Looping Over Collections

Introduction

Loops allow us to iterate over each element in a collection, like a list. Perhaps we could already do that by writing out a line of code for each element in the collection, but that wouldn't be very efficient, would it? No, not at all. With loops, we can write one line of code that operates on each element in a collection. Pretty cool, right? Let's get started!

Objectives

You will be able to:

- Use a `for` loop to iterate over a collection

What is a for loop and how do I write one?

A `for` loop in Python, is primarily used for going through elements of a list one by one. We'll use a simple collection with 4 elements `0,1,2,3` as an example. Without a loop, if we wanted to print each element of the list we'd have to write it out like we do below:

In [3]:
```
zero_to_three = [0, 1, 2, 3]
```

In [7]:
```
print(zero_to_three[0])
print(zero_to_three[1])
print(zero_to_three[2])
print(zero_to_three[3])
```

```
0
1
2
3
```

In [2]:
```
subjects = ['Hist' , 'Geo' , 'Math' , 'Kisw' , 'Engl']
for index, subject in enumerate(subjects):
    print(index, subject)
```

```
0 Hist
1 Geo
2 Math
3 Kisw
4 Engl
```

Press shift + enter

In the example above, we are sequentially accessing each index in the list and printing its value (the element). That works well enough, but if our list were 100 elements long it would become extremely tedious. And what if the length of our list were ***unknown***. Spooky, right?

In fact, it may very often be the case that we don't know the length of the collection we are working with. So, writing all this static code for each element becomes not only unmanageable but impossible.

Let's see how we would do the same operation above with a `for` loop!

In [4]:
```
for number in zero_to_three:
    print(number)
```

```
0
1
2
3
```

In [2]:
```
#practice
#my_list = [1 , 2 , 3 , 4 , 5]
#for num in my_list:
    #print(num)
```

```
1
2
3
4
5
```

Great! We were able to reproduce the exact same functionality as we had previously. However, in this example, we used only **2** lines of code.

Now, the `for` loop may look a bit confusing at first, so, let's take a closer look at the syntax. A `for` loop essentially has two necessary components, which we'll refer to as arguments. The first argument is the variable name we are assigning to an element, or in this case, `number`. The second argument is the collection we are iterating over, or in this case, the list `zero_to_three`.

We can give any name to the variable. The important thing to understand here is **it is the reference** to each **element** in the collection. So, when we print `number`, we are printing an element of the collection `zero_to_three`.

Every `for` loop needs to end the first line with a colon `:`. This indicates the start of the **block** of code. The block is simply the code that we want executed in each iteration of our loop. So, if all we want to do is print each element, then the line that prints the element is our block. Our block is indicated by indenting. So the first line after the colon `:` should be indented. When we want to end our block, we simply stop indenting. Any code following the `for` loop will only be executed after the `for` loop finishes.

> *Remember, the block of code in a `for` loop is executed the same amount of times as there are elements in the collection.*

Let's take a look at changing the variable names and adding more code to our example:

In [22]:
```
iteration_count = 0
for whatever_we_want in zero_to_three:
    iteration_count += 1
    print(whatever_we_want)
    print("This is iteration:", iteration_count)
print("The for loop is finished now! I am not in the for loop block, which is why
```

```
0
This is iteration: 1
1
This is iteration: 2
2
This is iteration: 3
3
This is iteration: 4
The for loop is finished now! I am not in the for loop block, which is why I on
ly printed once!
```

In [12]:
```
# Practice
iteration_count = 0
my_numbers = [0, 1, 2, 3, 4]
for num in my_numbers:
    print(num)
    print('my number is currently at:', iteration_count)
    iteration_count +=1
print('There are no more numbers and so i stop')
```

```
0
my number is currently at: 0
1
my number is currently at: 1
2
my number is currently at: 2
3
my number is currently at: 3
4
my number is currently at: 4
There are no more numbers and so i stop
```

Using list elements as indices

In the examples above, we used the elements in our list to perform an operation, which was printing them. We can also use a list of numbers to access elements from another list. Let's take a look at an example.

In [8]:
```
countries = ['Croatia', 'USA', 'Argentina', 'France', 'Brazil', 'Japan', 'Vietnam
```

In [31]:
```
for index in [0,1,2,3,4,5,6,7]:
    print(index)
    print(countries[index])
```

```
0
Croatia
1
USA
2
Argentina
3
France
4
Brazil
5
Japan
6
Vietnam
7
Israel
```

So, in the example above, we are still using the elements in the list of numbers from 0 to 7 in our `for` loop, but we are instead using them to access each element of another list. This example is a bit contrived, but perhaps you have two lists that are ordered correctly and have information like the capital cities in one list and the corresponding countries in another. How would we print both of those out in the same line?

