PREV
Lesson 16. Repeating tasks with loops

NEXT
.esson 18. Repeating tasks while conditions hold

## Lesson 17. Customizing loops

After reading lesson 17, you'll be able to

- Write more-complicated `for` loops that start and end at custom values
- Write loops that iterate over strings

You write programs so you can make the user's life easier in some way, but that doesn't mean that the programmer's experience with writing a program should be tedious. Many programming languages have added customizations to certain language constructs so that a programmer can take advantage of them and write code more efficiently.

**Consider this**

You give your spouse a list of movies you want to watch over the course of one year. Every odd-numbered movie is action, and every even-numbered movie is comedy:

- What pattern can you follow to reliably make sure you watch every comedy movie in the list?
- What pattern can you follow to reliably make sure you watch every action movie in the list?

Answer:

- Go through the list and watch every other movie, starting with the second in the list.
- Go through the list and watch every other movie, starting with the first in the list.

### 17.1. CUSTOMIZING LOOPS

You can specify starting values, ending values, and step sizes when using the `range` keyword. `range(start,end,step)` takes at least one number and can take up to three numbers in its parentheses. The numbering rules are similar to indexing into strings:

- The first number represents the index value at which to start.
- The middle number represents the index value at which to stop, but for which the code won't be executed.
- The last number represents a step (every "how many numbers to skip").

You can remember the following rules of thumb:

- When you give only one number in the parentheses, this corresponds to the end in `range(start,end,step)`. The `start` is by default `0`, and the `step` is by default `1`.
- When you give only two numbers in the parentheses, this corresponds to the `start` and `end` (in that order) in `range(start,end,step)`. The `step` is `1` by default.
- When you give all three numbers in the parentheses, this corresponds to `start`, `end`, and `step` (in that order) in `range(start,end,step)`.

Here are examples of using `range` and the sequence of values to which each corresponds:

- `range(5)` is equivalent to `range(0, 5)` and `range(0,5,1)`—0, 1, 2, 3, 4
- `range(2,6)`—2, 3, 4, 5
- `range(6,2)`—No values
- `range(2,8,2)`—2, 4, 6

- `range(6,2,-1)`—6, 5, 4, 3

**Quick check 17.1**

To what sequence of values do the following expressions evaluate? If you want to check yourself in Spyder, write a loop that iterates over the ranges and prints the value of the loop variable:

**1**

`range(0,9)`

**2**

`range(3,8)`

**3**

`range(-2,3,2)`

**4**

`range(5,-5,-3)`

**5**

`range(4,1,2)`

A `for` loop can iterate over any sequence of values, not just numbers. For example, a string is a sequence of character strings.

## 17.2. LOOPING OVER STRINGS

Recall that a loop variable successively takes on the value of each item in a sequence with each iteration. A loop variable iterating over the numbers 0, 1, 2 takes on the value `0` the first time through the loop, then the value `1` the second time through the loop, and then the value `2` the third time through the loop. If you iterate over a sequence of characters in a string, then instead of a sequence 0, 1, 2, you may have a sequence a, b, c. A loop variable iterating over the sequence of characters will take on the value `a` the first time through the loop, the value `b` the second time through the loop, and the value `c` the third time through the loop.

In section 16.2, you saw the `in` keyword used in the context of `for` loops. There, the `in` keyword was used when you wanted to iterate over a sequence of values; in section 16.2, the values were numbers from 0 to N. The `in` keyword can also be used to iterate over characters in a string, as in the following listing. Given a string, say `"abcde"`, you can think of it as made up of a sequence of string characters `a`, `b`, `c`, `d`, `e`.

**Listing 17.1. A `for` loop that iterates over each character in a string**

```
for ch in "Python is fun so far!":                1
    print("the character is", ch)                 2
```

- ***1* ch is the loop variable.**
- ***2* Prints the loop variabl**

Any string you create has a set length, so a `for` loop that iterates over every character in a string will repeat however many times the length of the string is. In listing 17.1, `ch` is the loop variable, and this can be named any legal Python variable name. The length of the string you're iterating over is 21, because spaces and punctuation also count as characters. The `for` loop in listing 17.1 repeats 21 times; each time, the variable `ch` will take on the value of every different character in the string `"Python is fun so far!"`. Inside the code block, you're printing the value of the loop variable to the Python console.

**Quick check 17.2**

**Q1:**

Write a piece of code that asks the user for input. Then write a loop that iterates over every character. The code prints `vowel` every time it encounters a vowel character.

The method depicted in listing 17.1 is an intuitive way to iterate over every

If Python didn't allow you to iterate over the characters in a string directly, you'd have to use a loop variable that iterates over a sequence of integers representing the position of every character, 0, 1, 2, ... to the length of the string minus 1. Listing 17.2 shows a rewrite of listing 17.1 using this technique. In listing 17.2, you must create a variable for the string so you can later access it inside the loop code block. The loop still repeats 21 times, except now the loop variable takes on values 0, 1, 2,...20 to represent every index location in the string. Inside the code block, you must index into your string variable to find the value of the character at each index. This code is cumbersome and not nearly as intuitive as listing 17.1.

