Repetition

3

How would you write a program that repeats the same task multiple times? You could copy the code and paste it several times, but such a solution is inelegant. It only allows the task to be performed a fixed number of times, and any enhancements or corrections need to be made to every copy of the code.

Python provides two looping constructs that overcome these limitations. Both types of loop allow statements that occur only once in your program to execute multiple times when your program runs. When used effectively, loops can perform a large number of calculations with a small number statements.

3.1 While Loops

A `while` loop causes one or more statements to execute as long as, or *while*, a condition evaluates to `True`. Like an `if` statement, a `while` loop has a condition that is followed by a body which is indented. If the `while` loop's condition evaluates to `True` then the body of the loop is executed. When the bottom of the loop body is reached, execution returns to the top of the loop, and the loop condition is evaluated again. If the condition still evaluates to `True` then the body of the loop executes for a second time. Once the bottom of the loop body is reached for the second time, execution once again returns to the top of the loop. The loop's body continues to execute until the `while` loop condition evaluates to `False`. When this occurs, the loop's body is skipped, and execution continues at the first statement after the body of the `while` loop.

Many `while` loop conditions compare a variable holding a value read from the user to some other value. When the value is read in the body of the loop the user is able to cause the loop to terminate by entering an appropriate value. Specifically,

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_3

the value entered by the user must cause the `while` loop's condition to evaluate to `False`. For example, the following code segment reads values from the user and reports whether each value is positive or negative. The loop terminates when the user enters `0`. Neither message is displayed in this case.

```
# Read the first value from the user
x = int(input("Enter an integer (0 to quit): "))

# Keep looping while the user enters a non-zero number
while x != 0:
  # Report the nature of the number
  if x > 0:
    print("That's a positive number.")
  else:
    print("That's a negative number.")

  # Read the next value from the user
  x = int(input("Enter an integer (0 to quit): "))
```

This program begins by reading an integer from the user. If the integer is `0` then the condition on the `while` loop evaluates to `False`. When this occurs, the loop body is skipped and the program terminates without displaying any output (other than the prompt for input). If the condition on the `while` loop evaluates to `True` then the body of the loop executes.

When the loop body executes the value entered by the user is compared to `0` using an `if` statement and the appropriate message is displayed. Then the next input value is read from the user at the bottom of the loop. Since the bottom of the loop has been reached control returns to the top of the loop and its condition is evaluated again. If the most recent value entered by the user is `0` then the condition evaluates to `False`. When this occurs the body of the loop is skipped and the program terminates. Otherwise the body of the loop executes again. Its body continues to execute until the user causes the loop's condition to evaluate to `False` by entering `0`.

3.2 For Loops

Like `while` loops, `for` loops cause statements that only appear in a program once to execute several times when the program runs. However the mechanism used to determine how many times those statements will execute is rather different for a `for` loop.

A `for` loop executes once *for* each item in a collection. The collection can be a range of integers, the letters in a string, or as we'll see in later chapters, the values stored in a data structure, such as a list. The syntactic structure of a `for` loop is shown below, where `<variable>`, `<collection>` and `<body>` are placeholders that must be filled in appropriately.

```
for <variable> in <collection>:
  <body>
```

The body of the loop consists of one or more Python statements that may be executed multiple times. In particular, these statements will execute once for each item in the collection. Like a `while` loop body, the body of a `for` loop is indented.

Each item in the collection is copied into `<variable>` before the loop body executes for that item. This variable is created by the `for` loop when it executes. It is not necessary to create it with an assignment statement, and any value that might have been assigned to this variable previously is overwritten at the beginning of each loop iteration. The variable can be used in the body of the loop in the same ways that any other Python variable can be used.

A collection of integers can be constructed by calling Python's `range` function. Calling `range` with one argument returns a range that starts with 0 and increases up to, but does not include, the value of the argument. For example, `range(4)` returns a range consisting of 0, 1, 2 and 3.

