**Option 6: What is an iterator? How does it relate to a loop?**

According to Programiz, an iterator in the context of Python programming “…is simply an object that can be iterated upon. An object which will return data, one element at a time. Technically speaking, a Python iterator object must implement two special methods, \_\_iter\_\_() and \_\_next\_\_() , collectively called the iterator protocol.” A list can be used to build an iterator since elements can be returned one at a time.

For example:

# define a list

my\_list = [1, 3, 5, 6]

# define an iterator using iter()

my\_iterator = iter(my\_list)

# iterate through the iterator object using next()

print(next(my\_iterator))

print(next(my\_iterator))

print(next(my\_iterator))

print(next(my\_iterator))

The output would like this:

1

3

5

6

Iterators are commonly used in loops as follows:

*# define a list*my\_list = [1, 3, 5, 6]  
  
*# create an iterator object from that list*iter\_obj = iter(my\_list)  
  
*# infinite loop*while True:  
 try:  
 *# get the next item* element = next(iter\_obj)  
 print(element)  
 except StopIteration:  
 *# Break from the loop when there are no more elements* break

There can be infinite iterators (which never ends). We must be careful when handling such iterators.

An example of an infinite iterator can be built using the iter() function and the next() function as follows:

inf = iter(int,1)

next(inf)

Because the int() function always returns 0 this will be infinitely 0.

Additionally, we can check if an object is iterable by creating a function isiterable() on a given object. The function will return True if the object is iterable or False if it is not. Most Python objects, like strings, and lists are iterable. This method is known as “duck typing” (McKinney, 2018)

def isiterable(obj):  
 try:  
 iter(obj)  
 return True  
 except TypeError:  
 return False  
  
print(isiterable(**'the house is white'**))  
  
print(isiterable(3.4))  
  
print(isiterable([3,4,5,**'a'**,**'b'**]))

The results are:

True

False

True

Iterators can not only be used within a loop, but they can also be built within a loop. An example would be as follows:

class InfIter:  
 *"""Infinite iterator to return all  
 odd numbers"""* def \_\_iter\_\_(self):  
 self.num = 1  
 return self  
  
 def \_\_next\_\_(self):  
 num = self.num  
 self.num += 2  
 return num  
   
   
a = iter(InfIter())  
  
n = 1  
while n < 10:  
 print(next(a))  
 n += 1

The result would be as follows:

1

3

5

7

9

11

13

15

17

As you can see, iterators are widely used in Python programming and can be very helpful in for or while loops. They can be explicitly defined or built “on the fly” which can improve the use of memory by the program since the objects are not necessarily stored before being used. This is very important, especially when there is a need to use “infinite” or “almost infinite” loops.

**References:**

Programiz. https://www.programiz.com/python-programming/iterator

Severance C. 2016. Python fpr Everybody: Exploring Data in Python 3. Creative Commons.

McKinney W. 2018. *Python for Data Analysis: Data Wrangling with Pandas, Numpy, and Ipython.* O’Reilly. Second Edition.

Sweigart A. (2020), Automate The Boring Stuff With Python. No Starch Press, Inc. 2nd Edition.

Python Software Foundation. September 22, 2020, The Python Standard Library, <https://docs.python.org/3/tutorial/modules.html>