Loops

Loops refer to structures (namely `for` and `while`) that allow you to repeat code over and over as long you still have a value to process (`for`) or the condition remains `True` (`while`). These two loops, although quite similar, have distinct advantages and disadvantages over each other and of course, are coded differently.

For Loops

`for` loops are loops that iterate over something: in other words, they run as long they have some value to use. As an analogy, imagine you had to clean several items. You would clean each item one by one until you ran out of items to clean. "Cleaning" would be the code to perform and the "items" will be the values provided to the `for` loop.

The basic `for` loop depends on a counter, mainly a generator called `range()`. `range()` takes at least one parameter:

```
range(start, stop, step)
 - start: optional parameter, must be an integer, will be the first number generated (default: 0)
 - stop: mandatory parameter, must be an integer, dictates when the range function stops
 - step: optional parameter, must be an integer, dictates what number to add to the previous number
to generate the upcoming number (default: 1)
```

Mathematically, you can think of `range()` as a function such that it returns a list of numbers within these boundaries: [start, stop). Examples of range are as follows with Example 6-1.

In [ ]:
```
# Example 6-1: range()
print("Output of range(6):\t\t", list(range(6)))
print("Output of range(1, 6):\t\t", list(range(1, 6)))
print("Output of range(1, 6, 2): \t", list(range(1, 6, 2)))
```

```
Output of range(6):             [0, 1, 2, 3, 4, 5]
Output of range(1, 6):          [1, 2, 3, 4, 5]
Output of range(1, 6, 2):       [1, 3, 5]
```

As you can see, `range()` will never output `stop`, but `start` will always appear. Instead of adding 1 every time to get the next number, you can change `step` to be some other number. This is also how you count down instead of up. Note that `step` is always the third number: you cannot specify step by name, so you must always provide a start and stop value.

In [ ]:
```
# Example 6-2: Backwards counting
print("Output of range(10, 0, -1): ", list(range(10, 0, -1)))
```

```
Output of range(10, 0, -1):  [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

By now, hopefully you see that `range()` can help generate numbers needed for the `for` loop to iterate with. The format of a basic `for` loop looks like the following:

```
for loop_variable in range(iterations):
    pass    # Replace with code
```

`loop_varaible` technically can be a pre-existing variable, but it is recommended you use a variable name that hasn't been used (the `for` loop will create the variable for you without needing to assign it a value). In other words, if `loop_variable` does exist, that will be used to store the value from `range()`, but if it does not exist, Python will create a variable *local* to the `for` loop with that name to store the value from `range()`.

A very basic `for` loop will be to count the first `n` numbers. It can look something like Example 6-3.

In [ ]:
```
# Example 6-3: Counting the first n numbers
number = int(input("Enter a number to count to: "))

for i in range(1, number+1):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

So I inputted in "10" and you can see that the numbers from 1-10 are outputted. The reason I put `number+1` as my stop is due to the fact the stop value is *not included* in the list of numbers generated. To include it, I add 1 to it so it does end up being included.

`for` loops will also work for any sort of iterable: anything that you can iterate over (more simply, anything you can index). This means that you can loop through strings and other data structures such as tuples, lists, and dictionaries (Modules 7 and 8).

In [ ]:
```
# Example 6-4: Iterating over a string
name = "Anthony"

for letter in name:
    print(letter)
```

```
A
n
t
h
o
n
y
```

Sometimes this is referred to as a "for-each" loop. In English, you can interpret Example 6-4 as "**for each** letter **in** name, print out the letter". Note that in this case, `letter` doesn't store the index of the letter, it actually just stores the letter itself. This means that this type of loop is generally used only when you don't really care about the index, just the value itself.

If you want both at the same time, then `enumerate()` becomes your best friend:

In [ ]:
```
# Example 6-5: enumerate()

name = "Anthony"

for index, letter, in enumerate(name):
    print(f"Letter \"{letter}\" can be found at index {index}")
```

```
Letter "A" can be found at index 0
Letter "n" can be found at index 1
Letter "t" can be found at index 2
Letter "h" can be found at index 3
Letter "o" can be found at index 4
Letter "n" can be found at index 5
Letter "y" can be found at index 6
```

While Loops

`while` loops are loops that run *while* a condition is met. There are no limitations to how long a `while` loop can run other than the condition itself. The most basic loop taught is actually the infinite loop:

