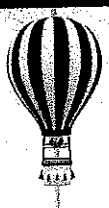
27. Write a `for` loop that prints all even numbers between 10 and 20 (inclusive).

28. Write a `for` loop that computes the total of the integers from 1 to `n`.

29. How would you modify the loop of the `investment.py` program to print all balances until the investment has doubled?

Practice It Now you can try these exercises at the end of the chapter: R4.18, R4.19, P4.8.

Programming Tip 4.1



Count Iterations

Finding the correct lower and upper bounds for a loop can be confusing. Should you start at 0 or at 1? Should you use `<= b` or `< b` as a termination condition?

Counting the number of iterations is a very useful device for better understanding a loop. Counting is easier for loops with asymmetric bounds. The loop

```
int i = a
while i < b :
   . . .
   i = i + 1
```

is executed `b - a` times. The same is true for the equivalent for loop

```
for i in range(a, b) :
```

These asymmetric bounds are particularly useful for traversing the characters in a string: The loop

```
for i in range(0, len(str)) :
   do something with i and str[i]
```

runs `len(str)` times, and `i` traverses all valid string positions from 0 to `len(str) - 1`. (Because these loops are so common, you can omit the 0 in the call to the `range` function.)

The loop with symmetric bounds,

```
int i = a
while i <= b :
   . . .
   i = i + 1
```

is executed `b - a + 1` times. That "+1" is the source of many programming errors. For example, when `a` is 10 and `b` is 20, then `i` assumes the values 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20. Those are eleven values: 20 - 10 + 1.

One way to visualize this "+1" error is by looking at a fence. Each section has one fence post to the left, and there is a final post on the right of the last section. Forgetting to count the last value is often called a "fence post error".

In a Python for loop, the "+1" can be quite noticeable:

```
for year in range(1, numYears + 1) :
```

You must specify an upper bound that is one more than the last value to be included in the range.

How many posts do you need for a fence with four sections? It is easy to be "off by one" with problems such as this one.

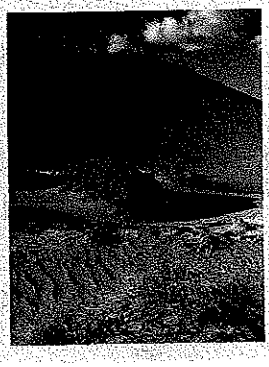
HOW TO 4.1 Writing a Loop

This How To walks you through the process of implementing a loop statement.

Problem Statement Read twelve temperature values (one for each month), and display the number of the month with the highest temperature. For example, according to `http://worldclimate.com`, the average maximum temperatures for Death Valley are (in order by month, in degrees Celsius):

18.2 22.6 26.4 31.1 36.6 42.2 45.7 44.5 40.2 33.1 24.2 17.6

In this case, the month with the highest temperature (45.7 degrees Celsius) is July, and the program should display 7.



Step 1 Decide what work must be done *inside* the loop.

Every loop needs to do some kind of repetitive work, such as

- Reading another item.
- Updating a value (such as a bank balance or total).
- Incrementing a counter.

If you can't figure out what needs to go inside the loop, start by writing down the steps that you would take if you solved the problem by hand. For example, with the maximum temperature problem, you might write

```
Read first value.
Read second value.
If second value is higher than the first, set highest temperature to that value, highest month to 2.
Read next value.
If value is higher than the first and second, set highest temperature to that value, highest month to 3.
Read next value.
If value is higher than the highest temperature seen so far, set highest temperature to that value,
   highest month to 4.
. . .
```

Now look at these steps and reduce them to a set of *uniform* actions that can be placed into the loop body. The first action is easy:

```
Read next value.
```

The next action is trickier. In our description, we used tests "higher than the first", "higher than the first and second", and "higher than the highest temperature seen so far". We need to settle on one test that works for all iterations. The last formulation is the most general.

Similarly, we must find a general way of setting the highest month. We need a variable that stores the current month, running from 1 to 12. Then we can formulate the second loop action:

```
If value is higher than the highest temperature, set highest temperature to that value,
   and set highest month to current month.
```

Altogether our loop is

```
Repeat
   Read next value.
   If value is higher than the highest temperature,
      set highest temperature to that value,
      set highest month to current month.
   Increment current month.
```

Step 2 Specify the loop condition.

What goal do you want to reach in your loop? Typical examples are

- Has a counter reached its final value?
- Have you read the last input value?
- Has a value reached a given threshold?

In our example, we simply want the current month to reach 12.

Step 3 Determine the loop type.

We distinguish between two major loop types. A *count-controlled* loop is executed a definite number of times. In an *event-controlled* loop, the number of iterations is not known in advance—the loop is executed until some event happens.

Count-controlled loops can be implemented as `for` statements. The `for` statement can either iterate over the individual elements of a container, such as a string, or be used with the `range` function to iterate over a sequence of integers.

Event-controlled loops are implemented as `while` statements in which the loop condition determines when the loop terminates. Sometimes, the condition for terminating a loop changes in the middle of the loop body. In that case, you can use a Boolean variable that specifies when you are ready to leave the loop; such a variable is called a flag. Follow this pattern:

```
done = False
while not done :
   Do some work.
   If all work has been completed :
      done = True
   else :
      Do more work.
```

In summary,

- If you need to iterate over all the elements of a container, without regard to their positions, use a plain `for` loop.
- If you need to iterate over a range of integers, use a `for` loop with the `range` function.
- Otherwise, use a `while` loop.

In our example, we read 12 temperature values. Therefore, we choose a `for` loop that uses the `range` function to iterate over a sequence of integers.

Step 4 Set up variables for entering the loop for the first time.

List all variables that are used and updated in the loop, and determine how to initialize them. Commonly, counters are initialized with 0 or 1, totals with 0.

In our example, the variables are

```
current month
highest value
highest month
```

We need to be careful how we set up the highest temperature value. We can't simply set it to 0. After all, our program needs to work with temperature values from Antarctica, all of which may be negative.

A good option is to set the highest temperature value to the first input value. Of course, then we need to remember to read in only 11 more values, with the current month starting at 2.

We also need to initialize the highest month with 1. After all, in an Australian city, we may never find a month that is warmer than January.

Step 5 Process the result after the loop has finished.

In many cases, the desired result is simply a variable that was updated in the loop body. For example, in our temperature program, the result is the highest month. Sometimes, the loop

computes values that contribute to the final result. For example, suppose you are asked to average the temperatures. Then the loop should compute the sum, not the average. After the loop has completed, you are ready to compute the average: divide the sum by the number of inputs.

Here is our complete loop:

```
Read first value; store as highest value.
highest month = 1
For current month from 2 to 12
   Read next value.
   If value is higher than the highest value
      Set highest value to that value.
      Set highest month to current month.
```

Step 6 Trace the loop with typical examples.

Hand trace your loop code, as described in Section 4.2. Choose example values that are not too complex—executing the loop 3–5 times is enough to check for the most common errors. Pay special attention when entering the loop for the first and last time.

Sometimes, you want to make a slight modification to make tracing feasible. For example, when hand-tracing the investment doubling problem, use an interest rate of 20 percent rather than 5 percent. When hand-tracing the temperature loop, use 4 data values, not 12.

