While Loops, Break and Continue

Introduction

Earlier in the course, we learned how to iterate over collections. But is there a way to have a loop **without** a collection to iterate over? Well, another way to create a loop is with **while** loops. We can use a while loop to perform the same action over and over until a condition is no longer `True`. We don't even need an *iterable* or collection to iterate over. We can just define a condition and perform the given code block until the condition is no longer `True`. Pretty cool, right?

Or what if we would like to have a loop that stops at a certain point? Let's say we only want to collect half of the elements of a list, or stop a list once we find the first matching element? To perform operations like these we'll need `break` and `continue` statements. These statements **control the flow** of our loops and will help us make our loops even more effective.

Objectives

You will be able to:

- Use a `while` loop
- Use `break` and `continue` to add control flow to a `while` loop

What is a `while` loop and how does it work?

A `while` loop is just that; a loop! Similar to a `for` loop, except there is no need for a collection to iterate over. Instead, a `while` loop uses a condition to know when to stop executing. When the condition is true, the block inside the while loop is executed. When that condition is false, we exit the while loop and move on to the next piece of our code.

Let's look at an example:

In [1]:
```
stop_number = 4
while stop_number > 0:
    print(stop_number)
    stop_number -=1
print("The stop_number reached", stop_number, "so the while loop's condition beca
```

```
4
3
2
1
The stop_number reached 0 so the while loop's condition became False and stoppe
d execution
```

In [3]:
```
manager = 4
while manager < 10:
    manager +=1
    print(manager)
print("The managers hired were",manager)
```

```
5
6
7
8
9
10
The managers hired were 10
```

In [8]:
```
students = 10
while students < 20:
    students += 1
    print(students)
print("the stop number has been reached at;" , students)
```

```
11
12
13
14
15
16
17
18
19
20
the stop number has been reached at; 20
```

Note the lack of a `list` or other collection, and that our second print statement only printed after our `stop_number` become 0.

Also, notice that the structure of a `while` loop is such that it could execute for an *unknown* amount of times. For example, if we didn't know the `stop_number` because it changed from time to time, our while loop could execute 100 times or 3.

For example, if we used a random number using the random library from `numpy` :

In [11]:
```python
import numpy as np
random_num = np.random.randint(1,20)
while random_num > 0:
    random_num -= 1
    print(random_num)
```

```
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
```

However, we know that eventually that number will be less than 0 and the loop will eventually stop. This is of critical importance. A while loop must always have a condition that will stop the loop, otherwise we will have an **infinite** loop. Infinite loops can crash your browser or program if you don't have a way to end it. So, it is very important to make sure your loops have a fairly defined **end** case.

If you do ever accidentally create an infinite loop, don't worry. Your current Notebook might freeze, and then kill the page to stop the execution. You can then re-open the browser again normally.

When To Use While Loops

While loops are fairly straight forward. We use them in instances where we have a **condition** that serves as the point at which we want a process to stop. For example, if we think about our appetite, we should eat until we aren't hungry, right? Some days that might be two slices of pizza, some days that might be 5 slices of pizza (and that is assuming all pizza slices are of equal size, which is a *generous* assumption).

In keeping with our food theme, let's see how we can make sure we're drinking enough water during the day using a while loop:

In [4]:
```
hydration = 0
water = 1 # in gallons
while hydration < 100 and water > 0:
    print('----[sips water]-----')
    water -= .1
    print('water is now at' , water)
    print('ah, that was refreshing')
    hydration += 10
    print('hydration is now at', hydration, '%\n')
```

```
----[sips water]-----
water is now at 0.9
ah, that was refreshing
hydration is now at 10 %

----[sips water]-----
water is now at 0.8
ah, that was refreshing
hydration is now at 20 %

----[sips water]-----
water is now at 0.7000000000000001
ah, that was refreshing
hydration is now at 30 %

----[sips water]-----
water is now at 0.6000000000000001
ah, that was refreshing
hydration is now at 40 %

----[sips water]-----
water is now at 0.5000000000000001
ah, that was refreshing
hydration is now at 50 %

----[sips water]-----
water is now at 0.40000000000000013
ah, that was refreshing
hydration is now at 60 %

----[sips water]-----
water is now at 0.30000000000000016
ah, that was refreshing
hydration is now at 70 %

----[sips water]-----
water is now at 0.20000000000000015
ah, that was refreshing
hydration is now at 80 %

----[sips water]-----
water is now at 0.10000000000000014
ah, that was refreshing
hydration is now at 90 %
```

```
----[sips water]-----
water is now at 1.3877787807814457e-16
ah, that was refreshing
hydration is now at 100 %
```