When two arguments are provided to `range` the collection of values returned increases from the first argument up to, but not including, the second argument. For example, `range(4, 7)` returns a range that consists of 4, 5 and 6. An empty range is returned when `range` is called with two arguments and the first argument is greater than or equal to the second. The body of the `for` loop is skipped any time a `for` loop is applied to an empty range. Execution continues with the first statement after the `for` loop's body.

The `range` function can also be called with a third argument, which is the step value used to move from the initial value in the range toward its final value. Using a step value greater than 0 results in a range that begins with the first argument and increases up to, but does not include, the second argument, incrementing by the step value each time. Using a negative step value allows a collection of decreasing values to be constructed. For example, while calling `range(0, -4)` returns an empty range, calling `range(0, -4, -1)` returns a range that consists of 0, -1, -2 and -3. Note that the step value passed to `range` as its third argument must be an integer. Problems which require a non-integer step value are often solved with a `while` loop instead of a `for` loop because of this restriction.

The following program uses a `for` loop and the range `function` to display all of the positive multiples of 3 up to (and including) a value entered by the user.

```
# Read the limit from the user
limit = int(input("Enter an integer: "))

# Display the positive multiples of 3 up to the limit
print("The multiples of 3 up to and including", limit, "are:")
for i in range(3, limit + 1, 3):
  print(i)
```

When this program executes it begins by reading an integer from the user. We will assume that the user entered 11 as we describe the execution of the rest of this program. After the input value is read, execution continues with the `print` statement that describes the program's output. Then the `for` loop begins to execute.

A range of integers is constructed that begins with 3 and goes up to, but does not include, $11 + 1 = 12$, stepping up by 3 each time. Thus the range consists of 3, 6 and

9. When the loop executes for the first time the first integer in the range is assigned to `i`, the body of the loop is executed, and `3` is displayed.

Once the loop's body has finished executing for the first time, control returns to the top of the loop and the next value in the range, which is 6, is assigned to `i`. The body of the loop executes again and displays `6`. Then control returns to the top of the loop for a second time.

The next value assigned to `i` is 9. It is displayed the next time the loop body executes. Then the loop terminates because there are no further values in the range. Normally execution would continue with the first statement after the body of the `for` loop. However, there is no such statement in this program, so the program terminates.

3.3 Nested Loops

The statements inside the body of a loop can include another loop. When this happens, the inner loop is said to be *nested* inside the outer loop. Any type of loop can be nested inside of any other type of loop. For example, the following program uses a `for` loop nested inside a `while` loop to repeat messages entered by the user until the user enters a blank message.

```
# Read the first message from the user
message = input("Enter a message (blank to quit): ")

# Loop until the message is a blank line
while message != "":
  # Read the number of times the message should be displayed
  n = int(input("How many times should it be repeated? "))

  # Display the message the number of times requested
  for i in range(n):
    print(message)

  # Read the next message from the user
  message = input("Enter a message (blank to quit): ")
```

When this program executes it begins by reading the first message from the user. If that message is not blank then the body of the `while` loop executes and the program reads the number of times to repeat the message, `n`, from the user. A range of integers is created from 0 up to, but not including, `n`. Then the body of the `for` loop prints the message `n` times because the message is displayed once for each integer in the range.

The next message is read from the user after the `for` loop has executed `n` times. Then execution returns to the top of the `while` loop, and its condition is evaluated. If the condition evaluates to `True` then the body of the `while` loop runs again. Another integer is read from the user, which overwrites the previous value of `n`, and then the `for` loop prints the message `n` times. This continues until the condition on the `while` loop evaluates to `False`. When that occurs, the body of the `while` loop is skipped and the program terminates because there are no statements to execute after the body of the `while` loop.

3.4 Exercises

The following exercises should all be completed with loops. In some cases the exercise specifies what type of loop to use. In other cases you must make this decision yourself. Some of the exercises can be completed easily with both `for` loops and `while` loops. Other exercises are much better suited to one type of loop than the other. In addition, some of the exercises require multiple loops. When multiple loops are involved one loop might need to be nested inside the other. Carefully consider your choice of loops as you design your solution to each problem.