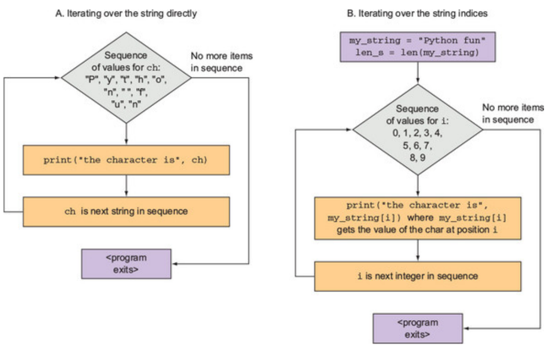
**Listing 17.2. A `for` loop that iterates over each index in a string**

```
my_string = "Python is fun so far!"
len_s = len(my_string)
for i in range(len_s):
    print("the character is", my_string[i])
```

- ***1* Stores string and its length in a variable**
- ***2* Iterates between 0 to len_s - 1**
- ***3* Indexes into the string**

Figure 17.1 shows a flowchart of listing 17.1 (on the left) and listing 17.2 (on the right). When you iterate over the characters directly, the loop variable gets the value of each character in the string. When you iterate over the indices, the loop variable gets the value of integers 0 to the length of the string minus 1. Because the loop variable contains integers, you have to use the integer to index into the string to retrieve the value of the character at that position. This is an extra step calculated using `my_string[i]`. Notice that you have to keep track of a lot more things in listing 17.2, whereas in listing 17.1 the loop variable already knows the value of the character directly.

**Figure 17.1. Comparison of the flowcharts for (A) listing 17.1 and (B) listing 17.2. In (A), the loop variable `ch` takes on the values of each character. In (B), the loop variable takes on integer values representing the indices in a string 0 to the length of the string minus 1. In (B) there's an extra step inside the body of the loop to convert the loop index into the character at that loop index.**


A. Iterating over the string directly
Sequence of values for ch: "P", "y", "t", "h", "o", "n", " ", "f", "u", "n"
No more items in sequence
print("the character is", ch)
ch is next string in sequence
<program exits>
B. Iterating over the string indices
my_string = "Python fun"
len_s = len(my_string)
Sequence of values for i: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
No more items in sequence
print("the character is", my_string[i]) where my_string[i] gets the value of the char at position i
i is next integer in sequence
<program exits>

### Thinking like a programmer

Writing more lines of code or code that looks complicated doesn't make you a better programmer. Python is a great language to start with because it's easy to read, so write your code to follow this idea. If you find yourself writing convoluted logic to achieve a simple task or repeating yourself several times, take a step back and use a piece of paper to draw out what you want to achieve. Use the internet to see whether Python has an easy way to do what you want to do.

## SUMMARY

In this lesson, my objective was to introduce you to sequences that can be a series of integers. You saw that the expression `range` can be customized by changing the sequence start value or end value, or even skipping over numbers. Sequences can also be a series of string characters. You saw how to write code that takes advantage of the capability to iterate over string characters directly. Here are the major takeaways:

- A `for` loop uses a loop variable that takes on values from a sequence of items; the items can be integers or character strings.
- When the items in the sequence are integers, you can use a special `range` expression to create special sequences.
- When the items in the sequence are string characters, the loop variable iterates over the characters in a string directly as opposed to using the index of the string as a middleman.

Let's see if you got this...

Write a program that iterates over all even numbers between 1 and 100. If the number is also divisible by 6, increment a counter. At the end of your program, print how many numbers are even and also divisible by 6.

**Q17.2**

Write a program that asks the user for a number, `n`. Then use loops to repeatedly print a message. For example, if the user inputs `99`, your program should print this:

```
99 books on Python on the shelf 99 books on Pythor
Take one down, pass it around, 98 books left.
98 books on Python on the shelf 98 books on Pythor
Take one down, pass it around, 97 books left.
```

... < and so on >

```
1 book on Python on the shelf 1 book on Python
Take one down, pass it around, no more books!
```

**Q17.3**

Write a program that asks the user to input names separated by a single space. Your program should print a greeting for every name entered, separated by a newline. For example, if the user enters `Zoe Xander Young`, your program prints out `Hi Zoe` and then on the next line `Hi Xander` and then on the next line `Hi Young`. This problem is a bit more involved. Think back to what you learned about strings; you'll have to use a loop to look at every character in the input and save what you see up to a space in a variable representing the name. Don't forget to reset your name variable when you see the space!

PREV
Lesson 16. Repeating tasks with loops

NEXT
Lesson 18. Repeating tasks while conditions hold