Debugging Loops

by Sophia

WHAT'S COVERED

This lesson focuses on debugging issues that can occur with creating and using loops. Specifically, this lesson covers:

1. Infinite Loop

An endless source of amusement for programmers is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop because there is no iteration variable telling you how many times to execute the loop.

In the case of a countdown scenario, we can prove that the loop terminates because we know that the counter value that we use will be finite, and we can see that the counter value gets smaller each time through the loop, so eventually we have to get to 0. Other times, a loop is obviously infinite because it has no iteration variable at all.

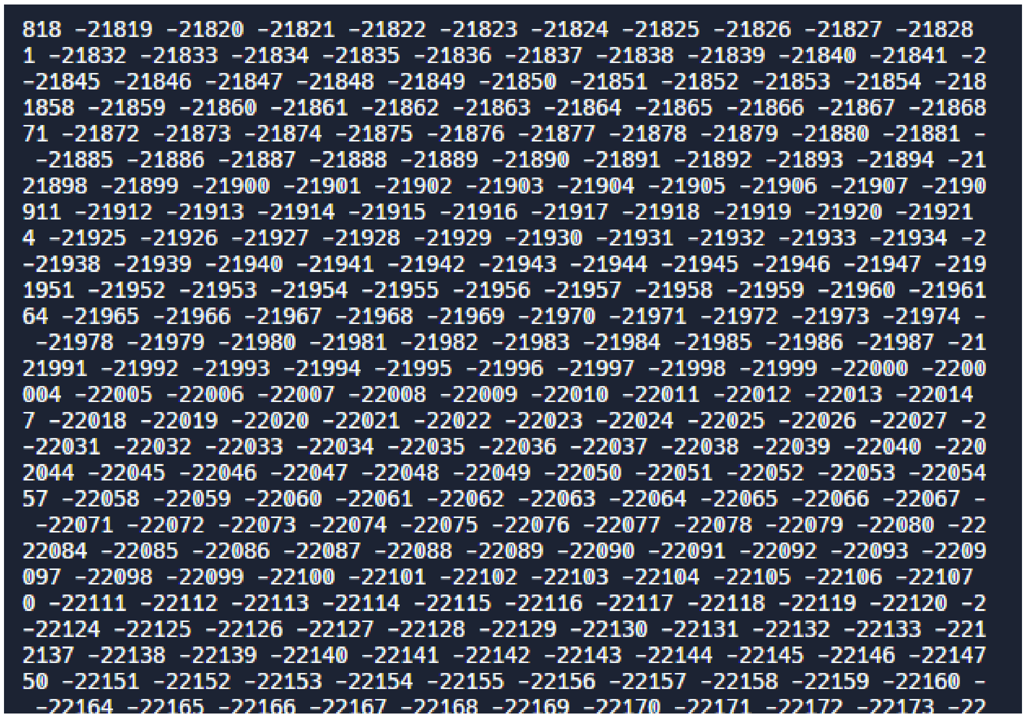
Sometimes you don’t know it’s time to end a loop until you get halfway through the body of the loop. In that case, you can write an infinite loop on purpose and then use the `break` statement to jump out of the loop.

This loop below is obviously an infinite loop because the logical expression on the `while` loop will always evaluate as being less than 20, so it will constantly run without breaking.

EXAMPLE

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Done!')
```

The output is an infinite loop.



```
818 -21819 -21820 -21821 -21822 -21823 -21824 -21825 -21826 -21827 -21828
1 -21832 -21833 -21834 -21835 -21836 -21837 -21838 -21839 -21840 -21841 -2
-21845 -21846 -21847 -21848 -21849 -21850 -21851 -21852 -21853 -21854 -218
1858 -21859 -21860 -21861 -21862 -21863 -21864 -21865 -21866 -21867 -21868
71 -21872 -21873 -21874 -21875 -21876 -21877 -21878 -21879 -21880 -21881 -
 -21885 -21886 -21887 -21888 -21889 -21890 -21891 -21892 -21893 -21894 -21
21898 -21899 -21900 -21901 -21902 -21903 -21904 -21905 -21906 -21907 -2190
911 -21912 -21913 -21914 -21915 -21916 -21917 -21918 -21919 -21920 -21921
4 -21925 -21926 -21927 -21928 -21929 -21930 -21931 -21932 -21933 -21934 -2
-21938 -21939 -21940 -21941 -21942 -21943 -21944 -21945 -21946 -21947 -219
1951 -21952 -21953 -21954 -21955 -21956 -21957 -21958 -21959 -21960 -21961
64 -21965 -21966 -21967 -21968 -21969 -21970 -21971 -21972 -21973 -21974 -
 -21978 -21979 -21980 -21981 -21982 -21983 -21984 -21985 -21986 -21987 -21
21991 -21992 -21993 -21994 -21995 -21996 -21997 -21998 -21999 -22000 -2200
004 -22005 -22006 -22007 -22008 -22009 -22010 -22011 -22012 -22013 -22014
7 -22018 -22019 -22020 -22021 -22022 -22023 -22024 -22025 -22026 -22027 -2
-22031 -22032 -22033 -22034 -22035 -22036 -22037 -22038 -22039 -22040 -220
2044 -22045 -22046 -22047 -22048 -22049 -22050 -22051 -22052 -22053 -22054
57 -22058 -22059 -22060 -22061 -22062 -22063 -22064 -22065 -22066 -22067 -
 -22071 -22072 -22073 -22074 -22075 -22076 -22077 -22078 -22079 -22080 -22
22084 -22085 -22086 -22087 -22088 -22089 -22090 -22091 -22092 -22093 -2209
097 -22098 -22099 -22100 -22101 -22102 -22103 -22104 -22105 -22106 -22107
0 -22111 -22112 -22113 -22114 -22115 -22116 -22117 -22118 -22119 -22120 -2
-22124 -22125 -22126 -22127 -22128 -22129 -22130 -22131 -22132 -22133 -221
2137 -22138 -22139 -22140 -22141 -22142 -22143 -22144 -22145 -22146 -22147
50 -22151 -22152 -22153 -22154 -22155 -22156 -22157 -22158 -22159 -22160 -
 -22164 -22165 -22166 -22167 -22168 -22169 -22170 -22171 -22172 -22173 -22
```

If you make the mistake and run this code, you will learn quickly how to stop a runaway Python process on your system or find where the power-off button is on your computer. In the IDE, you can click on the Stop button where the Run button is. This program will run forever or until your battery runs out because the logical expression at the top of the loop will always be less than 20.

We could have done the same thing by creating a condition that is always true without using the True value. Since `n` is starting at 10 and the value is decrementing, in the example below, `n` will always be < 20 so it will loop infinitely in the same way that a True logical constant would.

EXAMPLE