In [12]:
```
countries = ['Croatia', 'USA', 'Argentina', 'France', 'Brazil', 'Japan', 'Vietnam
cities = ['Zagreb', 'District of Columbia', 'Buenos Aires', 'Paris', 'Rio de Jane
print(countries)
print(cities)
```

```
['Croatia', 'USA', 'Argentina', 'France', 'Brazil', 'Japan', 'Vietnam', 'Israe
l']
['Zagreb', 'District of Columbia', 'Buenos Aires', 'Paris', 'Rio de Janeiro',
'Tokyo', 'Hanoi', 'Tel Aviv']
```

In [13]:
```
for index in [0,1,2,3,4,5,6,7]:
    print(cities[index]+",", countries[index])
```

```
Zagreb, Croatia
District of Columbia, USA
Buenos Aires, Argentina
Paris, France
Rio de Janeiro, Brazil
Tokyo, Japan
Hanoi, Vietnam
Tel Aviv, Israel
```

In [19]:
```
numbers = [0,1,2,3,4,5,6,7]
for num in numbers:
    print(cities[num]+',', countries[num])
```

```
Zagreb, Croatia
District of Columbia, USA
Buenos Aires, Argentina
Paris, France
Rio de Janeiro, Brazil
Tokyo, Japan
Hanoi, Vietnam
Tel Aviv, Israel
```

Of course, this does not work if our indices do not match up with the size of our list.

In [14]:
```
for index in [0,1,2,3,4,5,6,7,8,9,10]:
    print(cities[index]+",", countries[index])
```

```
Zagreb, Croatia
District of Columbia, USA
Buenos Aires, Argentina
Paris, France
Rio de Janeiro, Brazil
Tokyo, Japan
Hanoi, Vietnam
Tel Aviv, Israel

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_64/630477222.py in <module>
      1 for index in [0,1,2,3,4,5,6,7,8,9,10]:
----> 2     print(cities[index]+",", countries[index])

IndexError: list index out of range
```

In [6]:
```
first_names = ['Bonnie' , 'Ken' , 'Timo' , 'Stan']
second_names = ['Mutua' , 'Koome' , 'Muti' , 'Mwenda']
for number in [0, 1, 2 , 3]:
    print(first_names[number] +',', second_names[number])
```

```
Bonnie, Mutua
Ken, Koome
Timo, Muti
Stan, Mwenda
```

In [ ]:

In [ ]:

In [ ]:

So, the preferred way of figuring out the number of iterations on a list when you are unsure of its length would be to use the `len` function to calculate the size of the list.

In [21]:
```
len(countries)
```

Out[21]:
```
8
```

Then we can turn this length into a successive list of elements in the following way:

First, create a range object:

In [21]:
```
range(0, len(countries))
```

Out[21]:
```
range(0, 8)
```

And then convert this into a list:

In [22]:
```
list(range(0, len(countries)))
```

Out[22]:
```
[0, 1, 2, 3, 4, 5, 6, 7]
```

Note that the range object is marking the starting and ending point, and excluding the end. So this works perfectly:

In [24]:
```
for index in list(range(0, len(countries))):
    print(cities[index]+",", countries[index])
```

```
Zagreb, Croatia
District of Columbia, USA
Buenos Aires, Argentina
Paris, France
Rio de Janeiro, Brazil
Tokyo, Japan
Hanoi, Vietnam
Tel Aviv, Israel
```

And as we add or subtract countries, we will still be iterating through our list elements.

In [15]:
```
countries.append('Mexico')
cities.append('Mexico City')
for index in list(range(0, len(countries))):
    print(cities[index]+",", countries[index])
```

```
Zagreb, Croatia
District of Columbia, USA
Buenos Aires, Argentina
Paris, France
Rio de Janeiro, Brazil
Tokyo, Japan
Hanoi, Vietnam
Tel Aviv, Israel
Mexico City, Mexico
```

In [8]:
```
iteration_count = 0
countries = ['Kenya' , 'Tz' , 'Ug', 'Congo' , 'Zambia']
for country in countries:
    iteration_count += 1
    print(iteration_count)
    print('This is my country:', country)
print("I stop here after Zambia, since these countries are too many")
```

```
1
This is my country: Kenya
2
This is my country: Tz
3
This is my country: Ug
4
This is my country: Congo
5
This is my country: Zambia
I stop here after Zambia, since these countries are too many
```

> Note: More conventionally, these contrived examples would employ the `enumerate()` method, but that is beyond the scope of the current lesson. At some point in the future, examine how this code snippet works:

```
for idx, item in enumerate(['A', 'B', 'C']):
    print(idx, item)
```

Iterating through different datatypes

So far our loop variable has always been an element of a list that is a number. However, our loop variable can represent any data type. For example, let's have the loop variable represent each of the countries directly:

In [23]:
```
different_elements = ['A String', ["a", 'list', "of", 5, ["elements"]], {'this':
for element in different_elements:
    print(element)
```

```
A String
['a', 'list', 'of', 5, ['elements']]
{'this': 'is a dictionary'}
```

Now that we know we can iterate through a list that contains multiple data types, let's explore iterating through a data type that's **not a list**.

