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.

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:

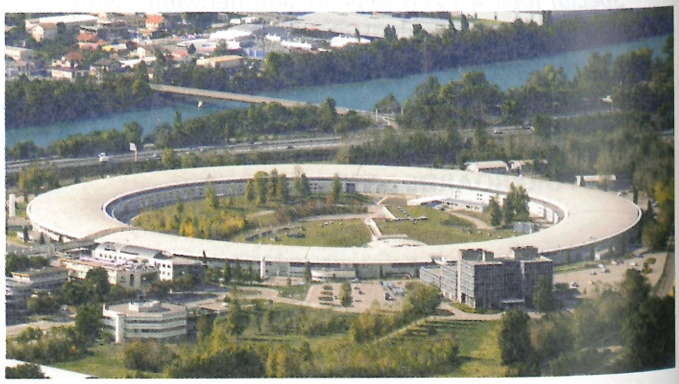
Because the interest earned also earns interest, a bank balance grows exponentially.

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

year	interest	balance
0		$10,000

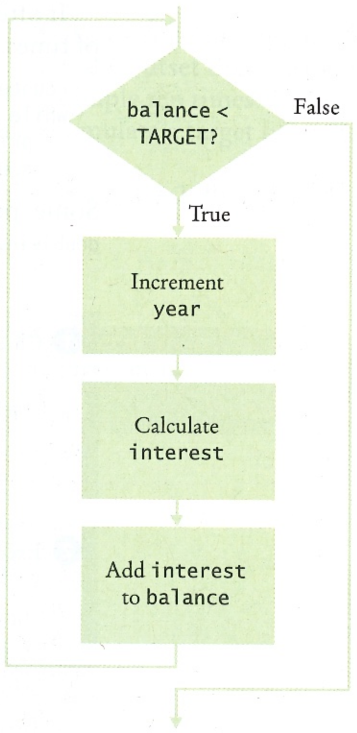
```
Repeat the following steps while the balance is less than $20,000.
    Add 1 to the year value.
    Compute the interest as balance x 0.05 (i.e., 5 percent interest).
    Add the interest to the balance.
Report the final year value as the answer.
```