Let's say the data are 22.6 36.6 44.5 24.2. Here is the trace:

current month	current value	highest month	highest value
		1	22.6
2	36.6	2	36.6
3	44.5	3	44.5
4	24.2		

The trace demonstrates that **highest month** and **highest value** are properly set.

Step 7 Implement the loop in Python.

Here's the loop for our example. Exercise P4.4 asks you to complete the program.

```
highestValue = int(input("Enter a value: "))
highestMonth = 1
for currentMonth in range(2, 13) :
   nextValue = int(input("Enter a value: "))
   if nextValue > highestValue :
      highestValue = nextValue
      highestMonth = currentMonth

print(highestMonth)
```

4.7 Nested Loops

In Section 3.3, you saw how to nest two `if` statements. Similarly, complex iterations sometimes require a **nested loop**: a loop inside another loop statement. When processing tables, nested loops occur naturally. An outer loop iterates over all rows of the table. An inner loop deals with the columns in the current row.

When the body of a loop contains another loop, the loops are nested. A typical use of nested loops is printing a table with rows and columns.

In this section you will see how to print a table. For simplicity, we will print the powers of x, x^n, as in the table at right.

Here is the pseudocode for printing the table:

```
Print table header.
For x from 1 to 10
   Print table row.
   Print new line.
```

x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
...	...	...	...
10	100	1000	10000

How do you print a table row? You need to print a value for each exponent. This requires a second loop.

```
For n from 1 to 4
   Print x^n.
```

This loop must be placed inside the preceding loop. We say that the inner loop is *nested* inside the outer loop.

There are 10 rows in the outer loop. For each x, the program prints four columns in the inner loop (see Figure 5). Thus, a total of $10 \times 4 = 40$ values are printed.

In this program, we want to show the results of multiple print statements on the same line. As shown in Special Topic 4.3, this is achieved by adding the argument `end=""` to the `print` function.

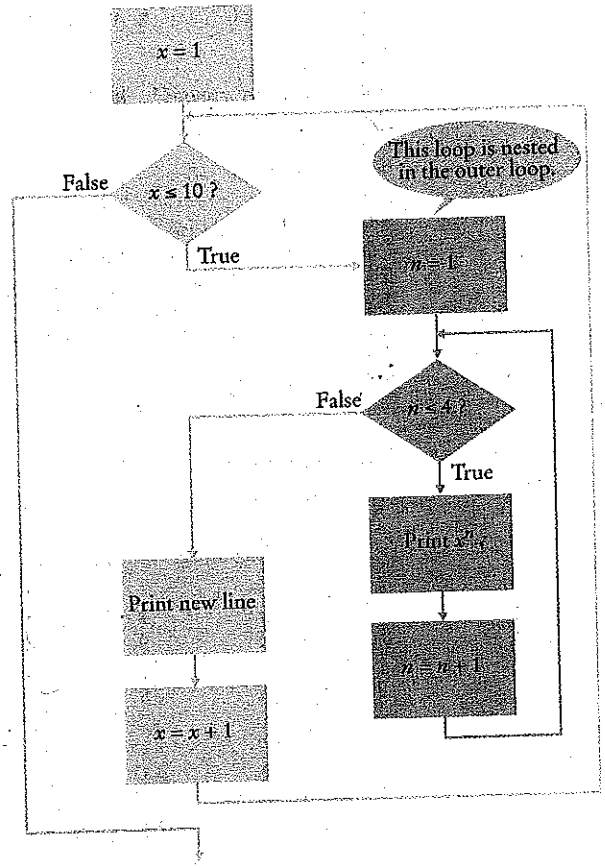
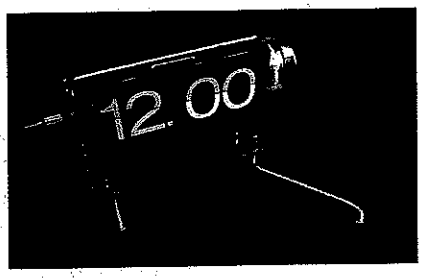


Figure 5
Flowchart of a Nested Loop

The hour and minute displays in a digital clock are an example of nested loops. The hours loop 12 times, and for each hour, the minutes loop 60 times.



Following is the complete program. Note that we also use loops to print the table header. However, those loops are not nested.

ch04/powertable.py

```
1  ##
2  #  This program prints a table of powers of x.
3  #
4
5  # Initialize constant variables for the max ranges.
6  NMAX = 4
7  XMAX = 10
8
9  # Print table header.
10 for n in range(1, NMAX + 1) :
11    print("%10d" % n, end="")
12
13 print()
14 for n in range(1, NMAX + 1) :
15    print("%10s" % "x ", end="")
16
17 print("\n", "    ", "-" * 35)
18
19 # Print table body.
20 for x in range(1, XMAX + 1) :
21    # Print the x row in the table.
22    for n in range(1, NMAX + 1) :
23       print("%10.0f" % x ** n, end="")
24
25    print()
```

Program Run

```
         1         2         3         4
         x         x         x         x
     -----------------------------------
         1         1         1         1
         2         4         8        16
         3         9        27        81
         4        16        64       256
         5        25       125       625
         6        36       216      1296
         7        49       343      2401
         8        64       512      4096
         9        81       729      6561
        10       100      1000     10000
```

30. Why is the newline suppressed (using `end=""`) in the inner loop but not in the outer loop?

31. How would you change the program to display all powers from x^0 to x^5?

32. If you make the change in Self Check 31, how many values are displayed?

33. What do the following nested loops display?

```
for i in range(3) :
   for j in range(1, 4) :
      print(i + j, end="")
   print()
```

34. Write nested loops that make the following pattern of brackets:

```
[][][][]
[][][][]
[][][][]
```

Practice It Now you can try these exercises at the end of the chapter: R4.26, P4.18, P4.22.

Table 3 Nested Loop Examples

Nested Loops	Output	Explanation

`for i in range(3) :`
`  for j in range(4) :`
`  print("*", end="")`
`  print()`
	`****`
	`****`
	`****`
		Prints 3 rows of 4 asterisks each.

`for i in range(4) :`
`  for j in range(3) :`
`  print("*", end="")`
`  print()`
	`***`
	`***`
	`***`
	`***`
		Prints 4 rows of 3 asterisks each.

`for i in range(4) :`
`  for j in range(i + 1) :`
`  print("*", end="")`
`  print()`
	`*`
	`**`
	`***`
	`****`
		Prints 4 rows of lengths 1, 2, 3, and 4.

`for i in range(3) :`
`  for j in range(5) :`
`  if j % 2 == 1 :`
`  print("*", end="")`
`  else :`
`  print("-", end="")`
`  print()`
	`-*-*-`
	`-*-*-`
	`-*-*-`
		Prints alternating dashes and asterisks.

`for i in range(3) :`
`  for j in range(5) :`
`  if i % 2 == j % 2 :`
`  print("*", end="")`
`  else :`
`  print(" ", end="")`
`  print()`
	`* * *`
	` * *`
	`* * *`
		Prints a checkerboard pattern.