In [ ]:
```
# Example 6-6: Infinite Loop

while True:
    pass
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
c:\Users\antho\Documents\ENGR-102-Fall-2023\Module 6\module_six_notes.ipynb Cell 14 line 3
      <a href='vscode-notebook-cell:/c%3A/Users/antho/Documents/ENGR-102-Fall-2023/Module%206/module_six_notes.ipynb#X16sZmlsZQ%3D%3D?line=0'>1</a> # Example 6: Infinite Loop
----> <a href='vscode-notebook-cell:/c%3A/Users/antho/Documents/ENGR-102-Fall-2023/Module%206/module_six_notes.ipynb#X16sZmlsZQ%3D%3D?line=2'>3</a> while True:
      <a href='vscode-notebook-cell:/c%3A/Users/antho/Documents/ENGR-102-Fall-2023/Module%206/module_six_notes.ipynb#X16sZmlsZQ%3D%3D?line=3'>4</a>     pass

KeyboardInterrupt:
```

As you can see, I had to physically stop the loop because it goes forever. The condition being given is `True`, which is a constant that cannot be changed. That means the code inside will run forever. `pass` is just a placeholder for any indented sections of code. I don't have anything to put inside the `while` loop yet, so I just put in `pass` to avoid any errors.

This being said, infinite loops are what you want to avoid. You want to use a `while` loop when you are not sure of the exact amount of iterations, but you know that you want code to be ran under a certain condition.

For example, let's say I wanted to take input until I get a negative number. Assume all inputs are valid and are numeric. Since I grab input *until* I get a negative number, this means that I grab input *while* my input is non-negative (zero or positive). Remember that prompts generally tell you the stop condition: you need to negate the condition to get your `while` loop condition.

In [ ]:
```
# Example 6-7: Grabbing input until negative number is inputted

number = int(input("Enter a number: "))

while number >= 0:
    number = int(input("Enter a number: "))
```

Also note a very crucial detail: I ask for input *before* the `while` loop is executed. I need an initial condition for the condition to be `True`. Otherwise, Python has no idea what `number` is as it checks the condition first and then executes the code inside.

The way I exit this loop is by inputting a negative number. I ensure this is possible by directly affecting the variable used in the condition

within the `while` loop.

"break" keyword

In some scenarios, you want to immediately break out of the loop (either `for` or `while`). This is generally not ideal since it means you didn't create the `range` properly (for `for` loops) or you didn't correctly form the loop condition (for `while` loops). However, there are always special cases where you want to immediately break the loop. This is available with the `break` keyword.

The second Python executes the `break` keyword, the *immediate* loop it is *inside* of is stopped. This is demonstrated in Example 6-8.

In [ ]:
```
# Example 6-8: using break

for i in range(3):
    for j in range(10):
        if j >= 1:
            break
        print(i, j)
```

```
0 0
1 0
2 0
```

As you can see, the `break` only affects the immediate loop it is inside of. In this case, it will only stop the `for` loop using `j`. The `for` loop that uses `i` does contain the `for` loop using `j`, but `break` belongs to just the `for` loop with `j` because it is directly contained by that loop.

"continue" keyword

Other times, you want to skip the current iteration and go to the next. This is good for when you're processing some type of data, and there are data points that you will want to skip. `continue` lets you skip to the next iteration. Example 6-9 highlights this.

In [ ]:
```
# Example 6-9: We don't like vowels
name = "Anthony"

for letter in name:
    if letter.lower() in "aeiou":
        continue
    print(letter)
```

```
n
t
h
n
y
```

As you can see, all the letters that matched with `"aeiou"` after being sent to lowercase.

"range" in detail

`range()` is an interesting built-in function. For the purposes of ENGR 102, you're told that `range()` gives you a list of numbers... which is almost true but not quite. The differences between the actual type of `range()` and lists (which will be defined in Module 7) are for the most part insignificant. There are some clear differences but at least for this course, as long as you only use them for `for` loops, you'll be able to dodge the intricacies between the two. As said before, `range()` can take up to three parameters, which include:

- `start`: the starting number (included in the numbers, defaults to 0 if not included)
- `end`: the ending number (excluded from the numbers, must be given)
- `step`: what number to be counting by (defautls to 1 if not included)

You can provide one, two, or three numbers for `range()` but depending on how many you give it, it will assume different things about each number:

- 1 number given: the number is the `end`
- 2 numbers given: the first number is the `start` and the second number is the `end`
- 3 numbers given: the first number is the `start`, the second number is the `end`, and the third number is the `end`

Examples of this are shown below:

In [ ]:
```
# Example 6-10: One number only

for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [ ]:
```python
# Example 6-11: Two numbers only

for i in range(1, 10):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

In [ ]:
```python
# Example 6-12: All three given

for i in range(1, 10, 2):
    print(i)
```

```
1
3
5
7
9
```

These three variations are essential to understand. You must know which number represents what type of value in order to properly determine the expected numbers that the `for` loop goes through. This sort of behavior will show up again in Module 7! Also note that Python always assumes you are going from a smaller number to a bigger number unless your step is negative (then you should have a `start` that is larger than your `end`). This is because the step value is assumed to be 1.