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
```

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

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.

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

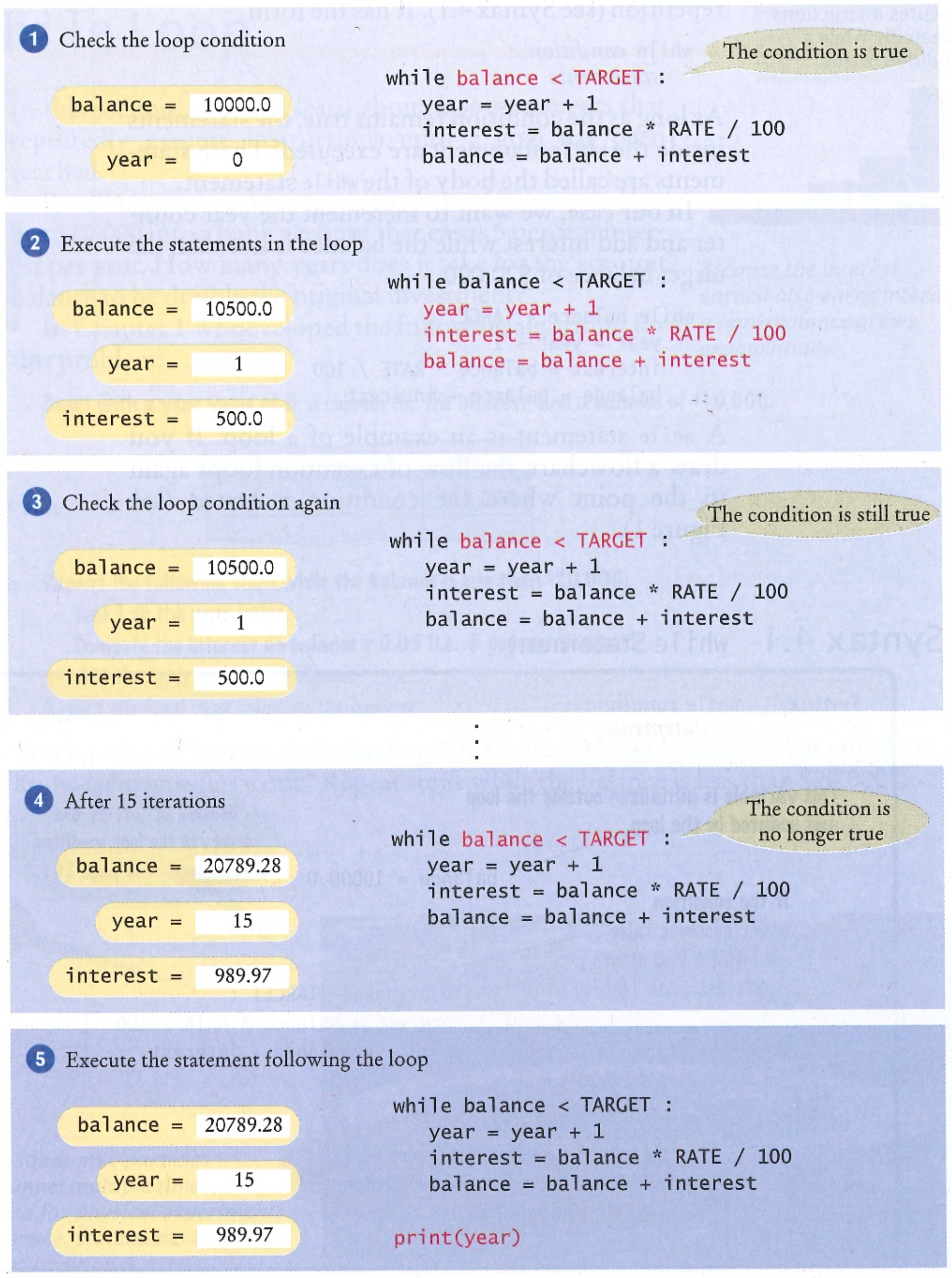


Figure 2
Execution of the `doubleinv.py` Loop

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

```
##
#  This program computes the time required to double an investment.
#

# Create constant variables.
RATE = 5.0
INITIAL_BALANCE = 10000.0
TARGET = 2 * INITIAL_BALANCE

# Initialize variables used with the loop.
balance = INITIAL_BALANCE
year = 0

# Count the years required for the investment to double.
while balance < TARGET :
   year = year + 1
   interest = balance * RATE / 100
   balance = balance + interest

# Print the results.
print("The investment doubled after", year, "years.")
```

Program Run

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

SELF CHECK

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.

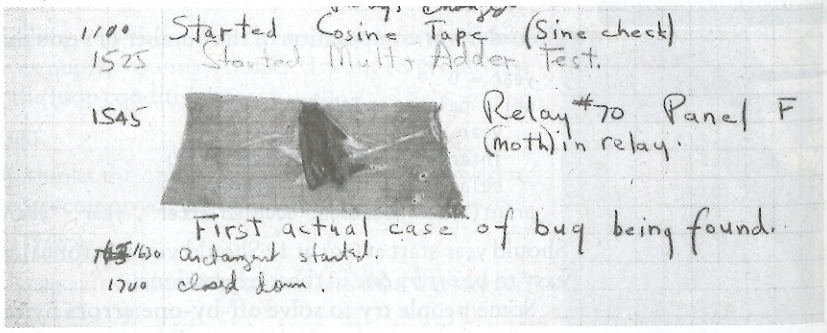
In other words, you keep adding interest while the balance *has not yet doubled*.

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."



1100 Started Cosine Tape (Sine check)
1525 Started Mult+ Adder Test.
1545 Relay #70 Panel F (moth) in relay.
First actual case of bug being found.
~~1[illegible]~~1630 Antangent started.
1700 closed down.

The First Bug