Special Topic 4.3

Special Form of the `print` Function

Python provides a special form of the `print` function that prevents it from starting a new line after its arguments are displayed.

```
print(value1, value2, . . ., valuen, end="")
```

For example, the output of the two statements

```
print("00", end="")
print(3 + 4)
```

is the single line

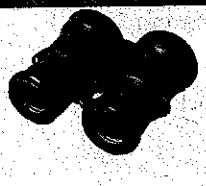
```
007
```

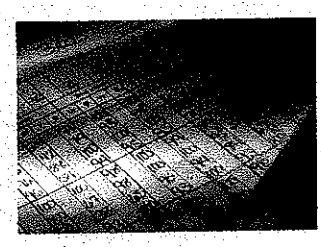
By including `end=""` as the last argument to the first `print` function, we indicate that an empty string is to be printed after the first argument is displayed instead of starting a new line. The output of the next `print` function starts on the same line where the previous one left off.

The `end=""` argument is called a *named argument*. Named arguments allow you to specify the contents of a specific optional argument defined for a function or method. Although named arguments can be used with many of Python's built-in functions and methods, we limit their use in this book to the `print` function.

WORKED EXAMPLE 4.1

Average Exam Grades





Problem Statement It is common to repeatedly read and process multiple groups of values. Write a program that can be used to compute the average exam grade for multiple students. Each student has the same number of exam grades.

Step 1 Understand the problem.

To compute the average exam grade for one student, we must enter and tally all of the grades for that student. This can be done with a loop. But we need to compute the average grade for multiple students. Thus, computing an individual student's average grade must be repeated for each student in the course. This requires a nested loop. The inner loop will process the grades for one student and the outer loop will repeat the process for each student.

```
Prompt user for the number of exams.
Repeat for each student
   Process the student's exam grades.
   Print the student's exam average.
```

Step 2 Compute the average grade for one student.

The algorithm from Section 4.5.1 can be used to extract the grades and compute the average. The difference in this problem, however, is that we can read a fixed number of grades for each student instead of reading until a sentinel value is entered. Because we know how many grades need to be read, we can use a `for` loop with the `range` function:

```
total score = 0
for i in range(1, numExams + 1) :
   Read the next exam score.
   Add the exam score to the total score.
Compute the exam average.
Print the exam average.
```

Step 3 Repeat the process for each student.

Because we are computing the average exam grade for multiple students, we must repeat the task in Step 2 for each student. Because we do not know how many students there are, we will use a `while` loop with a sentinel value. But what should the sentinel be? For simplicity, it can be based on a simple yes or no question. After the user enters the grades for a student, we can prompt the user whether they wish to enter grades for another student:

```
moreGrades = input("Enter exam grades for another student (Y/N)? ")
moreGrades = moreGrades.upper()
```

A no response serves as the terminating condition. Thus, each time the user enters "Y" at the prompt, the loop will be executed again.

We will use a loop condition set to `moreGrades == "Y"`, and initialize the loop variable to contain the string "Y". This allows the loop to be executed at least once so the user can enter the grades for the first student before being prompted for a yes or no response.

```
moreGrades = "Y"
while moreGrades == "Y" :
   Enter grades for one student.
   Compute average grade for one student.
   moreGrades = input("Enter exam grades for another student (Y/N)? ")
   moreGrades = moreGrades.upper()
```

Step 4 Implement your solution in Python.

Here is the complete program:

ch04/examaverages.py

```
1  ##
2  # This program computes the average exam grade for multiple students.
3  #
4
5  # Obtain the number of exam grades per student.
6  numExams = int(input("How many exam grades does each student have? "))
7
8  # Initialize moreGrades to a non-sentinel value.
9  moreGrades = "Y"
10
11 # Compute average exam grades until the user wants to stop.
12 while moreGrades == "Y" :
13
14    # Compute the average grade for one student.
15    print("Enter the exam grades.")
16    total = 0
17    for i in range(1, numExams + 1) :
18       score = int(input("Exam %d: " % i))   # Prompt for each exam grade.
19       total = total + score
20
21    average = total / numExams
22    print("The average is %.2f" % average)
23
24    # Prompt as to whether the user wants to enter grades for another student.
25    moreGrades = input("Enter exam grades for another student (Y/N)? ")
26    moreGrades = moreGrades.upper()
```

4.8 Processing Strings

A common use of loops is to process or evaluate strings. For example, you may need to count the number of occurrences of one or more characters in a string or verify that the contents of a string meet certain criteria. In this section, we explore several basic string processing algorithms.

4.8.1 Counting Matches

In Section 4.5.2, we saw how to count the number of values that fulfill a particular condition. We can also apply this task to strings. For example, suppose you need to count the number of uppercase letters contained in a string.

```
uppercase = 0
for char in string :
   if char.isupper() :
      uppercase = uppercase + 1
```

This loop iterates through the characters in the string and checks each one to see if it is an uppercase letter. When an uppercase letter is found, the `uppercase` counter is incremented. For example, if `string` contains `"My Fair Lady"`, `uppercase` is incremented three times (when `char` is `M`, `F`, and `L`).

Use the `in` operator to compare a character against multiple options.

Sometimes, you need to count the number of occurrences of multiple characters within a string. For example, suppose we would like to know how many vowels are contained in a word. Instead of individually comparing each letter in the word against the five vowels, you can use the `in` operator and a literal string that contains the five letters:

```
vowels = 0
for char in word :
   if char.lower() in "aeiou" :
      vowels = vowels + 1
```

Note the use of the `lower` method in the logical expression. This method is used to convert each uppercase letter to its corresponding lowercase letter before checking to see if it is a vowel. That way, we limit the number of characters that must be specified in the literal string.

4.8.2 Finding All Matches

When you need to examine every character within a string, independent of its position, you can use the `for` statement to iterate over the individual characters. This was the approach used in the previous section to count the number of uppercase letters in a string. Sometimes, however, you may need to find the position of each match within a string. For example, suppose you are asked to print the position of each uppercase letter in a sentence. You cannot use the `for` statement that iterates over all characters because you need to know the positions of the matches. Instead, iterate over the positions (using `for` with `range`) and look up the character at each position:

```
sentence = input("Enter a sentence: ")
for i in range(len(sentence)) :
   if sentence[i].isupper() :
      print(i)
```

4.8.3 Finding the First or Last Match

If your goal is to find a match, exit the loop when the match is found.

When you count the values that fulfill a condition, you need to look at all values. However, if your task is to find a match, then you can stop as soon as the condition is fulfilled.

Here is a loop that finds the position of the first digit in a string.

```
found = False
position = 0
while not found and position < len(string) :
   if string[position].isdigit() :
      found = True
   else :
      position = position + 1

if found :
   print("First digit occurs at position", position)
else :
   print("The string does not contain a digit.")
```

If a match was found, then `found` will be `True` and `position` will contain the index of the first match. If the loop did not find a match, then `found` remains `False` after the loop terminates. We can use the value of `found` to determine which of the two messages to print.

What if we need to find the position of the last digit in the string? Traverse the string from back to front:

```
found = False
position = len(string) - 1
while not found and position >= 0 :
   if string[position].isdigit() :
      found = True
   else :
      position = position - 1
```

When searching, you look at items until a match is found.

4.8.4 Validating a String

Validating a string can ensure it contains correctly formatted data.

In Chapter 3, you learned the importance of validating user input before it is used in computations. But data validation is not limited to verifying that user input is a specific value or falls within a valid range. It is also common to require user input to be entered in a specific format. For example, consider the task of verifying if a string contains a correctly formatted telephone number.

