PREV
Unit 4. Repeating tasks

NEXT
Lesson 17. Customizing loops

Lesson 16. Repeating tasks with loops

After reading lesson 16, you'll be able to

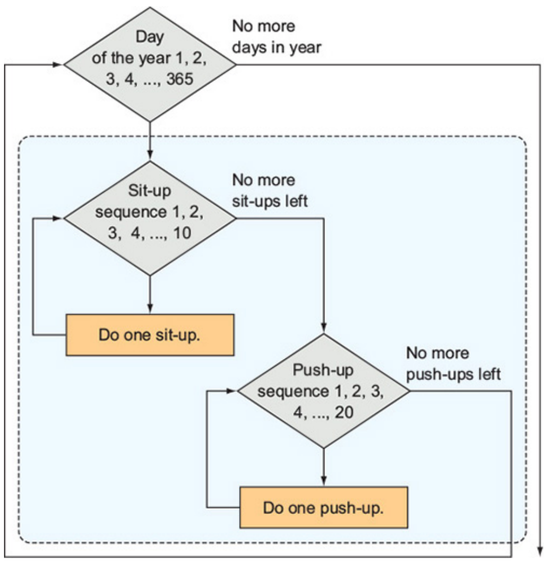
- Understand what it means for a line of code to repeat execution
- Write a loop in a program
- Repeat actions a certain number of times

The programs you've seen so far have statements that are executed once, at most. In the preceding unit, you learned to add decision points in your programs, which can break up the flow by making the program react to input. The decision points are governed by conditional statements that may cause your program to take a detour to execute other lines of code if a certain condition holds.

These kinds of programs still have a type of linearity to them; statements are executed top to bottom, and a statement can be executed either zero times or at most one time. Therefore, the maximum number of statements that can be executed in your program is the maximum number of lines in the program.

Consider this

It's a new year, and your resolution is to do 10 push-ups and 20 sit-ups every day. Look at the following flowchart to determine the number of sit-ups and push-ups you'll do in one year.


Day of the year 1, 2, 3, 4, ..., 365
No more days in year
Sit-up sequence 1, 2, 3, 4, ..., 10
No more sit-ups left
Do one sit-up.
Push-up sequence 1, 2, 3, 4, ..., 20
No more push-ups left
Do one push-up.

A flowchart illustrating how you can repeat certain tasks. Sit-ups are repeated 10 times, and push-ups are repeated 20 times. Both sit-ups and push-up sequences are done every day of the year

Answer: 3,650 sit-ups and 7,300 push-ups

16.1. REPEATING A TASK

The power of computers comes from their capability to do computations quickly. Using what you've learned so far, if you wanted to execute a statement or a slight variation on a statement more than once, you'd have to type it in your program again so that the interpreter sees it as a separate command. Doing this defeats the purpose of having a computer do the work for you. In this lesson, you'll construct loops, which tell the interpreter to repeat a certain task (represented by a set of statements) many times.

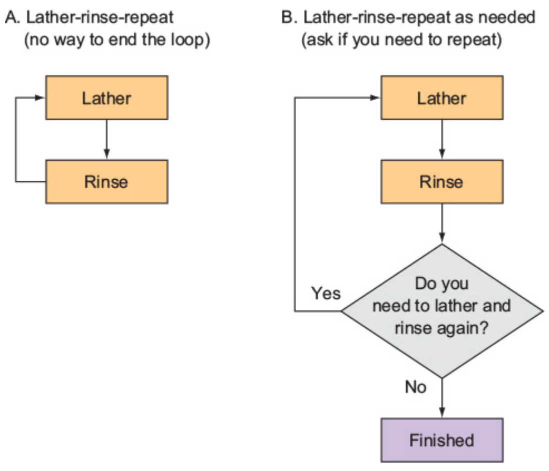
In your everyday life, you often repeat a certain task while changing a small part of it. For example, when you arrive at work or school, you might greet people with "Hi Joe" and then "Hi Beth" and then "Hi Alice." This task has something in common among all the repetitions (the word *Hi*), but a small part is changed with every repetition (a person's name). As another example, your shampoo bottle might indicate "lather, rinse, repeat." To *lather* and to *rinse* can be thought of as smaller subtasks that are both done, in the same order, with every repetition.

One of the many uses of computers is their capability to perform many computations in a short time. Doing repetitive tasks is what computers are best at, and programming a task such as playing every song in a playlist is easy. Every programming language has a way to tell the computer how to repeat a certain set of commands.

16.1.2. Infinite repetitions

Computers do only what you tell them to do, so you must be careful and explicit in your instructions. They can't guess your intentions. Suppose you write a program to implement the "lather, rinse, repeat" procedure. Pretend you've never used shampoo before and you're following the instructions without applying any other logic or reasoning. Notice anything wrong with the instructions? It's unclear when to stop the "lather, rinse" steps. How many times should you "lather and rinse"? If there's no set number of repetitions, when do you stop? These particular instructions are so vague that if you told a computer to do them, it would perform the "lather, rinse" procedure infinitely. A better instruction would be "lather, rinse, repeat as needed." The flowchart in figure 16.1 shows what happens when you tell a computer to "lather-rinse-repeat" and when you add an "as needed" clause to stop it from infinitely repeating.