Exercise 63: Average

(*26 Lines*)

In this exercise you will create a program that computes the average of a collection of values entered by the user. The user will enter 0 as a sentinel value to indicate that no further values will be provided. Your program should display an appropriate error message if the first value entered by the user is 0.

> Hint: Because the 0 marks the end of the input it should **not** be included in the average.

Exercise 64: Discount Table

(*18 Lines*)

A particular retailer is having a 60 percent off sale on a variety of discontinued products. The retailer would like to help its customers determine the reduced price of the merchandise by having a printed discount table on the shelf that shows the original prices and the prices after the discount has been applied. Write a program that uses a loop to generate this table, showing the original price, the discount amount, and the new price for purchases of $4.95, $9.95, $14.95, $19.95 and $24.95. Ensure that the discount amounts and the new prices are rounded to 2 decimal places when they are displayed.

Exercise 65: Temperature Conversion Table

(*22 Lines*)

Write a program that displays a temperature conversion table for degrees Celsius and degrees Fahrenheit. The table should include rows for all temperatures between 0 and 100 degrees Celsius that are multiples of 10 degrees Celsius. Include appropriate headings on your columns. The formula for converting between degrees Celsius and degrees Fahrenheit can be found on the Internet.

Exercise 66: No More Pennies

(*Solved, 39 Lines*)

February 4, 2013 was the last day that pennies were distributed by the Royal Canadian Mint. Now that pennies have been phased out retailers must adjust totals so that they are multiples of 5 cents when they are paid for with cash (credit card and debit card transactions continue to be charged to the penny). While retailers have some freedom in how they do this, most choose to round to the closest nickel.

Write a program that reads prices from the user until a blank line is entered. Display the total cost of all the entered items on one line, followed by the amount due if the customer pays with cash on a second line. The amount due for a cash payment should be rounded to the nearest nickel. One way to compute the cash payment amount is to begin by determining how many pennies would be needed to pay the total. Then compute the remainder when this number of pennies is divided by 5. Finally, adjust the total down if the remainder is less than 2.5. Otherwise adjust the total up.

Exercise 67: Compute the Perimeter of a Polygon

(*Solved, 42 Lines*)

Write a program that computes the perimeter of a polygon. Begin by reading the x and y coordinates for the first point on the perimeter of the polygon from the user. Then continue reading pairs of values until the user enters a blank line for the x-coordinate. Each time you read an additional coordinate you should compute the distance to the previous point and add it to the perimeter. When a blank line is entered for the x-coordinate your program should add the distance from the last point back to the first point to the perimeter. Then the perimeter should be displayed. Sample input and output values are shown below. The input values entered by the user are shown in bold.

```
Enter the first x-coordinate: 0
Enter the first y-coordinate: 0
Enter the next x-coordinate (blank to quit): 1
Enter the next y-coordinate: 0
Enter the next x-coordinate (blank to quit): 0
Enter the next y-coordinate: 1
Enter the next x-coordinate (blank to quit):
The perimeter of that polygon is 3.414213562373095
```

Exercise 68: Compute a Grade Point Average

(*62 Lines*)

Exercise 52 includes a table that shows the conversion from letter grades to grade points at a particular academic institution. In this exercise you will compute the grade point average of an arbitrary number of letter grades entered by the user. The

user will enter a blank line to indicate that all of the grades have been provided. For example, if the user enters `A`, followed by `C+`, followed by `B`, followed by a blank line then your program should report a grade point average of 3.1.

You may find your solution to Exercise 52 helpful when completing this exercise. Your program does not need to do any error checking. It can assume that each value entered by the user will always be a valid letter grade or a blank line.

Exercise 69: Admission Price

(*Solved, 38 Lines*)

A particular zoo determines the price of admission based on the age of the guest. Guests 2 years of age and less are admitted without charge. Children between 3 and 12 years of age cost $14.00. Seniors aged 65 years and over cost $18.00. Admission for all other guests is $23.00.

Create a program that begins by reading the ages of all of the guests in a group from the user, with one age entered on each line. The user will enter a blank line to indicate that there are no more guests in the group. Then your program should display the admission cost for the group with an appropriate message. The cost should be displayed using two decimal places.