In the United States, telephone numbers consist of three parts—area code, exchange, and line number—which are commonly specified in the form (###)###-####. We can examine a string to ensure that it contains a correctly formatted phone number. To do this, we must not only verify that it contains digits and the appropriate symbols, but that each are in the appropriate spots in the string. This requires an event-controlled loop that can exit early if an invalid character or an out of place symbol is encountered while processing the string:

```
valid = True
position = 0
while valid and position < len(string) :
   if position == and string[position] != "(" :
      valid = False
```

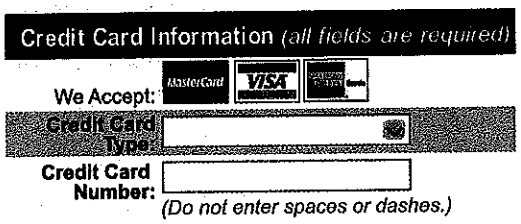
```
   elif position == 4 and string[position] != ")" :
      valid = False
   elif position == 8 and string[position] != "-" :
      valid = False
   elif not string[position].isdigit() :
      valid = False
   else :
      position = position + 1

if valid :
   print("The string contains a valid phone number.")
else :
   print("The string does not contain a valid phone number.")
```

As an alternative, we can combine the four logical conditions into a single expression to produce a more compact loop:

```
valid = True
position = 0
while valid and position < len(string) :
   if ((position == 0 and string[position] != "(")
       or (position == 4 and string[position] != ")")
       or (position == 8 and string[position] != "-")
       or not string[position].isdigit() :
      valid = False
   else :
      position = position + 1
```

4.8.5 Building a New String



One of the minor annoyances of online shopping is that many web sites require you to enter a credit card without spaces or dashes, which makes double-checking the number rather tedious. How hard can it be to remove dashes or spaces from a string?

You build a string by concatenating individual characters.

As you learned in Chapter 2, the contents of a string cannot be changed. But nothing prevents us from building a new string. For example, if the user enters a string that contains a credit card number in the format "4123-5678-9012-3450", we can remove the dashes by building a new string that only contains the digits: start with an empty string and append to it each character in the original string that is not a space or dash. In Python, characters can be appended to a string using the string concatenation operator:

```
newString = newString + "x"
```

Here is a loop that builds a new string containing a credit card number with spaces and dashes removed:

```
userInput = input("Enter a credit card number: ")
creditCardNumber = ""
for char in userInput :
   if char != " " and char != "-" :
      creditCardNumber = creditCardNumber + char
```

If the user enters "4123-5678-9012-3450", creditCardNumber will contain the string "4123567890123450" after the loop executes.

As another example, suppose we need to build a new string in which all uppercase letters in the original are converted to lowercase and all lowercase letters are converted to uppercase. Using the same technique of string concatenation used in the previous example, this is rather easy:

```
newString = ""
for char in original :
   if char.isupper() :
      newChar = char.tolower()
   elif char.islower() :
      newChar = char.toupper()
   else :
      newChar = char
   newString = newString + newChar
```

The following program demonstrates several of the string processing algorithms presented in this section. This program reads a string that contains a test taker's answers to a multiple choice exam and grades the test.

ch04/multiplechoice.py

```
1  ##
2  #  This program grades a multiple choice exam in which each question has four
3  #  possible choices: a, b, c, or d.
4  #
5
6  # Define a string containing the correct answers.
7  CORRECT_ANSWERS = "adbdcacbdac"
8
9  # Obtain the user's answers, and make sure enough answers are provided.
10 done = False
11 while not done :
12    userAnswers = input("Enter your exam answers: ")
13    if len(userAnswers) == len(CORRECT_ANSWERS) :
14       done = True
15    else :
16       print("Error: an incorrect number of answers given.")
17
18 # Check the exam.
19 numQuestions = len(CORRECT_ANSWERS)
20 numCorrect = 0
21 results = ""
22
23 for i in range(numQuestions) :
24    if userAnswers[i] == CORRECT_ANSWERS[i] :
25       numCorrect = numCorrect + 1
26       results = results + userAnswers[i]
27    else :
28       results = results + "X"
29
30 # Grade the exam.
31 score = round(numCorrect / numQuestions * 100)
32
33 if score == 100 :
34    print("Very Good!")
35 else :
36    print("You missed %d questions: %s" % (numQuestions - numCorrect, results))
37
38 print("Your score is: %d percent" % score)
```

Program Run

```
Enter your exam answers: acddcbcbcac
You missed 4 questions: aXXdcXcbXac
Your score is: 64 percent
```

35. How do you find the position of the second uppercase letter in a string?
36. How do you print the symbol and position of all punctuation symbols (`.?!,;:'`) contained in a string?
37. What changes are needed in the code from Section 4.7.3 if the format of the telephone number requires a space following the right parenthesis?
38. Design a loop that examines a string to verify that it contains a sequence of alternating `"x"` and `"o"` characters.
39. How do you verify that a string contains a valid integer value?

Practice It Now you can try these exercises at the end of the chapter: P4.3, P4.10.

4.9 Application: Random Numbers and Simulations

In a simulation, you use the computer to simulate an activity.

A *simulation program* uses the computer to simulate an activity in the real world (or an imaginary one). Simulations are commonly used for predicting climate change, analyzing traffic, picking stocks, and many other applications in science and business. In many simulations, one or more loops are used to modify the state of a system and observe the changes. You will see examples in the following sections.

4.9.1 Generating Random Numbers

Many events in the real world are difficult to predict with absolute precision, yet we can sometimes know the average behavior quite well. For example, a store may know from experience that a customer arrives every five minutes. Of course, that is an average—customers don't arrive in five minute intervals. To accurately model customer traffic, you want to take that random fluctuation into account. Now, how can you run such a simulation in the computer?

You can introduce randomness by calling the random number generator.

The Python library has a *random number generator* that produces numbers that appear to be completely random. Calling `random()` yields a random floating-point number that is ≥ 0 and < 1. Call `random()` again, and you get a different number. The `random` function is defined in the `random` module.

The following program calls `random()` ten times.

ch04/randomtest.py

```
1  ##
2  #  This program prints ten random numbers between 0 and 1.
3  #
4
```

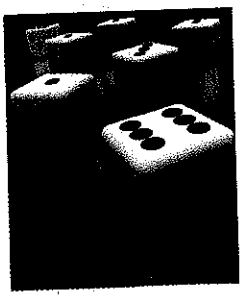
```
5  from random import random
6
7  for i in range(10) :
8     value = random()
9     print(value)
```

Program Run

```
0.580742512361
0.907222103296
0.102851584902
0.196652864583
0.957267274444
0.439101769744
0.299604096229
0.679313379668
0.0903726139666
0.801120553331
```

Actually, the numbers are not completely random. They are drawn from sequences of numbers that don't repeat for a long time. These sequences are actually computed from fairly simple formulas; they just behave like random numbers (see Exercise P4.26). For that reason, they are often called **pseudorandom** numbers.

4.9.2 Simulating Die Tosses



In actual applications, you need to transform the output from the random number generator into a specific range. For example, to simulate the throw of a die, you need random integers between 1 and 6.

Python provides a separate function for generating a random integer within a given range. The function

```
randint(a, b)
```

which is defined in the `random` module, returns a random integer that is between `a` and `b`, including the bounds themselves.

