Python Iterator and Generator

What is an Iteration

Iteration is a general term for taking each item of something, one after another. Any time you use a loop, explicit or implicit, to go over a group of items, that is iteration.

In [10]:
```
L=[1,2,3]
for i in L:
    print(i)
# iteration is happening
```

```
1
2
3
```

What is an Iterator

An Iterator is an object that allows the programmer to traverse through a sequence of data without having to store the entire data in the memory

In [17]:
```
# Example
L = [x for x in range(1,10000)]
import sys
print(sys.getsizeof(L)/64)
x = range(1,10000000000)
print(sys.getsizeof(x)/64)

# here x is an iterator which is iterating on range which is an iterable
```

```
1330.875
0.75
```

What is an Iterable

Iterable is an object, which one can iterate over It generates an Iterator when passed to iter() method.

In [26]:
```
# Example
L = [1,2,3]
print(type(L))
# L is an iterable
type(iter(L))
# iter(L) --> iterator
```

```
<class 'list'>
```

Out[26]: list_iterator

Point to remember

- Every **Iterator** is also and **Iterable**
- Not all **Iterables** are **Iterators**

Trick

- Every Iterable has an **iter function**
- Every Iterator has both **iter function** as well as a **next function**

In [36]:
```
L=[1,2,3]
# so here L is iterable but not iterator because it store whole data in one time
```

In [38]:
```
T=(1,2,3)
dir(T)
```

Out[38]:
```
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'count',
 'index']
```

In [40]:
```
# if a object is iterable then we can loop on it
```

In [ ]:
```
L=[12,34,5,6]
# L is an iterable but it is not iterator
```

```
iter_L=iter(L)
# iter_L is an iterator
print(type(iter_L))
print(dir(iter_L))
```

How to check if object is iterator ?

- if dir(obj) give **iter** method as well as **next** method then it is iterator

Understanding how for loop works

In [51]:
```
num=[1,2,3]
for i in num:
    print(i)
```

```
1
2
3
```

In [53]:
```
# step 1 : it fetch the iterator ---> iter_num=iter(n)
# step 2 : next (iter_num) -->  next(iter_num) --> next(iter_num)
```

In Python, a `for` loop on a list (or any iterable) essentially does two main things behind the scenes:

1. **It gets an iterator** for the iterable using the `iter()` function.
2. **It repeatedly calls `next()` on the iterator** to get each element, one by one, until the iterator is exhausted (raises `StopIteration`).

Let's walk through the example code you provided in detail:

```
num = [1, 2, 3]
for i in num:
    print(i)
```

Step-by-Step Explanation

1. **Creating the Iterator**:

- When the `for` loop starts, Python calls `iter(num)` to get an iterator object for the list `num`.
- Let's call this iterator `iter_num`.

```
iter_num = iter(num)
```

At this point, `iter_num` is an iterator object that "remembers" where it is in the sequence of items in `num`.

2. **First Call to `next(iter_num)`**:
   - The `for` loop calls `next(iter_num)` to get the next item in the sequence.
   - The iterator returns `1`, which is the first item in `num`.
   - The loop variable `i` is assigned the value `1`, and `print(i)` outputs `1`.
3. **Second Call to `next(iter_num)`**:
   - The `for` loop automatically calls `next(iter_num)` again to get the next item.
   - This time, the iterator returns `2`, which is the second item in `num`.
   - `i` is assigned the value `2`, and `print(i)` outputs `2`.
4. **Third Call to `next(iter_num)`**:
   - The `for` loop calls `next(iter_num)` again to get the third item.
   - The iterator returns `3`, which is the third item in `num`.
   - `i` is assigned the value `3`, and `print(i)` outputs `3`.
5. **End of the Iterator**:
   - When `for` calls `next(iter_num)` again, there are no more items left in `num`.
   - The iterator raises a `StopIteration` exception, which signals the end of the sequence.
   - The `for` loop automatically handles this exception and stops iterating, ending the loop.