Figure 16.1. The difference between (A) a lather-rinse-repeat instruction, which doesn't ever terminate if taken literally and (B) a lather-rinse-repeat as needed instruction, which asks whether to repeat every time


A. Lather-rinse-repeat
(no way to end the loop)
Lather
Rinse
B. Lather-rinse-repeat as needed
(ask if you need to repeat)
Lather
Rinse
Yes
Do you need to lather and rinse again?
No
Finished

Because computers do only what they're told, they can't make the decision of whether to repeat a set of commands on their own. You have to be careful to be specific when telling the computer to repeat commands: are there a certain number of times you want the commands to be repeated, or is there a condition that determines whether to repeat again? In the lather-rinse example, "as needed" was a condition that determined whether you were going to repeat lather-rinse. Alternatively, you might say that you want to lather-rinse three times and then stop.

Thinking like a programmer

Humans can fill in knowledge gaps and infer certain ideas in different situations. Computers do only what they're told. When writing code, the computer will execute everything you write according to the rules of the programming language. A bug doesn't spontaneously appear in your code. If a bug exists, it's because you put it there. Lesson 36 discusses formal ways to debug your programs.

16.2. LOOPING A CERTAIN NUMBER OF TIMES

In programming, you achieve repetition by constructing loops. One way to stop a program from infinitely repeating a set of instructions is to tell it the number of times to repeat the instructions. The name for this type of loop is a `for` *loop*.

16.2.1. for loops

In Python, the keyword that tells you how to loop a certain number of times is `for`. To start, here's one way to use the keyword to repeat a command many times:

Without using loops	Using loops

`print("echo")`
`print("echo")`
`print("echo")`
`print("echo")`
	`for i in range(4):`
	`    print("echo")`

Without using loops, you have to repeat the same command as many times as you need to. In this case, the command is printing the word *echo* four times. But using loops, you can condense the code into only two lines. The first line tells the Python interpreter the number of times to repeat a certain command. The second line tells the interpreter the command to repeat.

Quick check 16.1

1

Write a piece of code that prints the word *crazy* eight times on separate lines.

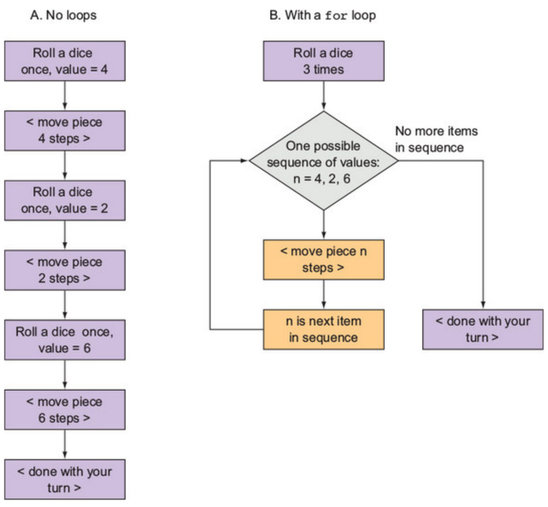
2

Write a piece of code that prints the word *centipede* 100 times on separate lines.

Now suppose you're playing a board game; on your turn, you have to roll the dice three times. After each roll, you move your game piece that many steps. Suppose the player rolls a 4, then a 2, and then a 6.

Figure 16.2 shows a flowchart of every step. With only three rolls, it's easy to model the game by writing commands to do the dice roll and move the piece, repeating those two actions three times. But this would get messy quickly if you allowed the player to do 100 rolls on their turn. Instead, it's better to model the player's turn by using a `for` loop.

Figure 16.2. In both (A) and (B), you roll a dice three times to give you values 4, 2, 6. (A) represents how you can move a game piece by explicitly writing out commands step-by-step. (B) shows how you can represent doing the same thing, except that you're using a `for` loop that iterates over the values representing each dice roll and you're generalizing the values by using a variable `n`.



The player rolls the dice three times to get a sequence of values. Represent the number on the dice as a variable `n`. The loop goes through the sequence representing the dice rolls, starting with the first number—in this case `n = 4`. Using this variable for the number of steps to take, you move the piece `n` steps. Then you go to the next number in the sequence, `n = 2`, and move the piece 2 steps. Lastly, you go to the final number in the sequence, `n = 6`, and move the piece 6 steps. Because there are no more numbers in the sequence, you can stop moving your piece, and your turn ends.

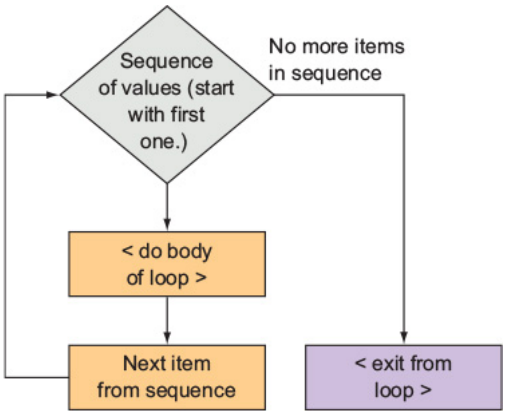
Listing 16.1 shows the general structure of a `for` loop. The same structure can be visualized using the flowchart in figure 16.3. The idea is that you're given a sequence of values. The body of the loop repeats execution as many times as there are values. With each repetition, you're changing a loop variable to be an item in the sequence of values. The loop stops repeating when you've gone through all the values in the sequence.