Here is a program that simulates the throw of a pair of dice:

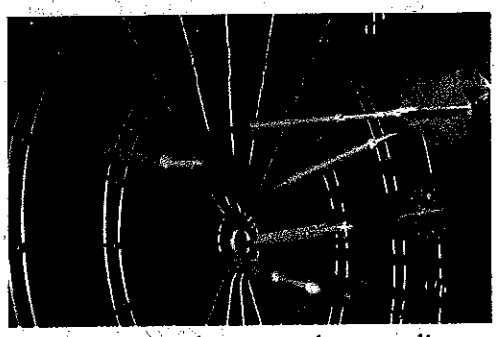
ch04/dice.py

```
1  ##
2  #  This program simulates tosses of a pair of dice.
3  #
4
5  from random import randint
6
7  for i in range(10) :
8     # Generate two random numbers between 1 and 6, inclusive.
9     d1 = randint(1, 6)
10    d2 = randint(1, 6)
11
12    # Print the two values.
13    print(d1, d2)
```

Program Run

```
1 5
6 4
1 1
4 5
6 4
3 2
4 2
3 5
5 2
4 5
```

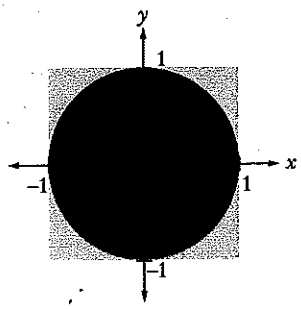
4.9.3 The Monte Carlo Method



The Monte Carlo method is an ingenious method for finding approximate solutions to problems that cannot be precisely solved. (The method is named after the famous casino in Monte Carlo.) Here is a typical example. It is difficult to compute the number π, but you can approximate it quite well with the following simulation.

Simulate shooting a dart into a square surrounding a circle of radius 1. That is easy: generate random x- and y-coordinates between –1 and 1.

If the generated point lies inside the circle, we count it as a *hit*. That is the case when $x^2 + y^2 \leq 1$. Because our shots are entirely random, we expect that the ratio of *hits / tries* is approximately equal to the ratio of the areas of the circle and the square, that is, $\pi / 4$. Therefore, our estimate for π is $4 \times$ *hits / tries*. This method yields an estimate for π, using nothing but simple arithmetic.



To generate a random floating-point value between –1 and 1, you compute:

```
r = random()   # 0 ≤ r < 1
x = -1 + 2 * r   # -1 ≤ x < 1
```

As r ranges from 0 (inclusive) to 1 (exclusive), x ranges from $-1 + 2 \times 0 = -1$ (inclusive) to $-1 + 2 \times 1 = 1$ (exclusive). In our application, it does not matter that x never reaches 1. The points that fulfill the equation $x = 1$ lie on a line with area 0.

Here is the program that carries out the simulation:

ch04/montecarlo.py

```
1  ##
2  #  This program computes an estimate of pi by simulating dart throws onto a square.
3  #
4
5  from random import random
6
7  TRIES = 10000
8
```

```
hits = 0
for i in range(TRIES) :

   # Generate two random numbers between –1 and 1
   r = random()
   x = -1 + 2 * r
   r = random()
   y = -1 + 2 * r

   # Check whether the point lies in the unit circle
   if x * x + y * y <= 1 :
      hits = hits + 1

# The ratio hits / tries is approximately the same as the ratio
# circle area / square area = pi / 4.

piEstimate = 4.0 * hits / TRIES
print("Estimate for pi:", piEstimate)
```

Program Run

```
Estimate for pi: 3.1464
```

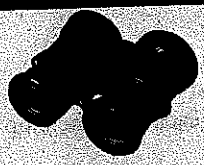
40. How do you simulate a coin toss with the `random` method?
41. How do you simulate the picking of a random playing card?
42. Why does the loop body in `dice.py` call `randint(1, 6)` twice?
43. In many games, you throw a pair of dice to get a value between 2 and 12. What is wrong with this simulated throw of a pair of dice?

    ```
    sum = randint(2, 12)
    ```

44. How do you generate a random floating-point number ≥ 0 and < 100?

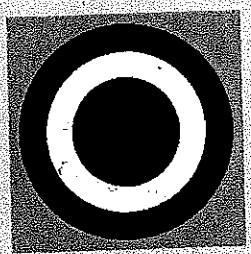
Practice It Now you can try these exercises at the end of the chapter: R4.27, P4.7, P4.25.

WORKED EXAMPLE 4.2 Bull's Eye



Problem Statement Develop a graphics program that draws a target with alternating black and white rings on a light gray background and a red bull's eye in the center.

The number of rings in the target should be obtained from the user but it must be between 2 and 10. Each ring should be 25 pixels wide and the bull's eye should have a diameter that is twice the width of the rings. The outermost ring must be colored black with each subsequent ring alternating between white and black. Finally, the size of the graphics window should be based on the size of the target with the outer ring offset 10 pixels from all four sides of the window.



Step 1 Define constant variables.

You should define constant variables for the constraints and sizes specified in the problem statement. This also makes it easy to change these constraints, if necessary.

The problem description specifies several magic numbers:

```
MIN_NUM_RINGS = 2
MAX_NUM_RINGS = 10
RING_WIDTH = 25
TARGET_OFFSET = 10
```

Step 2 Obtain the number of rings from the user.

Because there is a limitation on the number of rings contained in the target, we need to validate the user input.

```
numRings = int(input("Enter # of rings in the target: "))
While number of rings is outside the valid range
   Print an error message.
   numRings = int(input("Re-enter # of rings in the target: "))
```

Step 3 Determine how the rings will be drawn.

Each ring can be drawn as a filled circle, with the individual circles drawn on top of each other. The inner circles will fill the center part of the larger circles, thus creating the ring effect.

Step 4 Determine the size of the graphics window.

The size of the window is based on the size of the target, which is the size of the outer ring. To determine the radius of the outer ring we can sum the widths of all the rings and the radius of the bull's eye (which is equal to the width of a ring). We know the number of rings and the width of each ring, so the computation is

```
outer ring radius = (number of rings + 1) x ring width
```

The size of the target is simply the diameter of the outer ring, or 2 times its radius:

```
target size = 2 x outer ring radius
```

Finally, the target is offset from the window border by `TARGET_OFFSET` pixels. Accounting for the offset and the size of the target, we can compute the size of the window as

```
window size = target size + 2 x TARGET_OFFSET
```

Step 5 Draw the rings of the target.

To draw the rings of the target, we start with the outermost circle and work our way inward. We can use a basic `for` loop that iterates once for each ring and includes several steps:

```
Initialize circle parameters.
for i in range(numRings) :
   Select circle color.
   Draw the circle.
   Adjust circle parameters.
```

To select the color used to draw the circle, we can base our decision on the value of the loop variable `i`. Because the loop variable starts at `0`, a black circle will be drawn each time the loop variable is even and a white circle will be drawn each time it's odd.

```
If i is even
   canvas.setColor("black")
Else
   canvas.setColor("white")
```

To draw the circle, use the `drawOval` canvas method with both the width and height of the bounding box set to the diameter of the circle. The `drawOval` method also requires the position of the upper-left corner of the bounding box:

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

The diameter of each inner circle will decrease by 2 times the ring width and the position of the bounding box will move inward by a ring width in both directions.

```
diameter = diameter - 2 * RING_WIDTH
x = x + RING_WIDTH
y = y + RING_WIDTH
```

Finally, the parameters of the outer circle, which is drawn first, must be initialized before the first iteration of the loop. The diameter of the outer circle is equal to the size of the target. Its bounding box is offset from the window border by TARGET_OFFSET pixels in both the horizontal and vertical directions.

```
diameter = target size
x = TARGET_OFFSET
y = TARGET_OFFSET
```

Step 6 Draw the bull's eye in the center.