In [ ]:
```python
# Example 6-13: Assumed direction is small -> large so having start > end will not work!
# As seen here, no output is shown

for i in range(10, 1):
    print(i)
```

In [ ]:
```python
# Example 6-14: Start > end but the step is negative

for i in range(10, 1, -1):
    print(i)
```

```
10
9
8
7
6
5
4
3
2
```

Nested Loops

You can have a loop in a loop. This means that the inner loop is executed as many times as the outer loop is run. Let's see this in action.

In [ ]:
```python
# Example 6-15: Basic nested loop

for i in range(5):
    for j in range(5):
        print(f"i={i}, j={j}")
```

```
i=0, j=0
i=0, j=1
i=0, j=2
i=0, j=3
i=0, j=4
i=1, j=0
i=1, j=1
i=1, j=2
i=1, j=3
i=1, j=4
i=2, j=0
i=2, j=1
i=2, j=2
i=2, j=3
i=2, j=4
i=3, j=0
i=3, j=1
i=3, j=2
i=3, j=3
i=3, j=4
i=4, j=0
i=4, j=1
i=4, j=2
i=4, j=3
i=4, j=4
```

As you can see, the inner loop (with variable `j`) goes all the way through before incrementing the outer loop (with variable `i`). This is handy when you deal with multi-dimensional data. This becomes more obvious when you deal with lists, but the most basic example that is usually given is the multiplication table: one loop is for the left hand side factor and one loop is for the right hand side factor.

In [ ]:
```python
# Example 6-16: Basic multiplication table

for i in range(1, 5):
    for j in range(1, 5):
        print(f"{i} x {j} = {i*j}")
```

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
```

Neatly enough, you can nest these loops infinitely (I don't recommend this though unless you have extremely compelling reasons to do so). Your nest could be three-deep (as in a loop in a loop in a loop), four-deep, etc. You might also hear these sort of loops be defined by how many "dimensions" (loops) they have. The nested loop from Example 16 has two loops so it would be considered 2D. A three-deep loop would be called 3D, a four-deep loop would be called 4D, etc. I'm not entirely sure if there's an actual term for how deeply nested a loop is, but people should understand you regardless.

Loop Variables are Variables!

Don't forget that loop variables are just any variable... *including existing variables*. 99.99% of the time, you want to be using a new variable for the loop variable. Naming your loop variables so that they make sense is usually very helpful in this regard. However, you will commonly see these variable names used when making new loop variables:

- i, j, k (rarely do people go past 3D)
- a, b, c, d, ... (personally I see people start with `a` for 4D+)
- _ (underscore is used as a placeholder when the loop variable is not being used)

Reusing a pre-existing variable can be problematic, as seen in the following example.

In [ ]:
```python
# Example 6-17: Reusing a variable

counter = 100
for counter in range(1, 10):
    pass

print(counter)
```

```
9
```

Remember, `pass` does nothing, which means the only thing the loop does is assign the new number to the loop variable... which happens to be `counter`. The last integer to be stored in `counter` is 9 (because it's the largest integer between 1 and 10 but excluding 10). If `counter` was meant to be holding a different value, this override can lead to unexpected results although the code ran without errors.

However, this can be leveraged in nested loops. `range()` accepts any integer values, including those stored in variables. This is actually the basis of the very famous triangle problem:

In [ ]:
```
# Example 6-18: The triangle problem
"""
Create the following triangle:
1
22
333
4444
55555

Basically, each line should have the next number appear as many times as itself or in other words, the nth line
"""

for i in range(1, 6):
    for j in range(i):
        print(str(i), end="")
    print()
```

```
1
22
333
4444
55555
```

As seen here, the second loop loops from 0 to `i`! `i` goes from 1-5 (not 6 because the last number is excluded). This means that each iteration of the *inner loop* will change based on the outer loop. This is broken down as such:

- `i` is 1, `j` goes from 0-0 (just 0, so one loop)
- `i` is 2, `j` goes from 0-1 (0, 1; so two loops)
- `i` is 3, `j` goes from 0-2 (0, 1, 2; so three loops)
- `i` is 4, `j` goes from 0-3 (0, 1, 2, 3; so four loops)
- `i` is 5, `j` goes from 0-4 (0, 1, 2, 3, 4; so five loops)

The `end=""` prevents the `print()` from making the next `print()` start on the next line, allowing me to have all the numbers on the same line. If you remember string multiplication, then there's actually a solution involving just one loop:

In [ ]:
```
# Example 6-19: One loop solution to triangle problem

for i in range(1, 6):
    print(str(i) * i)
```

```
1
22
333
4444
55555
```