On To `break` And `continue` Statements

In the case of `break` and `continue` statements, it is almost best to not overthink. `break` and `continue` essentially do what they sound like. They are used in tandem with conditional statements (`if` , `elif`), inside loops, and they *break* out of a loop if a condition is met or *continue* a loop if a different condition is met. Before we dive too deeply into how these statements function, let's look at an example.

In [5]:
```python
numbers = list(range(0, 30))
new_list = []
for num in numbers:
    if len(new_list) > 4:
        print(f'We have enough even numbers in new_list ({len(new_list)}). break
        break
    elif num % 2 == 0:
        new_list.append(num)
    elif num % 2 != 0:
        continue
        print('i never get executed')
    print(num, 'is even.')
    print('this does not print for odd numbers\nbecause the continue statement sk
```

```
0 is even.
this does not print for odd numbers
because the continue statement skips
the code that follows in the for loop
and goes straight back to the next element in the for loop
2 is even.
this does not print for odd numbers
because the continue statement skips
the code that follows in the for loop
and goes straight back to the next element in the for loop
4 is even.
this does not print for odd numbers
because the continue statement skips
the code that follows in the for loop
and goes straight back to the next element in the for loop
6 is even.
this does not print for odd numbers
because the continue statement skips
the code that follows in the for loop
and goes straight back to the next element in the for loop
8 is even.
this does not print for odd numbers
because the continue statement skips
the code that follows in the for loop
and goes straight back to the next element in the for loop
We have enough even numbers in new_list (5). break will stop the for loop now
```

So, let's unpack what's happening here.

First, we have a `for` loop that is iterating over our list, `numbers`, which has 30 numbers in it. Then we see that we have an `if` statement that only executes its code when our `new_list` has more than 4 numbers in it. Once this condition is met, we **break** out of our loop. We then have an `elif` statement that adds any even number to the `new_list`, and a final `elif` statement that tests to see if the number is odd, and if it is odd, then it **continue**s to the next element in our for loop's iteration process.

So, first let's dig into the `continue` statement. `continue` is telling our loop to **skip** any code that comes after it and go straight to the next element in our loop. So, if we are dealing with number `1` in our list of numbers, we will hit our `continue` statement and immediately go to the next element in our for loop (i.e. `2`). Any code that comes after the `continue` will **not** be executed, **but** our for loop does **not** end execution. This contrasts with the `break` statement. Our break statement is only executed when we have met our condition that the `new_list` has reached the number of elements we want it to have. Once the `break` statement is executed, it will end the execution of the `for` loop altogether. All code after the `break` statement will be ignored, similar to our `continue`, but there will be no next step to the iteration. It stops the loop and that's the end.

So, essentially, we can use `continue` to *skip* operation on elements and code below `continue`. We can then use `break` when we have a condition that tells us we want to stop the process altogether.

We can look at two examples to really reinforce this difference in execution.

In [12]:
```python
for i in list(range(0, 5)):
    if True:
        print(i)
    print("Since we don't have a continue statement,\nI'll always get executed")
```

```
0
Since we don't have a continue statement,
I'll always get executed
1
Since we don't have a continue statement,
I'll always get executed
2
Since we don't have a continue statement,
I'll always get executed
3
Since we don't have a continue statement,
I'll always get executed
4
Since we don't have a continue statement,
I'll always get executed
```

In [8]:
```python
for i in list(range(0, 5)):
    if True:
        print(i)
        continue
    print("I'll never get executed")
```

```
0
1
2
3
4
```

In [6]:
```
for i in list(range(0, 5)):
    if True:
        print(i)
        break
    print("I'll never get executed either")
```

```
0
```

These examples are a bit contrived but they very clearly show how `continue` and `break` work inside loops and conditional statements as well as the difference between their execution.