After drawing the black and white rings, we still have to draw the bull's eye in the center as a red filled circle. When the loop terminates, the circle parameters (position and diameter) will be set to the values needed to draw that circle.

Step 7 Implement your solution in Python.

The complete program is provided below. Note that we use the setBackground canvas method to set the background color of the canvas to a light gray instead of the default white. (See Appendix D for a complete description of the graphics module.)

ch04/bullseye.py

```
##
#  Draws a target with a bull's eye using the number of rings specified by the user.
#

from graphics import GraphicsWindow

# Define constant variables.
MIN_NUM_RINGS = 2
MAX_NUM_RINGS = 10
RING_WIDTH = 25
TARGET_OFFSET = 10

# Obtain number of rings in the target.
numRings = int(input("Enter # of rings in the target: "))
while numRings < MIN_NUM_RINGS or numRings > MAX_NUM_RINGS :
   print("Error: the number of rings must be between",
         MIN_NUM_RINGS, "and", MAX_NUM_RINGS)
   numRings = int(input("Re-enter # of rings in the target: "))

# Determine the diameter of the outermost circle. It has to be drawn first.
diameter = (numRings + 1) * RING_WIDTH * 2

# Determine the size of the window based on the size of the outer circle.
winSize = diameter + 2 * TARGET_OFFSET

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

# Use a light gray background for the canvas.
canvas.setBackground("light gray")

# Draw the rings, alternating between black and white.
x = TARGET_OFFSET
```

```
35  y = TARGET_OFFSET
36  for ring in range(numRings) :
37     if ring % 2 == 0 :
38        canvas.setColor("black")
39     else :
40        canvas.setColor("white")
41     canvas.drawOval(x, y, diameter, diameter)
42
43     diameter = diameter - 2 * RING_WIDTH
44     x = x + RING_WIDTH
45     y = y + RING_WIDTH
46
47  # Draw the bull's eye in red.
48  canvas.setColor("red")
49  canvas.drawOval(x, y, diameter, diameter)
50
51  win.wait()
```

Computing & Society 4.2 Software Piracy

As you read this, you will have written a few computer programs and experienced firsthand how much effort it takes to write even the humblest of programs. Writing a real software product, such as a financial application or a computer game, takes a lot of time and money. Few people, and fewer companies, are going to spend that kind of time and money if they don't have a reasonable chance to make more money from their effort. (Actually, some companies give away their software in the hope that users will upgrade to more elaborate paid versions. Other companies give away the software that enables users to read and use files but sell the software needed to create those files. Finally, there are individuals who donate their time, out of enthusiasm, and produce programs that you can copy freely.)

When selling software, a company must rely on the honesty of its customers. It is an easy matter for an unscrupulous person to make copies of computer programs without paying for them. In most countries that is illegal. Most governments provide legal protection, such as copyright laws and patents, to encourage the development of new products. Countries that tolerate widespread piracy have found that they have an ample cheap supply of foreign software, but no local manufacturers willing to design good software for their own citizens, such as word processors in the local script or financial programs adapted to the local tax laws.

When a mass market for software first appeared, vendors were enraged by the money they lost through piracy. They tried to fight back by various schemes to ensure that only the legitimate owner could use the software, such as *dongles*—devices that must be attached to a printer port before the software will run. Legitimate users hated these measures. They paid for the software, but they had to suffer through inconveniences, such as having multiple dongles sticking out from their computer. In the United States, market pressures forced most vendors to give up on these copy protection schemes, but they are still commonplace in other parts of the world.

Because it is so easy and inexpensive to pirate software, and the chance of being found out is minimal, you have to make a moral choice for yourself. If a package that you would really like to have is too expensive for your budget, do you steal it, or do you stay honest and get by with a more affordable product?

Of course, piracy is not limited to software. The same issues arise for other digital products as well. You may have had the opportunity to obtain copies of songs or movies without payment. Or you may have been frustrated by a copy protection device on your music player that made it difficult for you to listen to songs that you paid for. Admittedly, it can be difficult to have a lot of sympathy for a musical ensemble whose publisher charges a lot of money for what seems to have been very little effort on their part, at least when compared to the effort that goes into designing and implementing a software package. Nevertheless, it seems only fair that artists and authors receive some compensation for their efforts. How to pay artists, authors, and programmers fairly, without burdening honest customers, is an unsolved problem at the time of this writing, and many computer scientists are engaged in research in this area.

CHAPTER SUMMARY

Explain the flow of execution in a loop.

- A `while` loop executes instructions repeatedly while a condition is true.
- An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.

Use the technique of hand-tracing to analyze the behavior of a program.

- Hand-tracing is a simulation of code execution in which you step through instructions and track the values of the variables.
- Hand-tracing can help you understand how an unfamiliar algorithm works.
- Hand-tracing can show errors in code or pseudocode.

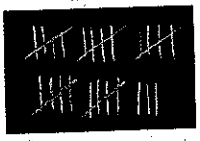
Implement loops that read sequences of input data.

- A sentinel value denotes the end of a data set, but it is not part of the data.
- A pair of input operations, known as the priming and modification reads, can be used to read a sentinel-terminated sequence of values.
- Use input redirection to read input from a file. Use output redirection to capture program output in a file.

Use the technique of storyboarding for planning user interactions.

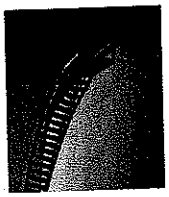
- A storyboard consists of annotated sketches for each step in an action sequence.
- Developing a storyboard helps you understand the inputs and outputs that are required for a program.

Know the most common loop algorithms.



- To compute an average, keep a total and a count of all values.
- To count values that fulfill a condition, check all values and increment a counter for each match.
- To find the largest value, update the largest value seen so far whenever you see a larger one.
- To compare adjacent inputs, store the preceding input in a variable.

Use `for` loops for implementing count-controlled loops.



- The `for` loop is used to iterate over the elements of a container.

Use nested loops to implement multiple levels of iteration.

- When the body of a loop contains another loop, the loops are nested. A typical use of nested loops is printing a table with rows and columns.

Use loops to process strings.

- Use the `in` operator to compare a character against multiple options.
- If your goal is to find a match, exit the loop when the match is found.
- Validating a string can ensure it contains correctly formatted data.
- You build a string by concatenating individual characters.

Apply loops to the implementation of simulations.

- In a simulation, you use the computer to simulate an activity.
- You can introduce randomness by calling the random number generator.

