Set 12: Loops Part 1

Skill 12.1: Provide an example of definite and indefinite iteration

Skill 12.1 Concepts

In our everyday lives, we tend to repeat a lot of processes without noticing. For instance, if we want to cook a delicious recipe, we might have to prepare our ingredients by chopping them up. We chop and chop and chop until all of our ingredients are the right size. At this point, we stop chopping. If we break down our chopping task into a series of three smaller steps, we have,

1. An *initialization*: We're ready to cook and have a collection of ingredients we want to chop. We will start at the first ingredient.
2. A *repetition*: We're chopping away. We are performing the action of chopping over and over on each of our ingredients, one ingredient at a time.
3. An *end condition*: We see that we have run out of ingredients to chop and so we stop.

In programming, this process of using an *initialization*, *repetitions*, and an *ending* condition is called a loop.

Programming languages like Python implement two types of iteration,

1. *Indefinite iteration*, where the number of times the loop is executed depends on how many times a condition is met.
2. *Definite iteration*, where the number of times the loop will be executed is defined in advance (usually based on the collection size).

Typically, we will find loops being used to iterate a collection of items. In the above example, we can think of our ingredients we want to chop as our collection. This is a form of definite iteration since we know how long our collection is in advance and thus know how many times we need to iterate over the collection of ingredients.

Some collections might be small — like a short sting, while other collections might be massive like a range of numbers from 1 to 10,000,000! But don't worry, loops give us the ability to masterfully handle both ends of the spectrum. This simple, but powerful, concept saves us a lot of time and makes it easier for us to work with large amounts of data.

Skill 12.1 Exercise

Skill 12.2: Explore the purpose of loops

Skill 12.2 Concepts

Before we get to writing our own loops, let's explore what programming would be like if we couldn't use loops.

Let's say we have a list of ingredients and we want to print every element in the list,

Code

```
ingredients = ["milk", "sugar", "vanilla extract", "dough", "chocolate"]
print(ingredients[0])
print(ingredients[1])
print(ingredients[2])
print(ingredients[3])
print(ingredients[4])
```

Output

```
milk
sugar
vanilla extract
dough
chocolate
```

That's still manageable. We're writing 5 *print()* statements (or copying and pasting a few times). Now imagine if we come back to this program and our list had 10, or 24601, or … 100,000,000 elements? It would take an extremely long time and by the end, we could still end up with inconsistencies and mistakes.

Skill 12.2 Exercise

Skill 12.3: Interpret *for* loops

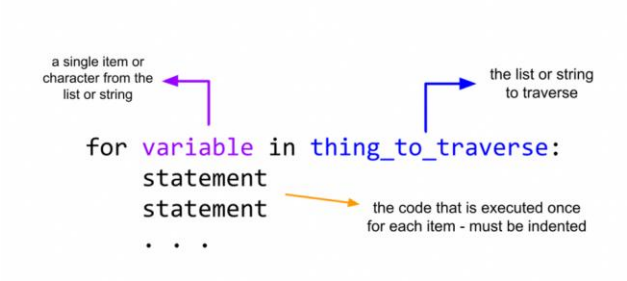
Skill 12.3 Concepts

Now that we can appreciate what loops do for us, let's start with your first type of loop, a *for* loop, a type of *definite* iteration.

In a *for* loop, we will know in advance how many times the loop will need to iterate because we will be working on a collection with a predefined length. In our examples, we will be using Python lists as our collection of elements.

With *for* loops, on each iteration, we will be able to perform an action on each element of the collection.

Before we work with any collection, let's examine the general structure of a *for* loop,



- We start by writing the keyword *for* followed by a variable name that we want to use to refer to each item we are traversing.
- Then we write the keyword *in* followed by the name of the list or string we want to traverse.
- The colon (:) operator is used to specify an indented block of code.

The code we want to execute is indented under the *for* loop header. This code is executed once for each item in the list or string we are traversing.

Let's link these concepts back to our ingredients example. This *for* loop prints each ingredient in ingredients:

Code

```
ingredients = ["milk", "sugar", "vanilla extract", "dough", "chocolate"]


for ingredient in ingredients:
  print(ingredient)
```

Output

```
milk
sugar
vanilla extract
dough
chocolate
```

Some things to note about *for* loops,

Temporary Variables

A temporary variable's name is arbitrary and does not need to be defined beforehand. Both of the following code snippets do the exact same thing as our above example,

```
for i in ingredients:
   print(i)


for item in ingredients:
  print(item)
```

Programming best practices suggest we make our temporary variables as descriptive as possible. Since each iteration (step) of our loop is accessing an ingredient it makes more sense to call our temporary variable ingredient rather than i or item.

Indentation

Notice that in all of these examples the print statement is indented. Everything at the same level of indentation after the *for* loop declaration is included in the loop body and is run on every iteration of the loop.

```
for ingredient in ingredients:
  # Any code at this level of indentation
  # will run on each iteration of the loop
  print(ingredient)
```

If we ever forget to indent, we'll get an *IndentationError* or unexpected behavior.

Skill 12.3 Exercises 1 and 2

Skill 12.4: Apply the *range()* function to *for* loops

Skill 12.4 Concepts

Often, we won't be iterating through a specific list (or any collection), but rather only want to perform a certain action multiple times.

