CSI 1401 Week 7 Resources

by Allen Yan

This week, CSI 1401 will finish the remaining parts of chapter 5, which introduces the for loop and some more advanced topics of looping. Compared to the while loop covered last week, the for loop is more frequently seen in Python programming due to the amount of convenient short-hands it can use.

As a reminder, I will be leading a group tutoring session on CSI 1401 every Wednesday from 7:00 pm to 8:00 pm (central time). The session will be conducted online via Microsoft Meetings, where I will be available to provide interactive help to students. If the above time window does not work for you, or if you need additional help, Baylor's Tutoring Center also provides 1-on-1 online tutoring appointments. For more information on signing up for group tutoring or individual tutoring, please visit https://www.baylor.edu/tutoring.

Week 7 Important Topics

Chapter 5 – Looping (Continued)

- For loops
- Range function
- Break, continue, loop else

1. For loops

For loops is a very powerful loop structure in Python. It is very convenient for looping over a container, or iterating through a range of numbers. The syntax for the for loop works as follows:

```
for loop_variable in container_or_range:
    # loop body here
```

While loops could be used in two ways depending on the situation. One of them is simply looping through a range of numbers. This is usually the more verbose way, especially when the for loop is used to iterate through a container. See example below:

```
my_list = ['a', 'b', 'c']

for i in range(len(my_list)):
    print(my_list[i])
```

The `range()` function shown above returns a series of integers ranging from 0 to 1 less than the length of the list. More details about the range function will be covered in the next section. Sometimes looping by a range of numbers is the only option available, and in those situations, it is perfectly fine to use that approach. But for looping through containers, there is a more convenient short-hand that does the same thing:

```
my_list = ['a', 'b', 'c']

for element in my_list:
    print(element)
```

The `element` variable in the above code is a temporary variable managed by the for loop, and it automatically updates to the value of the next element inside the container for each iteration of the loop. This method could save you a bit of work on long programs and make your code more readable. However, be aware of the fact that this method of iterating through a container does not allow you to change the contents of the original container. For example, if you execute the code below, you will notice the contents of `my_list` did not change:

```
my_list = ['a', 'b', 'c']

for element in my_list:
    element = 'd'

for element in my_list:
    print(element)
```

This is due to the fact that when you structure a for loop to directly access the contents of a container as the loop variable, Python returns a copy of the value of the items in the list instead of the actual items themselves. If you need to modify the contents of a container, then you have to use the number-based looping approach, or use the `enumerate()` function, which will return both the index and the element of the container. I have included both approach in the example below:

```
my_list = ['a', 'b', 'c']

# number-based
for i in range(len(my_list)):
    my_list[i] = 'd'

for element in my_list:
    print(element)

# enumerate approach
for index, element in enumerate(my_list):
    my_list[index] = 'e'
    # element does not update right away
    print(element)

# now the updates will show
for element in my_list:
    print(element)
```

The above example also shows that the `element` variable is only updated once at the beginning of each iteration of the loop, so if the content of the list is changed inside an iteration, the `element` variable will not reflect that change within the same iteration.

For loops are very convenient compared to while loops, but does not completely replace the latter. It is important to be able to make the judgement call of which loop to use for different programs. Personally, my rule of thumb is that if the loop is iterating through a container, or if the number of iterations is known, then use for loop, otherwise use while loop. The course's textbook offers these advices, which are also good:

1. *Use a for loop* when the number of iterations is computable before entering the loop, as when counting down from X to 0, printing a string N times, etc.
2. *Use a for loop* when accessing the elements of a container, as when adding 1 to every element in a list, or printing the key of every entry in a dict, etc.
3. *Use a while loop* when the number of iterations is not computable before entering the loop, as when iterating until a user enters a particular character.

2. Range function

The range function is useful for generating a consecutive range of integers. Which makes it very useful when paired with loops, but it can also be used on its own. Aside from the use case shown in for loops, the book also provides the following ways to use the function:

Range	Generated sequence	Explanation
`range(5)`	0 1 2 3 4	Every integer from 0 to 4.
`range(0, 5)`	0 1 2 3 4	Every integer from 0 to 4.
`range(3, 7)`	3 4 5 6	Every integer from 3 to 6.
`range(10, 13)`	10 11 12	Every integer from 10 to 12.
`range(0, 5, 1)`	0 1 2 3 4	Every 1 integer from 0 to 4.
`range(0, 5, 2)`	0 2 4	Every 2nd integer from 0 to 4.
`range(5, 0, -1)`	5 4 3 2 1	Every 1 integer from 5 down to 1
`range(5, 0, -2)`	5 3 1	Every 2nd integer from 5 down to 1

3. Break, continue, loop else

When programming with loops, sometimes you may run into cases where a certain iteration meets a special condition that will require the current iteration of the loop to be skipped, or in some cases even stop the entire loop. The `break` and `continue` keywords will help you in those situations. The `break` keyword will end the loop immediately, and the `continue` keyword will skip to the next iteration of the loop. Note that if used in nested loops, both `break` and `continue` will only apply to the layer of loop they are called on.

Python loops also come with the unique functionality of attaching an `else` statement to its end if the user elects to implement it. The loop else functions similarly to the else statement used in if statements: the else statement will execute once when the loop terminates. Loop else is fairly situational, because most of the time the user can get the same functionality by simply putting the same code after a loop without the else statement. The one catch with loop else statements is that it only executes if the loop terminates normally, and not by the `break` keyword.

Useful Resources

- Basic Python Tutorial on GeeksforGeeks:
  https://www.geeksforgeeks.org/python-programming-language/
    - This page provides links to detailed explanations to many entry-level Python concepts along with examples. I encourage taking a look at it if the examples in your textbook were not clear enough.
- Official Python Documentation:
  https://docs.python.org/3/
    - This may be a bit heavy-handed for a beginner-level programmer since the official Python documentation is very thorough and technical. However, learning how to read official documentations is crucial to becoming a good programmer, because the official documentation contains information on everything you can find about Python elsewhere and more. Therefore, I encourage slowly easing yourself into learning how to read the official documentation.