REVIEW QUESTIONS

■ **R4.1** Write a `while` loop that prints

a. All squares less than `n`. For example, if `n` is 100, print 0 1 4 9 16 25 36 49 64 81.

b. All positive numbers that are divisible by 10 and less than `n`. For example, if `n` is 100, print 10 20 30 40 50 60 70 80 90

c. All powers of two less than `n`. For example, if `n` is 100, print 1 2 4 8 16 32 64.

■■ **R4.2** Write a loop that computes

a. The sum of all even numbers between 2 and 100 (inclusive).

b. The sum of all squares between 1 and 100 (inclusive).

c. The sum of all odd numbers between `a` and `b` (inclusive).

d. The sum of all odd digits of `n`. (For example, if `n` is 32677, the sum would be 3 + 7 + 7 = 17.)

■ **R4.3** Provide trace tables for these loops.

a.
```
i = 0
j = 10
n = 0
while i < j :
   i = i + 1
   j = j - 1
   n = n + 1
```

b.
```
i = 0
j = 0
n = 0
while i < 10 :
   i = i + 1
   n = n + i + j
   j = j + 1
```

c.
```
i = 10
j = 0
n = 0
while i > 0 :
   i = i - 1
   j = j + 1
   n = n + i - j
```

d.
```
i = 0
j = 10
n = 0
while i != j :
   i = i + 2
   j = j - 2
   n = n + 1
```

▪ **R4.4** What do these loops print?

a.
```
for i in range(1, 10) :
   print(i)
```

b.
```
for i in range(1, 10, 2) :
   print(i)
```

c.
```
for i in range(10, 1, -1) :
   print(i)
```

d.
```
for i in range(10) :
   print(i)
```

e.
```
for i in range(1, 10) :
   if i % 2 == 0 :
      print(i)
```

▪ **R4.5** What is an infinite loop? On your computer, how can you terminate a program that executes an infinite loop?

▪ **R4.6** Write a program trace for the pseudocode in Exercise P4.6, assuming the input values are 4 7 –2 –5 0.

▪▪ **R4.7** What is an "off-by-one" error? Give an example from your own programming experience.

▪ **R4.8** What is a sentinel value? Give a simple rule when it is appropriate to use a numeric sentinel value.

▪ **R4.9** Which loop statements does Python support? Give simple rules for when to use each loop type.

▪ **R4.10** How many iterations do the following loops carry out?

a. `for i in range(1, 11) . . .`

b. `for i in range(10) . . .`

c. `for i in range(10, 0, -1) . . .`

d. `for i in range(-10, 11) . . .`

e. `for i in range(10, 0) . . .`

f. `for i in range(-10, 11, 2) . . .`

g. `for i in range(-10, 11, 3) . . .`

▪▪ **R4.11** Give an example of a for loop where symmetric bounds are more natural. Give an example of a for loop where asymmetric bounds are more natural.

■■ **R4.12** Write pseudocode for a program that prints a calendar such as the following:

```
Su  M  T  W Th  F Sa
            1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

■ **R4.13** Write pseudocode for a program that prints a Celsius/Fahrenheit conversion table such as the following:

```
Celsius | Fahrenheit
--------+-----------
      0 |          32
     10 |          50
     20 |          68
  . . .        . . .
    100 |         212
```

■ **R4.14** Write pseudocode for a program that reads a student record, consisting of the student's first and last name, followed by a sequence of test scores and a sentinel of –1. The program should print the student's average score. Then provide a trace table for this sample input:

```
Harry
Morgan
94
71
86
95
-1
```

■■ **R4.15** Write pseudocode for a program that reads a sequence of student records and prints the total score for each student. Each record has the student's first and last name, followed by a sequence of test scores and a sentinel of –1. The sequence is terminated by the word `END`. Here is a sample sequence:

```
Harry
Morgan
94
71
86
95
-1
Sally
Lin
99
98
100
95
90
-1
END
```

Provide a trace table for this sample input.

■ **R4.16** Rewrite the following `for` loop as a `while` loop.

```
s = 0
for i in range(1, 10) :
   s = s + i
```

■ **R4.17** Provide trace tables of the following loops.

a.
```
s = 1
n = 1
while s < 10 :
   s = s + n
```

b.
```
s = 1
for n in range(1, 5) :
   s = s + n
```

■ **R4.18** What do the following loops print? Work out the answer by tracing the code, not by using the computer.

a.
```
s = 1
for n in range(1, 6) :
   s = s + n
   print(s)
```

b.
```
s = 1
for n in range(1, 11) :
   n = n + 2
   s = s + n
   print(s)
```

c.
```
s = 1
for n in range(1, 6) :
   s = s + n
   n = n + 1
print(s, n)
```

■ **R4.19** What do the following program segments print? Find the answers by tracing the code, not by using the computer.

a.
```
n = 1
for i in range(2, 5) :
   n = n + i
print(n)
```

b.
```
n = 1 / 2
i = 2
while i < 6 :
   n = n + 1 / i
   i = i + 1
print(i)
```

c.
```
x = 1.0
y = 1.0
i = 0
while y >= 1.5 :
   x = x / 2
   y = x + y
   i = i + 1
print(i)
```

■ **R4.20** Add a storyboard panel for the conversion program in Section 4.4 on page 170 that shows a scenario where a user enters incompatible units.

■ **R4.21** In Section 4.4, we decided to show users a list of all valid units in the prompt. If the program supports many more units, this approach is unworkable. Give a storyboard panel that illustrates an alternate approach: If the user enters an unknown unit, a list of all known units is shown.

■ **R4.22** Change the storyboards in Section 4.4 to support a menu that asks users whether they want to convert units, see program help, or quit the program. The menu should be displayed at the beginning of the program, when a sequence of values has been converted, and when an error is displayed.

■ **R4.23** Draw a flow chart for a program that carries out unit conversions as described in Section 4.4.

■■ **R4.24** In Section 4.5.4, the code for finding the largest and smallest input initializes the `largest` and `smallest` variables with an input value. Why can't you initialize them with zero?

■ **R4.25** What are nested loops? Give an example where a nested loop is typically used.

■■ **R4.26** The nested loops

```
for i in range(height) :
   for j in range(width) :
      print("*", end="")
   print()
```

display a rectangle of a given width and height, such as

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

Write a *single* `for` loop that displays the same rectangle.

■■ **R4.27** Suppose you design an educational game to teach children how to read a clock. How do you generate random values for the hours and minutes?

■■■ **R4.28** In a travel simulation, Harry will visit one of his 15 friends who are located in three states. He has ten friends in California, three in Nevada, and two in Utah. How do you produce a random number between 1 and 3, denoting the destination state, with a probability that is proportional to the number of friends in each state?

PROGRAMMING EXERCISES

■ **P4.1** Write programs with loops that compute

a. The sum of all even numbers between 2 and 100 (inclusive).

b. The sum of all squares between 1 and 100 (inclusive).

c. All powers of 2 from 2^0 up to 2^{20}.

d. The sum of all odd numbers between `a` and `b` (inclusive), where `a` and `b` are inputs.

e. The sum of all odd digits of an input. (For example, if the input is 32677, the sum would be 3 + 7 + 7 = 17.)

■■ **P4.2** Write programs that read a sequence of integer inputs and print

a. The smallest and largest of the inputs.

b. The number of even and odd inputs.

c. Cumulative totals. For example, if the input is 1 7 2 9, the program should print 1 8 10 19.

d. All adjacent duplicates. For example, if the input is 1 3 3 4 5 5 6 6 6 2, the program should print 3 5 6.

■■ **P4.3** Write programs that read a line of input as a string and print

a. Only the uppercase letters in the string.

b. Every second letter of the string.

c. The string, with all vowels replaced by an underscore.

d. The number of digits in the string.

e. The positions of all vowels in the string.

■■ **P4.4** Complete the program in How To 4.1 on page 182. Your program should read twelve temperature values and print the month with the highest temperature.

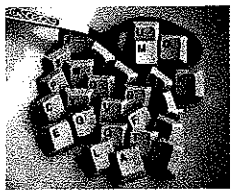
■■ **P4.5** Write a program that reads a set of floating-point values. Ask the user to enter the values, then print

- the average of the values.
- the smallest of the values.
- the largest of the values.
- the range, that is the difference between the smallest and largest.

■ **P4.6** Translate the following pseudocode for finding the minimum value from a set of inputs into a Python program.