For example, if we wanted to print out a "Learning Loops!" message six times using a for loop, we would follow this structure,

```
for <temporary variable> in <list of length 6>:
  print("Learning Loops!")
```

Notice that we need to iterate through a list with a length of six, but we don't necessarily care what is inside of the list.

To create arbitrary collections of any length, we can pair our for loops with Python's built in *range()* function.

Below is an example of how we can use the range function in loops,

Code

```
six_steps = range(6)
```
Creates a range object with values 0 up to 6
-> (0, 1, 2, 3, 4, 5)

```
for temp in range(6):
  print("Learning Loops!")
```
For each value in the range we print Learning Loops!

Output

```
Learning Loops!
Learning Loops!
Learning Loops!
Learning Loops!
Learning Loops!
Learning Loops!
```

Something to note is we are not using temp anywhere inside of the loop body. If we are curious about which loop iteration (step) we are on, we can use *temp* to track it. Since our range starts at 0, we will add + 1 to our temp to represent how many iterations (steps) our loop takes more accurately.

Code

```
for temp in range(6):
    print("This is the current index: " + str(temp))
    print("Loop is on iteration number: " + str(temp + 1))
```

Output

```
This is the current index: 0
Loop is on iteration number: 1
This is the current index: 1
Loop is on iteration number: 2
This is the current index: 2
Loop is on iteration number: 3
This is the current index: 3
Loop is on iteration number: 4
This is the current index: 4
Loop is on iteration number: 5
This is the current index: 5
Loop is on iteration number: 6
```

Skill 12.4 Exercise 1

Skill 12.5: Interpret *while* loops

Skill 12.5 Concepts

In Python, *for* loops are not the only type of loops we can use. Another type of loop is called a *while* loop and is a form of indefinite iteration.

A *while* loop performs a set of instructions as long as a given condition is true.

The structure follows this pattern,

```
while <conditional statement>:
     <action>
```

Let's examine this example, where we print the integers 0 through 3,

Code

```
count = 0
while count <= 3:
  # Loop Body
  print(count)
  count += 1
```

Output

```
0
1
2
3
```

Let's break the loop down:

- Count is initially defined with the value of 0. The conditional statement in the while loop is count <= 3, which is true at the initial iteration of the loop, so the loop body executes.
- Inside the loop body, count is printed and then incremented by 1.
- When the first iteration of the loop has finished, Python returns to the top of the loop and checks the conditional again. After the first iteration, count would be equal to 1 so the conditional still evaluates to True and so the loop continues.
- This continues until the count variable becomes 4. At that point, when the conditional is tested it will no longer be True and the loop will stop.

Note the following about *while* loops before we write our own,

Indentation

Notice that in our example the code under the loop declaration is indented. Similar to a *for* loop, everything at the same level of indentation after the *while* loop declaration is run on every iteration of the loop while the condition is true.

```
while count <= 3:
    # Loop Body
    print(count)
    count += 1
    # Any other code at this level of indentation will
    # run on each iteration
```

If we ever forget to indent, we'll get an *IndentationError* or unexpected behavior.

Skill 12.5 Exercises 1 and 2

Skill 12.6: Apply *while* loops to lists

Skill 12.6 Concepts

A *while* loop isn't only good for counting! Similar to how we saw *for* loops working with lists, we can use *while* loops to iterate through a list as well.

Let's return to our ingredient list,

```
ingredients = ["milk", "sugar", "vanilla extract", "dough", "chocolate"]
```

We know that *while* loops require some form of a variable to track the condition of the loop to start and stop.

We can use the Python's built in *len()* function to accomplish this,

```
length = len(ingredients)
```

We can then use this length in addition to another variable to construct the while loop,

Code

```
ingredients = ["milk", "sugar", "vanilla extract", "dough", "chocolate"]


length = len(ingredients)
index = 0


while index < length:
  print(ingredients[index])
  index += 1
```

length is set to 5

index is set to 0

0 < 5 is true so we execute the code block in the while loop

The first iteration will print ingredients[0]

Increments index so we can access the next element.
We then return to the top of the loop and compare the new value of index to the length: 1 < 5 is true so we continue

Output

```
milk
sugar
vanilla extract
dough
chocolate
```

Skill 12.6 Exercise 1

Skill 12.7: Identify (and fix) infinite loops

Skill 12.7 Concepts

We've iterated through lists that have a discrete beginning and end. However, let's consider this example,

```
my_favorite_numbers = [4, 8, 15, 16, 42]
for number in my_favorite_numbers:
  my_favorite_numbers.append(1)
```

Every time we enter the loop, we add a 1 to the end of the list that we are iterating through. As a result, we never make it to the end of the list. It keeps growing forever!

A loop that never terminates is called an *infinite loop*. These are very dangerous for our code because they will make our program run forever and thus consume all of your computer's resources.

A program that hits an infinite loop often becomes completely unusable. The best course of action is to avoid writing an infinite loop.

Note: If you accidentally stumble into an infinite loop while developing on your own machine, you can end the loop by using control + c to terminate the program.

Skill 12.7 Exercises 1 and 2