REPETITION STRUCTURES

Introduction

A repetition structure causes a statement or set of statements to execute repeatedly.

A repetition structure, which is more commonly known as a ***loop*** **causes repeating of a sequence of statements.**

Instead of writing the same sequence of statements over and over we can write the code for the operation once, and then place that code in a structure that makes the computer repeat it as many times as necessary.

There are two types of loops:

- ***A count-controlled loop*** repeats a specific number of times. (for)
- ***A condition-controlled loop*** uses a true/false condition to control the number of times that it repeats. (while)

- **The for Loop: A Count-Controlled Loop**

A count-controlled loop iterates a specific number of times. In Python you use the **for** statement to write a count-controlled loop.

In Python, the **for** statement is designed to work with a sequence of data items. When the statement executes, it iterates once for each item in the sequence.

Using a list with the for Loop

The general format of the for loop in Python:

for *variable* in *[value1, value2, etc.]*:
    statement
    statement
    etc.

In the **for** clause, *variable* is the name of a variable. Inside the brackets a sequence of values appears, with a comma separating each value.

In Python, a comma-separated sequence of data items that are enclosed in a set of brackets is called a ***list***.

The **for** statement executes in the following manner:
The *variable* is assigned the first value in the list, and then the statements that appear in the block are executed. Then, *variable* is assigned the next value in the list, and the statements in the block are executed again. This continues until *variable* has been assigned the last value in the list.

Using the range Function with the for Loop

Python provides a built-in function named **range** that simplifies the process of writing a count-controlled for loop.

The range function creates a type of object known as an iterable.

An *iterable* is an object that is similar to a list; it contains a sequence of values that can be iterated over with something like a loop.

Example of a for loop that uses the range function:

```
for num in range(5):
   print(num)
```

This code works the same as the following:

```
for num in [0, 1, 2, 3, 4]:
    print(num)
```

If you pass one argument to the range function, that argument is used as the ending limit of the sequence of numbers.

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

Example:

```
for num in range(1, 5):
print(num)
```

This code will display the following:

```
1
2
3
4
```

By default, the range function produces a sequence of numbers that increase by 1 for each successive number in the list.

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

Example:

```
for num in range(1, 10, 2):
print(num)
```

This code will display the following:

```
1
3
5
7
9
```

Generating an Iterable Sequence that Ranges from Highest to Lowest

We can use the range function to generate sequences of numbers that go from highest to lowest.

Example:

```
range(10, 0, –1)
```

Example of a for loop that prints the numbers 5 down to 1:

```
for num in range(5, 0, –1):
    print(num)
```

Calculating a Running Total

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

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

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

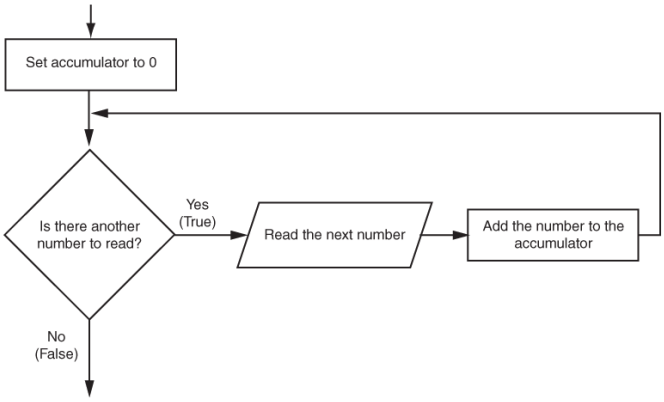

Figure 4-6 Logic for calculating a running total

The Augmented Assignment Operators

Programs can have assignment statements in which the variable that is on the left side of the = operator also appears on the right side of the = operator.

Example:

```
x = x + 1
```

On the right side of the assignment operator, 1 is added to x. The result is then assigned to x, replacing the value that x previously referenced. Effectively, this statement adds 1 to x.

Examples of this type of statement:

```
total = total + number

balance = balance - withdrawal
```

Table 4-2 Augmented assignment operators

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