Generators and Decorators

Session 07

Session Overview

In this session, you will be able to:

- Describe iterators and iterables
- Explain custom and infinite iterators
- Outline the significance of generators
- List the benefits and uses of generators
- Explain generator expressions
- Explain function and class decorators

Introduction to Generators and Decorators

- Python offers the most advanced but highly useful language features.
- Some of them allow writing smart solutions for solving a few types of mathematical problems in Web applications.
- These features are iterators, generators, and decorators.
- They make it easy to write code for handling mathematical objects and infinite sequences.
- Of all the three features, iterators form the base to comprehend and implement the other two features.

Iterators and Iterables (1-2)

An iterator object has the following properties:

- It holds data in a sequence.
- It contains the `next()` method to return the subsequent element on each call.
- It contains the `__iter__()` method to initialize and return the iterator itself with the `__next__()` method.

Iterators and Iterables (2-2)

Unlike an iterator, an iterable in Python is more generic.

It is a container that exposes either of the methods or both:

`__getitem__(k)`	• Returns the value at the kth position or raise the IndexError exception in absence of value.
`__iter__()`	• Returns an iterator on the data that the iterable holds. So, an object is an iterable if Python can obtain an iterator from it.

Working with Custom Iterators

Create a custom iterator in Python:

Code Snippet:

```
>>> class PowerOfTwo:
        def __init__(self, max = 0):
          self.maximum = max
        def __iter__(self):
          self.number = 0
          return self
        def __next__(self):
          if self.number <= self.max:
               output = 2 ** self.number
               self.number += 1
               return output
          else :
               raise StopIteration
```

Working with Infinite Iterators

Create an infinite iterator:

Code Snippet :

```
>>> int()
>>> it_no = iter(int,1)
>>> next(it_no)
```

Introduction to Generators

A generator is an iterator. It automatically handle all the aforementioned overheads.

A generator, unlike an iterator, is defined using a function notation.

It is defined as a function that returns an iterator object.

It does not return a single value but a sequence of results.

Python automatically remembers the context of a generator.

It knows where its actual location is and what its values are.

Creating Generators (1-2)

It is easy to create a Python generator, as a developer defines it as a standard function with yield statement rather than return statement.

So, if a function contains even one yield statement, it is a generator function.

A generator is a function with the `yield` keyword.

Following is the generator or keyword's syntax: `yield list_expression`.

Creating Generators (2-2)

Differences between generator functions and normal functions:

A generator function contains a single or multiple yield statements. Further, it returns an iterator object but does not trigger execution instantly.

Unlike a normal function whose whole body is executed, execution stops prior to the first instruction in the generator's body.

Using Generators with a Loop

Using a generator function with a loop to reverse a string:

Code Snippet :

```
>>> def str_reverse(test_str):
    lngth = len(test_str)
    for k in range(lngth - 1,-1,-1):
      yield test_str[k]
>>> for letter in str_reverse("python"):
      print(letter)
```

Output:

```
n
o
h
t
y
p
```

Generator Expressions

Generator expression to obtain the results of squaring four numbers:

Code Snippet:

```
>>> test_list = [4,6,1,8]
>>> [a**2 for a in test_list]

     >>> num = (a**2 for a in test_list)
     >>> print (next(num))
     >>> print (next(num))
     >>> print (next(num))
     >>> print (next(num))
```

```
Output
16
36
1
64
```

Uses of Generators and Generator Expressions

Python generators facilitate lazy evaluation or computation on demand.

By implementing generators in managing huge files, developers get the benefit of space efficiency.

This is because only a few parts of the file are run at one point of time.

Generators are also useful for substituting callbacks.

Advantages of Generators and Generator Expressions

Advantages of using generator and generator expressions in Python:

- Easier implementation than iterators
- Memory-efficient
- Smooth handling of data

Decorators

The action of modifying such objects prior to binding them to their names is known as decorating.

A decorator hides two things namely, the function that decorates and the expression abiding by the decorator syntax.

Following is the decorator syntax:

Syntax

```
@mydecorator
def myfunc():
pass
```

Implementing Decorators

- In Python, everything is in the form of an object. Even functions are objects but they have attributes.
- It is possible to bind different names to the same object of a function.
- A developer can even pass functions as arguments to some other function.
- The latter function is known as a higher order function.
- In Python, methods of an object and functions are termed as callable.
- This is because Python can call them during execution.

Decorating Functions with Parameters

Create a general decorator:

Code Snippet:

```
>>> def generic_decorator(func):
      def inner_func(*args, **kwargs):
            print("Inside decorator")
            return func(*args, **kwargs)
      return inner_func
```

Decorating Classes

- A developer can also define a decorator as a class.
- In other words, it is possible to create and use a class decorator.
- In this case, the decorator accepts a class as the parameter (a type object) and returns a changed class.
- Prior to defining a class decorator, it is essential to know about the `__call__()` method of classes.

Summary (1-5)

- Iterators, generators, and decorators make it easy to handle mathematical objects acting as recurrent expressions and infinite sequences.
- All unordered containers and ordered sequences in Python support iteration.
- An iterator refers to an object that facilitates iterating over a set of items.
- An iterator object has `__iter__()` and `__next__()` methods that are implicitly called and raises the `StopIteration` exception.

Summary (2-5)

- Unlike an iterator, an iterable in Python is more generic and has `__getitem__()` and `__iter__()` methods.
- Python supports infinite iterators, which are objects holding endless items, but it is essential to specify a suitable terminating condition.
- The biggest benefit of using an iterator is that it saves memory.
- Generators automatically handle all overheads of an iterator by simplifying the syntax of defining an iterator.

Summary (3-5)

- Generators are iterators, but are defined using a function notation involving the `yield` keyword.
- A `yield` statement stops the function at that point, saves its state (context), and continues to execute upon receiving successive calls.
- Generator expressions are lazy generators producing items only when they are in demand. This make them more efficient in terms of performance and memory usage than a list comprehension.

Summary (4-5)

- The action of modifying such objects prior to binding them to their names is known as decorating.
- A decorator hides two things namely, the function that decorates and the expression abiding by the decorator syntax.
- A decorator accepts a function, adds some kind of logic to it, and returns it.

Summary (5-5)

- It is possible to make general decorators that can function with any number of parameters.
- A class decorator is analogous to a function decorator. However, it is executed at the end of a class statement for rebinding the name of a class to a callable.