Loops and Other Stuff[1]

Robin Dawes

September 13, 2021

[1] Like a Circle in a Spiral...

Looks at loops and a few important Python features.

.

While Loops

Like most modern programming languages, Python provides us with two forms of loops: while loops and for loops.

It is not actually necessary to have two different types of loop, since while loops can be used to replicate the actions of for loops. For that matter, it is not actually necessary to have loops at all: the Haskell language does not have loops as a built-in feature.

While loops can also be called conditional loops - the loop executes as long as some Boolean (true/false) condition is true.

Here's a very simple example:

```
n = 100
while (n > 1):
   print(n)
   if (n % 2 == 0):
      n = n // 2
   else:
      n = 3*n + 1
```

You might want to run that code to see the sequence of integers it prints. This example illustrates one of the most famous unsolved questions in mathematics. The **Collatz Conjecture** is:

Not one of the most important problems, just one of the most famous.

Let n be any positive integer. The Collatz sequence for n is created by repeatedly applying the following rule to the current value of n :

If n is even, let $n = \frac{n}{2}$ and if n is odd, let $n = 3n + 1$.

We conjecture that $\forall$ positive values of n, the Collatz sequence for n contains the value 1

Example: Here is the Collatz sequence for $n = 7$, up to the first occurrence of 1

7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1

It has been shown that the Collatz Conjecture is true for millions of integers ... but nobody has been able to prove that it is true for **all** integers. Nor has anyone found an integer for which the Collatz sequence goes on forever without ever containing 1.

This is a perfect illustration of when we should use a while loop: whenever we don't know how many times we need the loop to execute, and the termination is based on a logical condition.

For Loops

For loops, which are also referrred to as iterated loops, are typically used when we need to repeat some action a known number of times, or we need to repeat the action on every element of a list or other indexed structure (Python uses the word "iterable" for such structures - we will discuss this word soon.)

For example, we may wish to print 12 consecutive blank lines. We would use a for loop to execute the print statement 12 times.

Another example: we would use a for loop to find the largest value in a set of numbers.

A for loop usually uses an index variable which is assigned a new value each time the loop repeats. The loop executes until the index variable reaches some specific value.

Many languages use a three-part header to define for loops, often looking something like this

```
for (i = 1; i <= 12; i++)
   {body of loop}
```

which means "create an integer variable called i, initialize it to 1, loop until i is > 12, and increment i by 1 each time"

Python does it differently. In Python we specify the full set of values for the variable to take, in order. So a Python for loop might look like this:

```
for i in [7,45,3,82]:
     body of loop
```

This loop would execute four times. The first time through, i would have the value 7, the next time through i would equal 45, etc.

To recreate the effect of the standard "increment by 1" for loop, Python provides a built-in function called **range**. The instruction

```
range(n)
```

creates a list that looks like this:

```
[0,1,2,3,..., n-1]
```

where the ... is just my shorthand for "all the integers between 3 and n-1". The actual Python list doesn't have three dots in it!

The **range** function has more options ... you can specify the start value, and the step size. We'll discuss those options if we need them, or you can look them up.

Now, about that "0 to n-1" instead of "1 to n" thing. Like most (but not all) programming languages Python uses "0-based addressing". In any indexed object (such as a list or string) the first position is considered to be position 0. So if the list has n elements, the last one is in position n-1. Since we *very* often use a for loop to iterate through all the positions in a list, it does make sense for the **range** function to create the list of index values that are needed.

This unnatural way of addressing is a perfect example of making humans adapt to technology rather than designing technology to fit with humans. I've been against it for 50 years ... but there's not much hope of my opposition having any effect!

Of course Python lists can change in size (we have already seen the **append** function in some of our examples). We can use the built-in **len** function to find the number of elements currently in a list. A for loop to print all the values in a list might look like this:

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

and that's pretty concise.

But remember that the Python for loop just needs a list of all the values the index variable is to take. In the example just presented, *my_list* is a list of values. In this loop, we don't really need to know the exact position of the values in the list, we just need the values. So we can just use *my_list* as the list of values, and use the index variable to take on those values one after the other.

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

and that's brilliant. It doesn't introduce an unnecessary index variable, it doesn't generate a list of positions, and it's immediately obvious what we want it to do. This is a triumph of language design.

Lists and Tuples

WE WILL TAKE a detailed look at Python lists in an upcoming class, but here's a very brief overview.

Python lists are a blend of what other languages call arrays and linked lists. Like an array, a Python list is indexed ... so we can access any position directly using **list_name[i]**. But like a linked list, a Python list can be extended to any length, new values can be added at any position in the list, and existing values can be deleted from the list. The elements in a Python list can be of any type at all, and objects of different types can occupy the same list. So a Python list is a very flexible thing.

Java has a structure called an **arraylist** which is similar.

We will talk later about the cost we pay for this flexibility ... dark foreshadowing!

Python **tuples** look a lot like lists. The visual difference is that a Python list is defined with square brackets "[" and "]", where as a Python tuple is defined with round brackets "(" and ")". So [1,2,3] is a list with three elements, and (1,2,3) is a tuple with exactly the same three elements. Like lists, tuples are indexed objects so we can use something like this:

```
some_numbers = (1,2,3,4)
x = some_numbers[2]
```

The important difference is that a Python tuple is an IMMUTABLE object, which means we cannot modify its value. We can't append, insert or delete values from a tuple, and we can't assign new values to the elements of the tuple.

So whereas something like

```
a_list = [1,2,3,4]
a_list[1] = 37.8
```

is legal, this

```
a_tuple = (1,2,3,4)
a_tuple[1] = 37.8
```

is not.

This may seem strange and useless, but it's not! There are several very good reasons why - in some circumstances - the immutability of tuples is exactly what we want. I'll let you think about what those reasons might be.