Key Concepts

- **Iterator**: An object that enables traversing through a sequence one element at a time.
- **`iter()`**: The function that returns an iterator from an iterable.
- **`next()`**: The function that retrieves the next item from an iterator, raising `StopIteration` when there are no more items.

Summary of Behind-the-Scenes Operations

The `for` loop:

1. Calls `iter()` on `num` to get an iterator.
2. Uses `next()` on the iterator to get each item, assigns it to `i`, and runs the loop body (`print(i)`).
3. Ends when `next()` raises `StopIteration`.

So, `for i in num:` is just a convenient and readable way to use `iter()` and `next()` under the hood without having to explicitly manage the iterator.

In [56]:
```
num = [1,2,3]

# fetch the iterator
iter_num = iter(num)

# step2 --> next
next(iter_num)
next(iter_num)
next(iter_num)
```

Out[56]: 3

Making our own for loop

In [60]:
```
def mera_khudka_for_loop(iterable):

    iterator = iter(iterable)

    while True:

        try:
            print(next(iterator))
        except StopIteration:
            break
```

In [64]:
```
a = [1,2,3]
b = range(1,11)
c = (1,2,3)
```

```
d = {1,2,3}
e = {0:1,1:1}

mera_khudka_for_loop(b)
```

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

A confusing point

In [76]:
```
num=[1,2,3]
iter_obj=iter(num)
print(id(iter_obj),"Address of iterator")
iter_obj2=iter(iter_obj)
print(id(iter_obj2),"Address of iterator 2")
```

```
1808077197792 Address of iterator
1808077197792 Address of iterator 2
```

- iter_obj2 = iter(iter_obj) attempts to create another iterator from iter_obj.
- However, in Python, calling iter() on an iterator itself simply returns the same iterator object (it doesn't create a new one).
- This means iter_obj2 is essentially the same object as iter_obj, not a new iterator.
- So, id(iter_obj2) will be the same as id(iter_obj), as iter_obj2 and iter_obj both reference the same iterator.

In [68]:
```
class mera_range:

    def __init__(self,start,end):
        self.start = start
        self.end = end
```

```
    def __iter__(self):
        return mera_range_iterator(self)
```

In [70]:
```
class mera_range_iterator:

    def __init__(self,iterable_obj):
        self.iterable = iterable_obj

    def __iter__(self):
        return self

    def __next__(self):

        if self.iterable.start >= self.iterable.end:
            raise StopIteration

        current = self.iterable.start
        self.iterable.start+=1
        return current
```

In [72]:
```
x = mera_range(1,11)
```

In [74]:
```
type(x)
```

Out[74]:
```
__main__.mera_range
```

In [80]:
```
iter(x)
```

Out[80]:
```
<__main__.mera_range_iterator at 0x1a4f9cc8bf0>
```

In [82]:
```
for i in x:
    print(i)
```

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

Step-by-Step Flow

Let's go through the flow assuming the following code:

```
my_range = mera_range(1, 5)
for num in my_range:
    print(num)
```

Execution Steps

1. **Creating the `mera_range` Object**:
   - `my_range = mera_range(1, 5)` calls `mera_range.__init__()`, setting `self.start = 1` and `self.end = 5`.
2. **Starting the `for` Loop**:
   - When `for num in my_range` is executed, `iter(my_range)` is implicitly called.
   - `mera_range.__iter__()` is executed, which returns an instance of `mera_range_iterator` initialized with `my_range`.
3. **Initializing the Iterator (`mera_range_iterator`)**:
   - Inside `mera_range.__iter__()`, `mera_range_iterator.__init__()` is called with `my_range` as the `iterable_obj`.
   - `self.iterable` in `mera_range_iterator` now holds a reference to `my_range`, so it can access `my_range.start` and `my_range.end`.
4. **First Iteration**:
   - `next()` is implicitly called on the iterator.
   - Inside `mera_range_iterator.__next__()`:

- **Check**: `self.iterable.start` (1) is less than `self.iterable.end` (5), so it does not raise `StopIteration`.
- **Assign `current`**: `current` is set to `self.iterable.start`, which is 1.
- **Increment**: `self.iterable.start` is incremented to 2.
- **Return**: The method returns `current`, which is 1, and `print(num)` outputs `1`.