Exercise 70: Parity Bits

(*Solved, 27 Lines*)

A parity bit is a simple mechanism for detecting errors in data transmitted over an unreliable connection such as a telephone line. The basic idea is that an additional bit is transmitted after each group of 8 bits so that a single bit error in the transmission can be detected.

Parity bits can be computed for either even parity or odd parity. If even parity is selected then the parity bit that is transmitted is chosen so that the total number of one bits transmitted (8 bits of data plus the parity bit) is even. When odd parity is selected the parity bit is chosen so that the total number of one bits transmitted is odd.

Write a program that computes the parity bit for groups of 8 bits entered by the user using even parity. Your program should read strings containing 8 bits until the user enters a blank line. After each string is entered by the user your program should display a clear message indicating whether the parity bit should be 0 or 1. Display an appropriate error message if the user enters something other than 8 bits.

Hint: You should read the input from the user as a string. Then you can use the `count` method to help you determine the number of zeros and ones in the string. Information about the `count` method is available online.

Exercise 71: Approximate π

(*23 Lines*)

The value of π can be approximated by the following infinite series:

$$\pi \approx 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \frac{4}{8 \times 9 \times 10} + \frac{4}{10 \times 11 \times 12} - \cdots$$

Write a program that displays 15 approximations of π. The first approximation should make use of only the first term from the infinite series. Each additional approximation displayed by your program should include one more term in the series, making it a better approximation of π than any of the approximations displayed previously.

Exercise 72: Fizz-Buzz

(*17 Lines*)

Fizz-Buzz is a game that is sometimes played by children to help them learn about division. The players are commonly arranged in a circle so that the game can progress from player to player continually. The starting player begins by saying one, and then play passes to the player to the left. Each subsequent player is responsible for the next integer in sequence before play passes to the following player. On a player's turn they must either say their number or one of following substitutions:

- If the player's number is divisible by 3 then the player says fizz instead of their number.
- If the player's number is divisible by 5 then the player says buzz instead of their number.

A player must say both fizz and buzz for numbers that are divisible by both 3 and 5. Any player that fails to perform the correct substitution or hesitates before answering is eliminated from the game. The last player remaining is the winner.

Write a program that displays the answers for the first 100 numbers in the Fizz-Buzz game. Each answer should be displayed on its own line.

Exercise 73: Caesar Cipher

(*Solved, 35 Lines*)

One of the first known examples of encryption was used by Julius Caesar. Caesar needed to provide written instructions to his generals, but he didn't want his enemies to learn his plans if the message slipped into their hands. As a result, he developed what later became known as the Caesar cipher.

The idea behind this cipher is simple (and as such, it provides no protection against modern code breaking techniques). Each letter in the original message is shifted by 3 places. As a result, A becomes D, B becomes E, C becomes F, D becomes G, etc.

The last three letters in the alphabet are wrapped around to the beginning: X becomes A, Y becomes B and Z becomes C. Non-letter characters are not modified by the cipher.

Write a program that implements a Caesar cipher. Allow the user to supply the message and the shift amount, and then display the shifted message. Ensure that your program encodes both uppercase and lowercase letters. Your program should also support negative shift values so that it can be used both to encode messages and decode messages.

Exercise 74: Square Root

(*14 Lines*)

Write a program that implements Newton's method to compute and display the square root of a number, x, entered by the user. The algorithm for Newton's method follows:

Read x from the user
Initialize *guess* to $x/2$
While *guess* is not good enough **do**
  Update *guess* to be the average of *guess* and $x/guess$

When this algorithm completes, *guess* contains an approximation of the square root of x. The quality of the approximation depends on how you define "good enough". In the author's solution, *guess* was considered good enough when the absolute value of the difference between $guess * guess$ and x was less than or equal to 10^{-12}.

Exercise 75: Is a String a Palindrome?

(*Solved, 26 Lines*)

A string is a palindrome if it is identical forward and backward. For example "anna", "civic", "level" and "hannah" are all examples of palindromic words. Write a program that reads a string from the user and uses a loop to determine whether or not it is a palindrome. Display the result, including a meaningful output message.