```
Set a Boolean variable "first" to true.
While another value has been read successfully
   If first is true
      Set the minimum to the value.
      Set first to false.
   Else if the value is less than the minimum
      Set the minimum to the value.
Print the minimum.
```

■■■ **P4.7** Translate the following pseudocode for randomly permuting the characters in a string into a Python program.



```
Read a word.
Repeat len(word) times
   Pick a random position i in the word, but not the last position.
   Pick a random position j > i in the word.
   Swap the letters at positions j and i.
Print the word.
```

To swap the letters, construct substrings as follows:

Then replace the string with

```
first + word[j] + middle + word[i] + last
```

■ **P4.8** Write a program that reads a word and prints each character of the word on a separate line. For example, if the user provides the input `"Harry"`, the program prints

```
H
a
r
r
y
```

■■ **P4.9** Write a program that reads a word and prints the word in reverse. For example, if the user provides the input `"Harry"`, the program prints

```
yrraH
```

■ **P4.10** Write a program that reads a word and prints the number of vowels in the word. For this exercise, assume that `a e i o u y` are vowels. For example, if the user provides the input `"Harry"`, the program prints `2 vowels`.

■■■ **P4.11** Write a program that reads a word and prints the number of syllables in the word. For this exercise, assume that syllables are determined as follows: Each sequence of adjacent vowels `a e i o u y`, except for the last `e` in a word, is a syllable. However, if that algorithm yields a count of 0, change it to 1. For example,

Word	Syllables
Harry	2
hairy	2
hare	1
the	1

■■■ **P4.12** Write a program that reads a word and prints all substrings, sorted by length. For example, if the user provides the input `"rum"`, the program prints

```
r
u
m
ru
um
rum
```

■■ **P4.13** Write a program that reads an integer value and prints all of its *binary digits* in reverse order: Print the remainder `number % 2`, then replace the number with `number // 2`. Keep going until the number is 0. For example, if the user provides the input 13, the output should be

```
1
0
1
1
```

■■ **P4.14** *Mean and standard deviation.* Write a program that reads a set of floating-point data values. Choose an appropriate mechanism for prompting for the end of the data set. When all values have been read, print out the count of the values, the average, and the standard deviation. The average of a data set $\{x_1, \ldots, x_n\}$ is $\overline{x} = \sum x_i / n$, where $\sum x_i = x_1 + \ldots + x_n$ is the sum of the input values. The standard deviation is

$$s = \sqrt{\frac{\sum (x_i - \overline{x})^2}{n - 1}}$$

However, this formula is not suitable for the task. By the time the program has computed $\overline{x}$, the individual x_i are long gone. Until you know how to save these values, use the numerically less stable formula

$$s = \sqrt{\frac{\sum x_i^2 - \frac{1}{n}\left(\sum x_i\right)^2}{n - 1}}$$

You can compute this quantity by keeping track of the count, the sum, and the sum of squares as you process the input values.

■■ **P4.15** The *Fibonacci numbers* are defined by the sequence

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Fibonacci numbers describe the growth of a rabbit population.

Reformulate that as

```
fold1 = 1
fold2 = 1
fnew = fold1 + fold2
```

After that, discard `fold2`, which is no longer needed, and set `fold2` to `fold1` and `fold1` to `fnew`. Repeat an appropriate number of times.

Implement a program that prompts the user for an integer n and prints the nth Fibonacci number, using the above algorithm.

■■■ **P4.16** *Factoring of integers.* Write a program that asks the user for an integer and then prints out all its factors. For example, when the user enters 150, the program should print

```
2
3
5
5
```

■■■ **P4.17** *Prime numbers.* Write a program that prompts the user for an integer and then prints out all prime numbers up to that integer. For example, when the user enters 20, the program should print

```
2
3
5
7
11
13
17
19
```

Recall that a number is a prime number if it is not divisible by any number except 1 and itself.

■ **P4.18** Write a program that prints a multiplication table, like this:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 . . .
10  20  30  40  50  60  70  80  90 100
```

■■ **P4.19** Modify the `examaverages.py` program from Worked Example 4.1 so it will also compute the overall average exam grade.

■■ **P4.20** Modify the `examaverages.py` program from Worked Example 4.1 to have it validate the input when the user is prompted as to whether they want to enter grades for another student.

■■ **P4.21** Write a program that reads an integer and displays, using asterisks, a filled and hollow square, placed next to each other. For example if the side length is 5, the program should display

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

■■ **P4.22** Write a program that reads an integer and displays, using asterisks, a filled diamond of the given side length. For example, if the side length is 4, the program should display

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

■■■ **P4.23** *The game of Nim.* This is a well-known game with a number of variants. The following variant has an interesting winning strategy. Two players alternately take marbles from a pile. In each move, a player chooses how many marbles to take. The player must take at least one but at most half of the marbles. Then the other player takes a turn. The player who takes the last marble loses.

Write a program in which the computer plays against a human opponent. Generate a random integer between 10 and 100 to denote the initial size of the pile. Generate a random integer between 0 and 1 to decide whether the computer or the human takes the first turn. Generate a random integer between 0 and 1 to decide whether the computer plays *smart* or *stupid*. In stupid mode the computer simply takes a random legal value (between 1 and $n/2$) from the pile whenever it has a turn. In smart mode the computer takes off enough marbles to make the size of the pile a power of two minus 1—that is, 3, 7, 15, 31, or 63. That is always a legal move, except when the size of the pile is currently one less than a power of two. In that case, the computer makes a random legal move.

You will note that the computer cannot be beaten in smart mode when it has the first move, unless the pile size happens to be 15, 31, or 63. Of course, a human player who has the first turn and knows the winning strategy can win against the computer.

■■ **P4.24** *The Drunkard's Walk.* A drunkard in a grid of streets randomly picks one of four directions and stumbles to the next intersection, then again randomly picks one of four directions, and so on. You might think that on average the drunkard doesn't move very far because the choices cancel each other out, but that is actually not the case.

Represent locations as integer pairs (x, y). Implement the drunkard's walk over 100 intersections, starting at (0, 0), and print the ending location.

■■ **P4.25** *The Monty Hall Paradox.* Marilyn vos Savant described the following problem (loosely based on a game show hosted by Monty Hall) in a popular magazine: "Suppose you're on a game show, and you're given the choice of three doors: Behind one