Python Iterators

June 8, 2024

0.1 Python Iterators

Dr. Labeed Al-Saad

**An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods **iter**() and **next**().

0.2 Iterator vs Iterable

**Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.

All these objects have a iter() method which is used to get an iterator:

Example:

Return an iterator from a tuple, and print each value:

[1]:
```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

```
apple
banana
cherry
```

Even strings are iterable objects, and can return an iterator:

Example:

Strings are also iterable objects, containing a sequence of characters:

[2]:
```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
```

```
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

```
b
a
n
a
n
a
```

0.3 Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

Example:

Iterate the values of a tuple:

[3]:
```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

```
apple
banana
cherry
```

Example:

Iterate the characters of a string:

[4]:
```
mystr = "banana"

for x in mystr:
  print(x)
```

```
b
a
n
a
n
a
```

**The for loop actually creates an iterator object and executes the next() method for each loop.

0.4 Create an Iterator

To create an object/class as an iterator you have to implement the methods **iter**() and **next**() to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called **init**(), which allows you to do some initializing when the object is being created.

The **iter**() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The **next**() method also allows you to do operations, and must return the next item in the sequence.

Example:

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

[5]:
```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

```
1
2
3
4
5
```

0.5 StopIteration

The example above would continue forever if you had enough next() statements, or if it was used in a for loop.

To prevent the iteration from going on forever, we can use the StopIteration statement.

In the **next**() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example:

Stop after 20 iterations:

[10]:
```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```