Aibohphobia is the extreme or irrational fear of palindromes. This word was constructed by prepending the -phobia suffix with it's reverse, resulting in a palindrome. Ailihphilia is the fondness for or love of palindromes. It was constructed in the same manner from the -philia suffix.

Exercise 76: Multiple Word Palindromes

(*35 Lines*)

There are numerous phrases that are palindromes when spacing is ignored. Examples include "go dog", "flee to me remote elf" and "some men interpret nine memos", among many others. Extend your solution to Exercise 75 so that it ignores spacing while determining whether or not a string is a palindrome. For an additional challenge, further extend your solution so that is also ignores punctuation marks and treats uppercase and lowercase letters as equivalent.

Exercise 77: Multiplication Table

(*Solved, 18 Lines*)

In this exercise you will create a program that displays a multiplication table that shows the products of all combinations of integers from 1 times 1 up to and including 10 times 10. Your multiplication table should include a row of labels across the top of it containing the numbers 1 through 10. It should also include labels down the left side consisting of the numbers 1 through 10. The expected output from the program is shown below:

```
     1   2   3   4   5   6   7   8   9  10
 1   1   2   3   4   5   6   7   8   9  10
 2   2   4   6   8  10  12  14  16  18  20
 3   3   6   9  12  15  18  21  24  27  30
 4   4   8  12  16  20  24  28  32  36  40
 5   5  10  15  20  25  30  35  40  45  50
 6   6  12  18  24  30  36  42  48  54  60
 7   7  14  21  28  35  42  49  56  63  70
 8   8  16  24  32  40  48  56  64  72  80
 9   9  18  27  36  45  54  63  72  81  90
10  10  20  30  40  50  60  70  80  90 100
```

When completing this exercise you will probably find it helpful to be able to print out a value without moving down to the next line. This can be accomplished by added `end=""` as the last argument to your `print` statement. For example, `print("A")` will display the letter `A` and then move down to the next line. The statement `print("A", end="")` will display the letter `A` without moving down to the next line, causing the next print statement to display its result on the same line as the letter `A`.

Exercise 78: The Collatz Conjecture

(*22 Lines*)

Consider a sequence of integers that is constructed in the following manner:

Start with any positive integer as the only term in the sequence
While the last term in the sequence is not equal to 1 **do**
  If the last term is even **then**
    Add another term to the sequence by dividing the last term by 2 using floor division
  Else
    Add another term to the sequence by multiplying the last term by 3 and adding 1

The Collatz conjecture states that this sequence will eventually end with one when it begins with any positive integer. Although this conjecture has never been proved, it appears to be true.

Create a program that reads an integer, n, from the user and reports all of the values in the sequence starting with n and ending with one. Your program should allow the user to continue entering new n values (and your program should continue displaying the sequences) until the user enters a value for n that is less than or equal to zero.

> The Collatz conjecture is an example of an open problem in mathematics. While many people have tried to prove that it is true, no one has been able to do so. Information on other open problems in mathematics can be found on the Internet.

Exercise 79: Greatest Common Divisor

(*Solved, 17 Lines*)

The greatest common divisor of two positive integers, n and m, is the largest number, d, which divides evenly into both n and m. There are several algorithms that can be used to solve this problem, including:

Initialize d to the smaller of m and n.
While d does not evenly divide m or d does not evenly divide n **do**
  Decrease the value of d by 1
Report d as the greatest common divisor of n and m

Write a program that reads two positive integers from the user and uses this algorithm to determine and report their greatest common divisor.

