ITERATORS IN PYTHON

An Introduction to Iteration in Python

MUKESH KUMAR

Agenda

- ✓ Introduction to Iterators
- ✓ The Iterator Protocol
- ✓ Creating an Iterator
- ✓ Using an Iterator
- ✓ Built-in Iterators in Python
- ✓ Practical Use Case of Iterators
- ✓ Advantages of Iterators
- ✓ Common Mistakes

Introduction to Iterators

- An iterator is an object in Python which implements the iterator protocol, consisting of the methods __iter__() and __next__().

- Iterators allow you to traverse through all the items in a collection like lists, tuples, and dictionaries without the need for an index.

The Iterator Protocol

- **__iter__() Method**: Returns the iterator object itself and is used in situations where an iterable needs to be accessed.

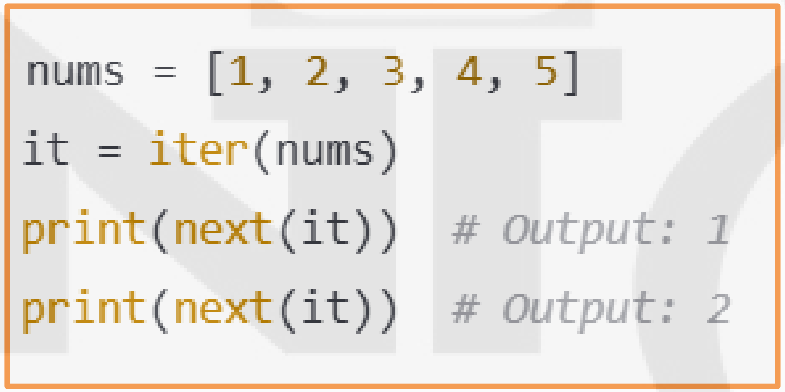
- **__next__() Method:** Returns the next item from the container. Once all items are exhausted, it raises a **StopIteration** exception.

Iterator Example

- Refer notebook iterator Example .ipynb

Built-in Iterators in Python

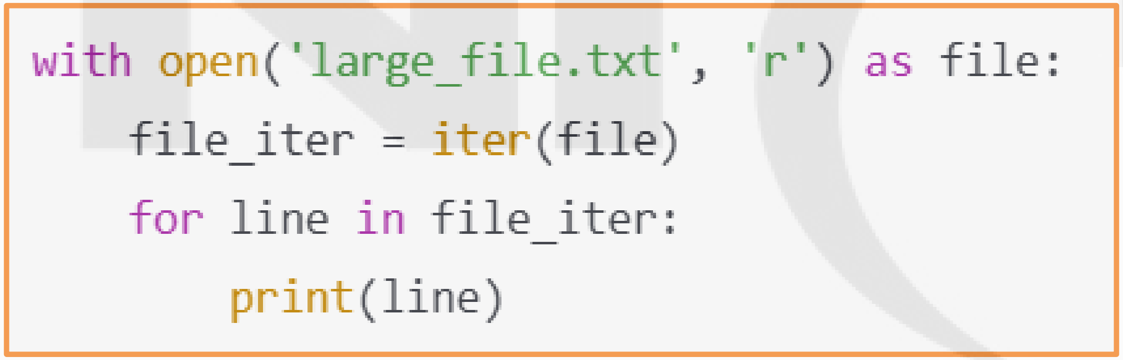
- Refer Notebook : Inbuilt_Iterators_Python.ipynb



```
nums = [1, 2, 3, 4, 5]
it = iter(nums)
print(next(it))  # Output: 1
print(next(it))  # Output: 2
```

Practical Use Case of Iterators

- Iterating over large datasets without loading everything into memory at once.
- Example:



```
with open('large_file.txt', 'r') as file:
    file_iter = iter(file)
    for line in file_iter:
        print(line)
```

Advantages of Iterators

- **Memory Efficient:** Iterators do not require all items to be in memory at once.
- **Lazy Evaluation:** They generate items on the fly as needed.
- **Infinite Sequences:** Iterators can be used to work with infinite sequences (e.g., Fibonacci).

Common Mistakes

- Forgetting to raise **StopIteration** in custom iterators.
- Calling **next()** on an exhausted iterator without handling **StopIteration**.

Conclusion

- Iterators and generators are powerful tools for efficient iteration in Python, especially when dealing with large datasets or infinite sequences.