```
n = 10
while n < 20:
    print(n, end=' ')
    n = n - 1
print('Done!')
```

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using a `break` statement when we have reached the exit condition where the loop condition evaluates to False.

For example, suppose you want to take input from the user until they type “done”. You could write the following.

EXAMPLE

```
while True:
    line = input('Enter input or type done when finished > ')
    if line == 'done':
        break
    print(line)
print('Done!')
```
The loop condition is True, which is always true, so the loop runs repeatedly until it hits the `break` statement.

Each time through, the body of the loop prompts the user with directions and an angle bracket. If the user types "done", the `break` statement exits the loop. Otherwise, the program prints out whatever the user types and goes back to the top of the loop. Here's a sample run.

```
Enter input or type done when finished > a
a
Enter input or type done when finished > b
b
Enter input or type done when finished > c
c
Enter input or type done when finished > d
d
Enter input or type done when finished > done
Done!
```

TRY IT

Directions: Try adding the code above and run the program a few times, testing out the word "done" to exit.

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the `continue` statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that prints out its user input until the user types "done", but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

EXAMPLE

```
while True:
    line = input('Enter input (# tag will not print). Type done when finished > ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

Here is a sample run of this new program with the `continue` statement added.

```
Enter input (# tag will not print). Type done when finished > Hello
Hello
Enter input (# tag will not print). Type done when finished > # Don't print this
Enter input (# tag will not print). Type done when finished > Print this
Print this
Enter input (# tag will not print). Type done when finished > done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the `continue` statement is executed, it ends the current iteration and jumps back to the beginning of the `while` loop to start the next iteration, thus skipping the `print()` function statement.

TRY IT

Directions: Go ahead and try the program with the `continue` statement. Notice that the program will only check if the user input starts with a hashtag. Using the symbol later in the sentence will not have the same effect.

BIG IDEA

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to "hide." One way to cut your debugging time is "debugging by bisection." For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Debugging by bisection is the practice of breaking your program code into "sections" for debugging and validation purposes. This will help speed up the debugging process since you may be able to find issues within a section rather than the full program, therefore shortening the time to debug if no issues are present in earlier sections.

Try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value, you can check. Add a print statement (or something else that has a verifiable effect) and run the program. If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice, it is not always clear what the "middle of the program" is, and not always possible to check it. It doesn't make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

Debugging by bisection is especially important if you've made changes to an existing program. You most likely will not need to test the whole program prior to where the change is made, but only on everything afterward.

TERM TO KNOW

Debugging by Bisection
Debugging by bisection is the practice of breaking your program code into "sections" for debugging and validation purposes. This will help speed up the debugging process since you may be able to find issues within a section rather than the full program.

2. Other Common Issues

It's important to note that we want to always ensure the following when using loops:

1. We're iterating through iterable objects like lists, tuples, dictionaries, or sets. Python will not be able to iterate over single elements like integers.
2. It's also important that all variables that we create and use as part of the loop's condition are initialized before the loop. If we don't even have it declared, we'll have a separate error to indicate that the variable isn't defined yet.
3. When using the `range()` function in `for` loops, it's important to note that the upper limit (end number) of the `range()` is not included as part of the execution.

EXAMPLE

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

The output:

```
1
2
3
4
```

If we wanted 5 to be in the output, we would also need to have that as part of the range.

EXAMPLE

```
for num in range(1,6):
  print(num)
```

The output:

```
1
```

```
2
3
4
5
```

TRY IT

Directions: Go ahead and try running this program and change the `range()` function to see how that function excludes the end number.

SUMMARY

In this lesson, we learned about **infinite loops** and why they are such an issue since they will not stop running unless you have a way to stop the process, like the stop button in the IDE. It is best practice to always create a loop that has some means of being able to exit. In most cases, infinite loops are errors that should be resolved. We also learned about **other common issues** when using loops. Always make sure you are iterating through iterable objects like the data collection types we have learned about. It is important that all variables we use as part of the loop's condition are initialized before the loop. And finally, we learned to keep in mind that when using the `range()` function, the end number will never be obtained.

Best of luck in your learning!

TERMS TO KNOW

Debugging by Bisection
Debugging by bisection is the practice of breaking your program code into "sections" for debugging and validation purposes. This will help speed up the debugging process since you may be able to find issues within a section rather than the full program.