5. **Second Iteration**:

- `next()` is called again.
- **Check**: `self.iterable.start` (2) is less than `self.iterable.end` (5).
- **Assign `current`**: `current` is set to 2.
- **Increment**: `self.iterable.start` is incremented to 3.
- **Return**: The method returns `current`, which is 2, and `print(num)` outputs `2`.

6. **Third and Fourth Iterations**:

- This process repeats similarly, with `self.iterable.start` being set to 3 and 4 in subsequent iterations, outputting `3` and `4`.

7. **Fifth Iteration (End Condition)**:

- `next()` is called, and now `self.iterable.start` is 5.
- **Check**: `self.iterable.start` (5) is equal to `self.iterable.end` (5).
- The method raises `StopIteration`, ending the loop.

Output

The output of `for num in my_range: print(num)` will be:

```
1
2
3
4
```

Summary

1. **`mera_range`** creates a custom range-like object with a start and end.
2. **`mera_range_iterator`** defines the iterator logic, keeping track of the current value and checking the end condition.

3. The iterator yields each value in the range `[start, end)`, incrementing `start` each time, and raises `StopIteration` when the end is reached.

This custom implementation mimics the behavior of Python's built-in `range` object.

Generator

- Python generators are a simple way of creating iterators.
- It reduces the problem of storing whole data at once in memory

In [101…
```
def gen_demo():

    yield "first statement"
    yield "second statement"
    yield "third statement"
```

In [105…
```
gen=gen_demo() # return a generator and never call the method again
for i in gen:
    print(i)
```

```
first statement
second statement
third statement
```

Here's what happens:

1. **Creating the Generator Object** (`gen = gen_demo()`):
   - When you call `gen_demo()`, it doesn't run the entire function immediately. Instead, it initializes a generator object, `gen`, that holds the state of the function and pauses execution at the start.
2. **Iterating Through the Generator** (`for i in gen`):
   - The `for` loop does not call `gen_demo()` repeatedly. Instead, it calls the `next()` function on the generator object `gen` in each iteration.
   - Each call to `next(gen)` advances `gen_demo` to the next `yield` statement.

- The generator resumes from where it last yielded a value and pauses again at the next `yield`, returning the yielded value to the loop.

So, in short:

- `gen_demo()` is only called once to create the generator.
- The `for` loop repeatedly calls `next()` on the generator object `gen`, advancing it through the `yield` statements until it's exhausted.

When you create an iterator, it needs a way to access or manage the data it's iterating over. In many cases, this means storing all the data in memory, which can be less memory-efficient than a generator.

In [108…
```python
def square(num):
    for i in range(1,num+1):
        yield i**2
```

In [110…
```python
gen = square(10)

print(next(gen))
print(next(gen))
print(next(gen))

for i in gen:
    print(i)
```

```
1
4
9
16
25
36
49
64
81
100
```

Range Function using Generator

In [114…
```
def mera_range(start,end):
    for i in range(start,end):
        yield i
```

In [116…
```
for i in mera_range(15,20):
    print(i)
```

```
15
16
17
18
19
```

Generator Expression

In [119…
```
# list comprehension
L = [i**2 for i in range(1,101)]
```

In [ ]:
```
gen = (i**2 for i in range(1,101))

for i in gen:
    print(i)
```

- This line creates a generator expression, which is similar to a generator function but written in a more compact form.
- (i**2 for i in range(1, 101)) is an expression that generates the squares of numbers from 1 to 100.
- gen is a generator object created by the expression, and it will yield each squared value one by one when requested.

In [ ]:
```
!pip install opencv-python
```

In [153…
```
import os
import cv2

def image_data_reader(folder_path):

    for file in os.listdir(folder_path):
        f_array = cv2.imread(os.path.join(folder_path,file))
```

```
        yield f_array
```

In [159…

```
gen = image_data_reader('image/')
next(gen)
```

Out[159…

```
array([[[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       ...,

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
```

```
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        ...,
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]], dtype=uint8)
```

END