Python

Topics: Iterators and Generators

Iterators and Generators

Tutorial

Python Iterators, generators, and the for loop

Iterators are containers for objects so that you can loop over the objects. In other words, you can run the "for" loop over the object. There are many iterators in the Python standard library. For example, list is an iterator and you can run a for loop over a list.

```
>>> for lib in popular_python_libs:
...     print(lib)
...
requests
scrapy
pillow
SQLAlchemy
NumPy
```

In this tutorial you will get to know:

1. How to create a custom iterator
2. How to create a generator
3. How to run for loops on iterators and generators

Python Iterators and the Iterator protocol

To create a Python iterator object, you will need to implement two methods in your iterator class.

?

__iter__: This returns the iterator object itself and is used while using the "for" and "in" keywords.

__next__: This returns the next value. This would return the StopIteration error once all the objects have been looped through.

Let us create a cool emoticon generator and l iterators.

```
# iterator_example.py
"""
This should give an iterator with a emoticon.
"""

import random

class CoolEmoticonGenerator(object):
    """docstring for CoolEmoticonGenerator."""

    strings = "!@#$^*_-=+?/,.:;~"
    grouped_strings = [("(", ")"), ("<", ">"), ("[", "]"), ("{", "}")]

    def create_emoticon(self, grp):
        """actual method that creates the emoticon"""
        face_strings_list = [random.choice(self.strings) for _ in range(3)]
        face_strings = "".join(face_strings_list)
        emoticon = (grp[0], face_strings, grp[1])
        emoticon = "".join(emoticon)
        return emoticon

    def __iter__(self):
        """returns the self object to be accessed by the for loop"""
        return self

    def __next__(self):
        """returns the next emoticon indefinitely"""
        grp = random.choice(self.grouped_strings)
        return self.create_emoticon(grp)
```

Now you can call the above class as an iterator. Which means you can run the next function on it.

```
from iterator_example import CoolEmoticonGenerator
g = CoolEmoticonGenerator()
print([next(g) for _ in range(5)])
```

Running the program above gives us the following output. The exact output may be different from what you get but it will be similar.

?

```
➜  python3.5 iterator_example.py
['<,~!>', '<;_~>', '<!;@>', '[~=#]', '{?^-}']
```

You can use the KeyboardInterrupt to stop the execution.

Python Generators

Python generator gives us an easier way to create python iterators. This is done by defining a function but instead of the return statement returning from the function, use the "yield" keyword. For example, see how you can get a simple vowel generator below.

```
>>> def vowels():
...     yield "a"
...     yield "e"
...     yield "i"
...     yield "o"
...     yield "u"
...
>>> for i in vowels():
...     print(i)
...
a
e
i
o
u
```

Now let's try and create the CoolEmoticonGenerator.

```
def create_emoticon_generator():
    while True:
        strings = "!@#$^*_-=+?/,.:;~"
        grouped_strings = [("(", ")"), ("<", ">"), ("[", "]"), ("{", "}")]
        grp = random.choice(grouped_strings)
        face_strings_list = [random.choice(strings) for _ in range(3)]
        face_strings = "".join(face_strings_list)
        emoticon = (grp[0], face_strings, grp[1])
        emoticon = "".join(emoticon)
        yield emoticon
```

Now, if you run the generator using the runner below

?

```
from iterator_example import CoolEmoticonGenerator
g = create_emoticon_generator()
print([next(g) for _ in range(5)])
```

You should get the following output

```
➜  python3.5 iterator_example.py
['(+~?)', '<**_>', '($?/)', '[#=+]', '{*=.}']
```

Contributed by: Joydeep Bhattacharjee