Another collection we commonly will iterate over is a **dictionary**. Dictionaries differ from lists, on a high level, in that elements are **key, value pairs** instead of one single element. So, when we go through each item in a dictionary, we are actually working with a two-part element (with a key & value). Similarly to how we name a variable for the element in a list for a `for` loop, we name a variable for both the **key** and **value** when we iterate over a dictionary. However, in Python, we can't iterate directly over the dictionary, we iterate over the **items** of a dictionary, which are the key-value pairs. Let's take a look at an example.

In [10]:
```
example_dictionary = {'first_name': "Terrance", 'last_name': "KOAR", 'favorite_la
example_dictionary.items()
print(example_dictionary.items())
type(example_dictionary.items())
```

```
dict_items([('first_name', 'Terrance'), ('last_name', 'KOAR'), ('favorite_langu
age', 'Python')])
```

Out[10]:
```
dict_items
```

Here we can see this **dict_items** object looks almost like a list, but each item has **two** parts, the **key** and **value**. So, in our first iteration, the first **key** will be **first_name**, and the first **value** will be **Terrance**.

In [12]:
```
for key, value in example_dictionary.items():
    print("this is the key:", key)
    print("this is the value:", value , "\n")
```

```
this is the key: first_name
this is the value: Terrance

this is the key: last_name
this is the value: KOAR

this is the key: favorite_language
this is the value: Python
```

In [31]:
```
# EXAMPLE
# myself = {'first_name' : 'mukaria' , 'second_name' : 'Mutua' , 'email' : 'muktu
# for key, value in myself.items():
#     print('this is the key :' , key)
#     print('this is the value:' , value , '\n')
```

```
this is the key : first_name
this is the value: mukaria

this is the key : second_name
this is the value: Mutua

this is the key : email
this is the value: muktua@com
```

In [37]:
```
# first_name = ''
# second_name = ''
# for key, value in myself.items():
#     if key == 'second_name':
#         last_name = value.title()
#     if key == 'first_name':
#         second_name = value
# print(first_name, second_name)
```

```
 Mukaria
```

So, we can see that the **dict_items** object groups the key values together in a way that we can iterate over them and access them. We can even use them to inform our program to operate on keys and values in a certain way. Such as, the last name is inexplicably in all caps. Let's look at how we can rectify that and title case the last name when we print out the full name of the `example_dictionary` object.

In [39]:
```
first_name = ""
last_name = ""
for key, value in example_dictionary.items():
    if key == "last_name":
        last_name = value.title()
    if key == "first_name":
        first_name = value
print(first_name, last_name)
```

```
Terrance Koar
```

Conventional Naming Patterns

Typically, when we are looping through a collection of things like `countries`, we will name the looping variable, `country`, since that is the singular version of the plural name that represents our list. This is convention and helps to not only remind us, but tell other people looking at our code what that variable is. Let's take a look at a couple examples.

In [40]:
```
for country in countries:
    print(country)
```

```
Croatia
USA
Argentina
France
Brazil
Japan
Vietnam
Israel
```

In [22]:
```
ice_cream_flavors = ['Mint Chocolate Chip', 'Coffee', 'Cookie Dough', 'Fudge Mint
for ice_cream_flavor in ice_cream_flavors:
    print('I love ' + ice_cream_flavor + ' ice cream!!')
```

```
I love Mint Chocolate Chip ice cream!!
I love Coffee ice cream!!
I love Cookie Dough ice cream!!
I love Fudge Mint Brownie ice cream!!
I love Vanilla Bean ice cream!!
```

Summary

In this lesson, we learned how to use loops to iterate through a collection of elements. We started with iterating through a list of numbers, and performed the same operation on each number. Then we saw how we can loop through the numbers and have each number be used to access a successive element from a separate list, like `countries`. We then saw that to ensure our list of numbers matched the indices of our other list, we had to use the expression, `for element in list(range(0, len(list)))`. Finally, we introduced a naming convention that is commonly used when naming the variable for our loops when iterating over a collection that is a list of common elements (i.e. `ice_cream_flavor` for a list of `ice_cream_flavors`).