Introduction to Iterators

You have probably been using iterators and generators since you started programming in Python but you may not have realized it. In this chapter, we will learn what an iterator and a generator are. We will also be learning how they are created so we can create our own should we need to.

Iterators

An iterator is an object that will allow you to iterate over a container. The iterator in Python is implemented via two distinct methods: **__iter__** and **__next__**. The **__iter__** method is required for your container to provide iteration support. It will return the iterator object itself. But if you want to create an iterator object, then you will need to define **__next__** as well, which will return the next item in the container.

Note: In Python 2, the naming convention was slightly different. You still needed__iter__, but__next__ was called **next**.*

To make things extra clear, let's go over a couple of definitions:

> - iterable - an object that has the __iter__ method defined
> - iterator - an object that has both __iter__ and __next__ defined where __iter__ will return the iterator object and __next__ will return the next element in the iteration.
>
> As with most magic methods (the methods with double-underscores), you should not call __iter__ or __next__ directly. Instead you can use a **for** loop or list comprehension and Python will call the methods for you automatically. There are cases when you may need to call them, but you can do so with Python's built-ins: **iter** and **next**.
>
> Before we move on, I want to mention Sequences. Python 3 has several sequence types such as list, tuple and range. The list is an iterable, but

not an iterator because it does not implement __next__. This can be easily seen in the following example:

```
my_list = [1, 2, 3]
next(my_list)
#Traceback (most recent call last):
#  Python Shell, prompt 2, line 1
#builtins.TypeError: 'list' object is not an iterator
```

```
my_list = [1, 2, 3]
next(my_list)
#Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 2, in <module>
# next(my_list)
#TypeError: 'list' object is not an iterator
```

When we tried to call the list's next method in the example above, we received a **TypeError** and were informed that the list object is not an iterator. But we can make it one! Let's see how:

```
iter(my_list)
#<list_iterator object at 0x7faaaa477a58>

list_iterator = iter(my_list)
next(list_iterator)
#1

next(list_iterator)
#2

next(list_iterator)
# 3

next(list_iterator)
#Traceback (most recent call last):
#   Python Shell, prompt 8, line 1
# builtins.StopIteration:
```

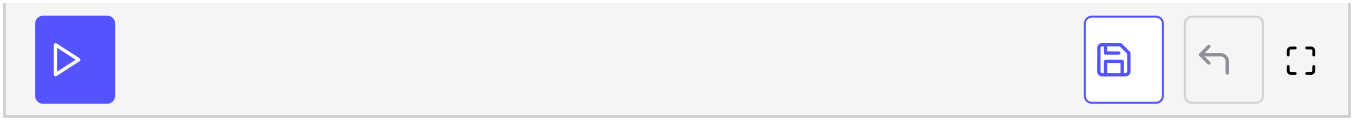
```
print (iter(my_list))
#<list_iterator object at 0x7f6484043f28>

list_iterator = iter(my_list)
print (next(list_iterator))
#1

print (next(list_iterator))
#2

print (next(list_iterator))
#3

print (next(list_iterator))
#Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 16, in <module>
# print (next(list_iterator))
#StopIteration:
```



To turn the list into an iterator, just wrap it in a call to Python's **iter** method. Then you can call **next** on it until the iterator runs out of items and **StopIteration** gets raised. Let's try turning the list into an iterator and iterating over it with a loop:

> ```
> for item in iter(my_list):
>     print(item)
>
> #1
> #2
> #3
> ```
>
> When you use a loop to iterate over the iterator, you don't need to call next and you also don't have to worry about the StopIteration exception being raised.

```
for item in iter(my_list):
    print(item)

#1
#2
#3
```