Iterators in Python

A Theoretical Overview with Examples

Bhimashankar Takalki

Introduction to Iterators

In Python, an iterator is an object that represents a stream of data. It enables you to traverse through a sequence of elements, such as lists, tuples, strings, and more, one at a time. Iterators are used extensively in Python, often in conjunction with loops, to efficiently process and manipulate data.

Theory of Iterators:

1. **Iterable Objects**: An iterable is any object capable of returning its elements one at a time. Examples of iterables include lists, tuples, strings, dictionaries, sets, and more.
2. **Iterator Protocol**: In Python, the iterator protocol is a standard way to make objects iterable. It involves implementing two methods:
   - `__iter__()`: Returns the iterator object itself and is called at the start of the iteration.
   - `__next__()`: Returns the next element from the iterable. When there are no more elements, it raises the `StopIteration` exception.
3. **Iterating with Iterators**: Iterators are commonly used with loops, such as `for` loops, to iterate over the elements of an iterable. The loop repeatedly calls the iterator's `__next__()` method until the `StopIteration` exception is raised.

Example 1: Creating an Iterator Class

```
class MyIterator:
    def __init__(self, data):
        self.data = data
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.data):
            raise StopIteration
        value = self.data[self.index]
        self.index += 1
        return value

my_iter = MyIterator([1, 2, 3, 4, 5])
for item in my_iter:
    print(item)
```

Example 2: Using built-in Iterators

```
my_list = [1, 2, 3, 4, 5]
my_iterator = iter(my_list)  # Creates an iterator from the list
print(next(my_iterator))     # Output: 1
print(next(my_iterator))     # Output: 2
```

Example 3: Using Iterators with Generators

```
def my_generator():
    yield 1
    yield 2
    yield 3

gen_iterator = my_generator()  # Creates an iterator from the
generator
print(next(gen_iterator))      # Output: 1
print(next(gen_iterator))      # Output: 2
```

Example 4: Using built-in Iterators with Iterable Objects

```
my_string = "Hello"
str_iterator = iter(my_string)  # Creates an iterator from the
string
print(next(str_iterator))      # Output: 'H'
print(next(str_iterator))      # Output: 'e'
```

These examples demonstrate the creation and usage of iterators in Python, illustrating how they can be implemented using custom classes, built-in iterables, generators, and iterable objects. Iterators are fundamental for efficient data processing and manipulation in Python.