Listing 16.1. A general way to write a `for` loop

```
for <loop_variable> in <values>:             1
    <do something>                           2
```

- *1* **Indicates beginning of loop. <loop_variable> systematically takes on the value of each item in <values>.**
- *2* **Code block to execute for each item in <values>**

Figure 16.3. A general way of writing a `for` loop. You start with the first item in the sequence of values and then execute the body of the loop by using that item. You get the next item in the sequence of values and execute the body of the loop using that item. The loop ends when you've gone through and executed the body of the loop by using every item in the sequence.


Sequence of values (start with first one.)
No more items in sequence
< do body of loop >
Next item from sequence
< exit from loop >

A `for` loop consists of two parts: the `for` loop line definition, and the code block that gets executed a certain number of times.

The keyword `for` tells Python you're introducing a block that will be repeated a certain number of times. After the keyword, you name a loop variable. This can be any valid variable name you want. This loop variable automatically changes its value for every repetition, with subsequent values taken from the values determined by what comes after the keyword `in`.

As with conditionals, indentation matters with loops. The indented code block tells Python that everything in that code block is part of the loop.

16.3. LOOPING N TIMES

In the previous section, you didn't impose any constraints on the sequence of values. It's often useful to have loops whose sequence of values follows a pattern. For example, a common and useful pattern is to have the items in a sequence of values be sequentially increasing 1, 2, 3... until some value *N*. Because in computer science counting starts from 0, an even more common sequence of numbers is 0, 1, 2... until some value *N* – 1, to give a total of *N* items in a sequence.

16.3.1. Loops over the common sequence 0 to N – 1

If you want to loop *N* times, you replace the `<values>` in listing 16.1 with the expression `range(N)`, where `N` is an integer number. `range` is a special procedure in Python. The expression `range(N)` yields the sequence 0, 1, 2, 3, ... N – 1.

Quick check 16.2

What sequence of values does the following evaluate to?

1

`range(1)`

2

`range(5)`

3

`range(100)`

Listing 16.2 shows a simple `for` loop that repeatedly prints the value of the loop variable `v`. In listing 16.2, the loop variable is `v`, the sequence of values that the loop variable takes on is given by `range(3)`, and the body of the loop is one `print` statement.

When the program in listing 16.2 runs, and it first encounters the `for` loop with `range(3)`, it first assigns `0` to the loop variable `v` and then executes the print statement. In this example, this process is repeated three times, effectively assigning the loop variable the numbers 0, 1, 2.

Listing 16.2. A for loop that prints the loop variable value

```
for v in range(3):              1
    print("var v is", v)        2
```

- *1* **v is the loop variable.**
- *2* **Prints the loop variable**

You can generalize the behavior in listing 16.2 by using a different variable, say `n_times`, instead of 3, to give a sequence of numbers denoted by `range(n_times)`. Then the loop will repeat `n_times` times. Every time the loop variable takes on a different value, the statements inside the code block are executed.

16.3.2. Unrolling loops

You can also think of loops in a different way. Listing 16.3 shows how to unroll a loop (write the repeated steps) to see exactly how Python executes the code in listing 16.2. In listing 16.3, you see that the variable v is assigned a different value. The lines that print the variable are the same for every different value of v. This code is inefficient, boring, and error-prone to write because the line to print the value of the variable v is repeated. Using loops instead of this code is much more efficient to write and easier to read.

Listing 16.3. Unrolled for loop from listing 16.2

```
v = 0                               1
print("var v is", v)
v = 1                               2
print("var v is", v)
v = 2                               3
print("var v is", v)
```

- *1* **The variable v (assigned to 0 here) is the loop variable from listing 16.2.**
- *2* **Manually change the value of v to be 1.**
- *3* **Manually change the value of v to be 2.**

SUMMARY

In this lesson, my objective was to teach you why loops are useful. You saw what a for loop does and how to set up a for loop in code. At a high level, a for loop repeats statements that are part of its code block a certain number of times. A loop variable is a variable whose value changes with every loop repetition going through items in the loop sequence.

Sequences can be a series of integers. You saw a special sequence created by the expression `range(N)`, where N is an integer. This expression creates the sequence 0, 1, 2, ... N – 1. Here are the major takeaways:

- Loops are useful for writing concise and easy-to-read code.
- A for loop uses a loop variable that takes on values from a sequence of items; the items can be integers.
- When the items in the sequence are integers, you can use a special range expression to create special sequences.

Let's see if you got this...

Q16.1

Write a piece of code that asks the user for a number. Then write a loop that iterates that number of times and prints Hello every time. Is it possible to write this code without using a for loop?