Identify Opportunities to Use Break and Continue

Let's say you have a collection of elements and that you want to filter out certain elements and create a new list with the elements you want. But you also want to perform a complex operation on each of the elements you **do** want. Well, you don't want to perform that operation on the elements that you don't want and you don't want to have to write multiple `for` loops on the collection. So, a `continue` statement would allow you to optimize your performance and avoid needing to perform those operations. And a `break` statement can be used when you want to put a limit on the number of iterations you perform, the number of elements you append to a new list, or even just stop your iteration when you have found the first element you want. Let's take a look:

In [9]:
```
names = ['aNNE', 'JaNe', 'willIAM', 'WanDA', 'WeSt', 'HELEN', 'tHoMaS', 'HENrY',
formatted_names = []
check_count = 0

for name in names:
    if name.startswith('w') or name.startswith('W'):
        check_count += 1
        continue
    elif len(formatted_names) >= 4:
        check_count += 1
        break
    else:
        formatted_names.append(name.title())

print('before', names)
print('after', formatted_names)
print(check_count)
```

```
before ['aNNE', 'JaNe', 'willIAM', 'WanDA', 'WeSt', 'HELEN', 'tHoMaS', 'HENrY',
'John', 'Marshall', 'May']
after ['Anne', 'Jane', 'Helen', 'Thomas']
4
```

Okay, so, as we can tell from our code. We wanted to create a list of 4 names that are properly formatted and that **don't** start with the letter `w`. To optimize our code, we first check to see if the name starts with `w`. If it does, we skip all other operations that we need to do and go to the next name. Next we check to see if we have hit our quota of 4 names. If we have, then stop the iteration altogether. Lastly, if neither condition is met, format the given name and append it to our new list of formatted names. This way, we are optimizing our code by making sure that we eliminate performing any operations that aren't absolutely necessary at each step. You may have noticed the variable, `check_count` . If you are feeling unconvinced that the `break` and `continue` are not cutting down on the number of times our code is executing, run the cell below and compare the `check_count` s, which increment after **each** check of a conditional statement.

In [16]:
```
names = ['aNNE', 'JaNe', 'willIAM', 'WanDA', 'WeSt', 'HELEN', 'tHoMaS', 'HENrY',
formatted_names = []
check_count = 0

for name in names:
    if len(formatted_names) < 4:
        check_count += 1
        if not (name.startswith('w') or name.startswith('W')):
            check_count += 1
            formatted_names.append(name.title())


print('before', names)
print('after', formatted_names)
print(check_count)
```

```
before ['aNNE', 'JaNe', 'willIAM', 'WanDA', 'WeSt', 'HELEN', 'tHoMaS', 'HENrY',
'John', 'Marshall', 'May']
after ['Anne', 'Jane', 'Helen', 'Thomas']
11
```

See that?! We have cut down on the different checks we perform by more than half (50%)! And on top of that, to get the same result we have to have a **nested** `if` statement, which is, technically put, ***gross***. All joking aside, nested `if` statements should be avoided if they can be since they make our code less readable and therefore harder to maintain on top of being half as efficient as our previous code.

It's important to note that these excess checks could represent much more expensive operations in our code. So, it is important to use `break` and `continue` when it makes sense.

Summary

Awesome! `while` loops are great, right? We can use them to perform operations based on the truthiness of a condition instead of needing to iterate over the elements in a collection. They provide a more dynamic way to perform operations, but we need to be careful when writing while loops because we need to have a condition that ends the loop. Otherwise we will have an **infinite** loop which will give us and our computer a real headache. In this lesson, we also introduced some *control flow* statements, `break` and `continue` , which allow us to make our conditional statements and the rest of our code more efficient and more readable. We call these statements control flow statements because they allow us to *control* the *flow* of our code's execution.