Python Iterator vs Iterable

Summary: in this tutorial, you'll learn about Python iterator and iterable and their differences.

Iterators

An iterator (https://www.pythontutorial.net/advanced-python/python-iterators/) is an object that implements the iterator protocol. In other words, an iterator is an object that implements the following methods:

- `__iter__` returns the iterator object itself.
- `__next__` returns the next element.

Once you complete iterating a collection using an iterator, the iterator becomes exhausted.

It means that you cannot use the iterator object again.

Iterables

An iterable (https://www.pythontutorial.net/python-basics/python-iterables/) is an object that you can iterate over.

An object is iterable when it implements the `__iter__` method. And its `__iter__` method returns a new iterator.

Examining the built-in list and list iterator

In Python, a list (https://www.pythontutorial.net/python-basics/python-list/) is an ordered collection of items. It's also an iterable because a list object has the `__iter__` method that returns an iterator. For example:

```
numbers = [1, 2, 3]

number_iterator = numbers.__iter__()
print(type(number_iterator))
```

Output:

```
<class 'list_iterator'>
```

In this example, the `__iter__` method returns an iterator with the type `list_iterator`.

Because the `list_iterator` implements the `__iter__` method, you can use the `iter` built-in function to get the iterator object:

```
numbers = [1, 2, 3]
number_iterator = iter(numbers)
```

Since the `list_iterator` also implements the `__next__` method, you can use the built-in function `next` to iterate over the list:

```
numbers = [1, 2, 3]

number_iterator = iter(numbers)

next(number_iterator)
next(number_iterator)
next(number_iterator)
```

If you call the `next` function once more, you'll get a `StopIteration` exception.

```
next(number_iterator)
```

Error:

```
StopIteration
```

This is because the list iterator has been exhausted. To iterate the list again, you need to create a new iterator.

This illustrates the separating the list from its iterator. The list is created once while the iterator is created every time you need to iterate over the list.

Python Iterator and Iterable

The following defines the `Colors` class:

```
class Colors:
    def __init__(self):
        self.rgb = ['red', 'green', 'blue']
        self.__index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.__index &gt;= len(self.rgb):
            raise StopIteration

        # return the next color
        color = self.rgb[self.__index]
        self.__index += 1
        return color
```

In this example, the `Colors` class plays two roles: iterable and iterator.

The `Colors` class is an iterator because it implements both `__iter__` and `__next__` method. The `__iter__` method returns the object itself. And the `__next__` method returns the next item from a list.

The `Colors` class is also an iterable because it implements the `__iter__` method that returns an object itself, which is an iterator.

The following creates a new instance of the `Colors` class and iterates over its elements using a `for` (https://www.pythontutorial.net/python-basics/python-for-loop-list/) loop:

```
colors = Colors()

for color in colors:
    print(color)
```

Once you complete iterating, the `colors` object becomes useless. If you attempt to iterate it again, you'll get a `StopIteration` exception:

```
next(colors)
```

Error:

```
StopIteration
```

If you use the `for` loop, you'll get nothing back. The iterator is empty:

```
for color in colors:
    print(color)
```

To iterate again, you need to create a new `colors` object with the `rgb` attribute. This is inefficient.

Separating an iterator from an iterable

Let's separate the color iterator from its iterable like what Python does with the list iterator and list.

The following defines the `Colors` class:

```
class Colors:
    def __init__(self):
        self.rgb = ['red', 'green', 'blue']

    def __len__(self):
        return len(self.rgb)
```

The following defines the `ColorIterator` class:

```
class ColorIterator:
    def __init__(self, colors):
        self.__colors = colors
        self.__index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.__index >= len(self.__colors):
            raise StopIteration

        # return the next color
        color = self.__colors.rgb[self.__index]
        self.__index += 1
        return color
```

How it works.

- The `__init__` method accepts an iterable which is an instance of the `Colors` class.
- The `__iter__` method returns the iterator itself.
- The __next__ method returns the next element from the `Colors` object.

The following shows how to use the `ColorIterator` to iterate over the `Colors` object:

```
colors = Colors()
color_iterator = ColorIterator(colors)

for color in color_iterator:
    print(color)
```

To iterate the `Colors` object again, you just need to create a new instance of the `ColorIterator` .

There’s one problem!

When you want to iterate the `Colors` object, you need to manually create a new `ColorIterator` object. And you also need to remember the iterator name `ColorIterator` .

It would be great if you can automate this. To do it, you can make the `Colors` class iterable by implementing the `__iter__` method:

```
class Colors:
    def __init__(self):
        self.rgb = ['red', 'green', 'blue']

    def __len__(self):
        return len(self.rgb)

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

The `__iter__` method returns a new instance of the `ColorIterator` class.

Now, you can iterate the `Colors` object without explicitly creating the `ColorIterator` object:

```
colors = Colors()

for color in colors:
    print(color)
```

Internally, the `for` loop calls the `__iter__` method of the `colors` object to get the iterator and uses this iterator to iterate over the elements of the `colors` object.

The following places the `ColorIterator` class inside the `Colors` class to encapsulate them into a single class:

```
class Colors:
    def __init__(self):
        self.rgb = ['red', 'green', 'blue']

    def __len__(self):
        return len(self.rgb)

    def __iter__(self):
        return self.ColorIterator(self)

    class ColorIterator:
        def __init__(self, colors):
            self.__colors = colors
            self.__index = 0

        def __iter__(self):
            return self

        def __next__(self):
            if self.__index >= len(self.__colors):
                raise StopIteration
            # return the next color
            color = self.__colors.rgb[self.__index]
            self.__index += 1
            return color
```

Summary

- An iterable is an object that implements the `__iter__` method which returns an iterator.
- An iterator is an object that implements the `__iter__` method which returns itself and the `__next__` method which returns the next element.
- Iterators are also iterables. However, they're iterables that become exhausted while iterables will never exhausted.