Chapter 6 - Iterators and Generators

1. Introduction to Iterators

In Python, **iterators** are objects that allow us to traverse through a collection (like a list, tuple, dictionary) one element at a time. Iterators implement two key methods:

1. `__iter__()`: This method returns the iterator object itself. It's used in loops to initialize the iteration process.
2. `__next__()`: This method returns the next item in the collection. When there are no more items, it raises the `StopIteration` exception.

---

2. Creating an Iterator Class

An **iterator** must implement both `__iter__()` and `__next__()` methods. Let's look at an example where we create a simple iterator to traverse through a custom range of numbers.

2.1 Example: Custom Range Iterator

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

    def __iter__(self):
        return self  # Return the iterator object itself

    def __next__(self):
        if self.current >= self.end:
            raise StopIteration  # Stop the iteration
        self.current += 1
        return self.current - 1

# Using the iterator
for number in MyRange(1, 5):
    print(number)
```

Output:

```
1
2
3
4
```

In this example, the class `MyRange` defines an iterator that starts from a given `start` and iterates until `end`. The `__next__()` method raises a `StopIteration` exception when there are no more elements to return.

---

3. Iterators in Django Models

In Django, iterators can be useful when handling large datasets, such as when fetching records from the database. Instead of loading all the records into memory, which might be inefficient for large datasets, iterators can be used to iterate through database querysets efficiently.

3.1 Example: Using `iterator()` with QuerySets

Django QuerySets support the `iterator()` method, which retrieves rows from the database one at a time, saving memory when dealing with large queries.

```python
from myapp.models import Product

# Using the iterator method to fetch products efficiently
for product in Product.objects.iterator():
    print(product.name, product.price)
```

This approach is more memory efficient because it avoids fetching all records at once.

---

4. Introduction to Generators

Generators are a simple and powerful tool in Python to create iterators. Instead of implementing `__iter__()` and `__next__()` methods manually, we can use Python's `yield` statement to make a function a generator. The `yield` statement produces a value and pauses the function's execution, which allows it to be resumed later when `next()` is called.

4.1 Example: Simple Generator

```
def my_range(start, end):
    while start < end:
        yield start  # Yield the current value and pause
        start += 1

# Using the generator
for number in my_range(1, 5):
    print(number)
```

Output:

```
1
2
3
4
```

Here, the function `my_range` is a generator. Each time `yield` is called, the function's state is saved, and the value is returned to the caller. The function picks up where it left off the next time `next()` is called.

---

5. Advantages of Using Generators

- **Memory Efficient**: Generators produce items one at a time and only when needed. This makes them much more memory efficient than creating and returning entire lists.
- **Lazy Evaluation**: Values are produced on demand, which can be useful for handling large datasets or infinite sequences.
- **Cleaner Code**: Using `yield` makes the code more concise and avoids the need to explicitly manage an internal iterator.

5.1 Example: Generator with Infinite Sequence

```
def infinite_counter():
    count = 0
    while True:
        yield count
        count += 1

# Using the generator
gen = infinite_counter()
```

```
for i in range(5):
    print(next(gen))
```

Output:

```
0
1
2
3
4
```

In this case, the generator produces an infinite sequence of numbers, but we can control how many numbers we want by using `next()`.

---

6. Using Generators for Django Tasks

Generators are particularly useful when dealing with tasks like:

- Streaming large datasets from a database
- Implementing paginated data fetching for APIs

For example, let's imagine you want to create a paginated API for fetching `Product` records from your database in chunks:

6.1 Example: Paginated Generator for API

```
from myapp.models import Product

def get_paginated_products(page_size):
    page = 0
    while True:
        products = Product.objects.all()[page * page_size: (page + 1) *
page_size]
        if not products:
            break
        for product in products:
            yield product
        page += 1

# Fetching products using the generator
```

```
for product in get_paginated_products(10):
    print(product.name)
```

In this example, we use a generator to fetch products in chunks of 10. Each time `next()` is called, a new page of products is retrieved.

---

7. Exercise

Exercise 1: Implementing a Custom Iterator

Create a class `BookShelf` that holds a list of books and allows you to iterate over the books. Implement the `__iter__()` and `__next__()` methods.

```
class BookShelf:
    def __init__(self, books):
        self.books = books
        self.current = 0

    def __iter__(self):
        return self  # Return the iterator object itself

    def __next__(self):
        if self.current >= len(self.books):
            raise StopIteration  # Stop the iteration
        book = self.books[self.current]
        self.current += 1
        return book

# Test your implementation
shelf = BookShelf(["Python Basics", "Advanced Django", "Data Science"])
for book in shelf:
    print(book)
```

Exercise 2: Generator for Fibonacci Sequence

Create a generator `fibonacci(n)` that generates the Fibonacci sequence up to the nth number.

```python
def fibonacci(n):
    a, b = 0, 1
    while n > 0:
        yield a
        a, b = b, a + b
        n -= 1

# Test your generator
for num in fibonacci(10):
    print(num)
```

Exercise 3: Django QuerySet Generator

Create a generator that fetches products from the database in chunks of 5 and prints their names.

```python
from myapp.models import Product

def fetch_products_in_chunks():
    page = 0
    while True:
        products = Product.objects.all()[page * 5 : (page + 1) * 5]
        if not products:
            break
        for product in products:
            yield product
        page += 1

# Test the generator
for product in fetch_products_in_chunks():
    print(product.name)
```