Exercise 80: Prime Factors

(*27 Lines*)

The prime factorization of an integer, *n*, can be determined using the following steps:

```
Initialize factor to 2
While factor is less than or equal to n do
    If n is evenly divisible by factor then
        Conclude that factor is a factor of n
        Divide n by factor using floor division
    Else
        Increase factor by 1
```

Write a program that reads an integer from the user. If the value entered by the user is less than 2 then your program should display an appropriate error message. Otherwise your program should display the prime numbers that can be multiplied together to compute *n*, with one factor appearing on each line. For example:

```
Enter an integer (2 or greater): 72
The prime factors of 72 are:
2
2
2
3
3
```

Exercise 81: Binary to Decimal

(*18 Lines*)

Write a program that converts a binary (base 2) number to decimal (base 10). Your program should begin by reading the binary number from the user as a string. Then it should compute the equivalent decimal number by processing each digit in the binary number. Finally, your program should display the equivalent decimal number with an appropriate message.

Exercise 82: Decimal to Binary

(*Solved, 27 Lines*)

Write a program that converts a decimal (base 10) number to binary (base 2). Read the decimal number from the user as an integer and then use the division algorithm shown below to perform the conversion. When the algorithm completes, *result* contains the binary representation of the number. Display the result, along with an appropriate message.

Let *result* be an empty string
Let q represent the number to convert
repeat
  Set r equal to the remainder when q is divided by 2
  Convert r to a string and add it to the beginning of *result*
  Divide q by 2, discarding any remainder, and store the result back into q
until q is 0

Exercise 83: Maximum Integer

(*Solved, 34 Lines*)

This exercise examines the process of identifying the maximum value in a collection of integers. Each of the integers will be randomly selected from the numbers between 1 and 100. The collection of integers may contain duplicate values, and some of the integers between 1 and 100 may not be present.

Take a moment and think about how you would solve this problem on paper. Many people would check each integer in sequence and ask themself if the number that they are currently considering is larger than the largest number that they have seen previously. If it is, then they forget the previous maximum number and remember the current number as the new maximum number. This is a reasonable approach, and will result in the correct answer when the process is performed carefully. If you were performing this task, how many times would you expect to need to update the maximum value and remember a new number?

While we can answer the question posed at the end of the previous paragraph using probability theory, we are going to explore it by simulating the situation. Create a program that begins by selecting a random integer between 1 and 100. Save this integer as the maximum number encountered so far. After the initial integer has been selected, generate 99 additional random integers between 1 and 100. Check each integer as it is generated to see if it is larger than the maximum number encountered so far. If it is then your program should update the maximum number encountered and count the fact that you performed an update. Display each integer after you generate it. Include a notation with those integers which represent a new maximum.

After you have displayed 100 integers your program should display the maximum value encountered, along with the number of times the maximum value was updated during the process. Partial output for the program is shown below, with… representing the remaining integers that your program will display. Run your program several times. Is the number of updates performed on the maximum value what you expected?

```
30
74 <== Update
58
17
40
37
13
34
46
52
80 <== Update
37
97 <== Update
45
55
73
...

The maximum value found was 100
The maximum value was updated 5 times
```

Exercise 84: Coin Flip Simulation

(*47 Lines*)

What's the minimum number of times you have to flip a coin before you can have three consecutive flips that result in the same outcome (either all three are heads or all three are tails)? What's the maximum number of flips that might be needed? How many flips are needed on average? In this exercise we will explore these questions by creating a program that simulates several series of coin flips.

Create a program that uses Python's random number generator to simulate flipping a coin several times. The simulated coin should be fair, meaning that the probability of heads is equal to the probability of tails. Your program should flip simulated coins until either 3 consecutive heads of 3 consecutive tails occur. Display an `H` each time the outcome is heads, and a `T` each time the outcome is tails, with all of the outcomes for one simulation on the same line. Then display the number of flips that were needed to reach 3 consecutive occurrences of the same outcome. When your program is run it should perform the simulation 10 times and report the average number of flips needed. Sample output is shown below:

```
H T T T (4 flips)
H H T T H T H T T H H T H T T H T T T (19 flips)
T T T (3 flips)
T H H H (4 flips)
H H H (3 flips)
T H T T H T H H T T H H T H T H H H (18 flips)
H T T H H H (6 flips)
T H T T T (5 flips)
T T H T T H T H T H H H (12 flips)
T H T T T (5 flips)
On average, 7.9 flips were needed.
```