**Iterator Functions in Python**

An **iterator** in Python is an object that implements the **iterator protocol**, which consists of two methods:

1. \_\_iter\_\_() – Returns the iterator object itself.
2. \_\_next\_\_() – Returns the next value from the iterator. When there are no more items, it raises a StopIteration exception.

**Creating and Using Iterators**

Python has built-in iterators such as lists, tuples, dictionaries, and sets, but you can also create your own iterators.

**1. Using Built-in Iterators**

Many built-in Python objects like lists, tuples, and dictionaries can be iterated over using iterators.

my\_list = [1, 2, 3, 4]

iterator = iter(my\_list) # Get an iterator object

print(next(iterator)) # 1

print(next(iterator)) # 2

print(next(iterator)) # 3

print(next(iterator)) # 4

# next(iterator) would raise StopIteration

**2. Creating a Custom Iterator**

To create a custom iterator, define a class with \_\_iter\_\_() and \_\_next\_\_() methods.

class Counter:

def \_\_init\_\_(self, start, end):

self.current = start

self.end = end

def \_\_iter\_\_(self):

return self # Returns the iterator object itself

def \_\_next\_\_(self):

if self.current > self.end:

raise StopIteration # End iteration

else:

self.current += 1

return self.current - 1

# Usage

counter = Counter(1, 5)

for num in counter:

print(num) # Output: 1 2 3 4 5

**Iterator Functions in Python**

Python provides several iterator functions in the itertools module to work efficiently with iterators.

**1. count(start, step)**

Creates an infinite iterator that starts from start and increments by step.

from itertools import count

for num in count(5, 2):

if num > 15:

break

print(num) # Output: 5, 7, 9, 11, 13, 15

**2. cycle(iterable)**

Repeats an iterable indefinitely.

from itertools import cycle

counter = cycle(["A", "B", "C"])

for \_ in range(6):

print(next(counter)) # Output: A, B, C, A, B, C

**3. repeat(value, times)**

Repeats a value for a specified number of times.

from itertools import repeat

for num in repeat("Hello", 3):

print(num) # Output: Hello, Hello, Hello

**4. chain(\*iterables)**

Combines multiple iterables into a single iterator.

from itertools import chain

a = [1, 2, 3]

b = ["A", "B", "C"]

c = chain(a, b)

print(list(c)) # Output: [1, 2, 3, 'A', 'B', 'C']

**5. islice(iterable, start, stop, step)**

Slices an iterator similar to list slicing.

from itertools import islice

nums = range(10)

print(list(islice(nums, 2, 8, 2))) # Output: [2, 4, 6]

**6. combinations(iterable, r)**

Generates all possible combinations of length r.

from itertools import combinations

items = ['A', 'B', 'C']

print(list(combinations(items, 2))) # Output: [('A', 'B'), ('A', 'C'), ('B', 'C')]

**7. permutations(iterable, r)**

Generates all possible ordered arrangements of length r.

from itertools import permutations

print(list(permutations(['A', 'B', 'C'], 2)))

# Output: [('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]

**8. groupby(iterable, key)**

Groups consecutive identical elements together.

from itertools import groupby

data = "AAABBBCCDA"

groups = [(k, list(g)) for k, g in groupby(data)]

print(groups)

# Output: [('A', ['A', 'A', 'A']), ('B', ['B', 'B', 'B']), ('C', ['C', 'C']), ('D', ['D']), ('A', ['A'])]

**Generators as Iterators**

A generator is a simpler way to create iterators using the yield keyword.

def my\_generator():

yield 1

yield 2

yield 3

gen = my\_generator()

print(next(gen)) # 1

print(next(gen)) # 2

print(next(gen)) # 3

**Generator with yield**

def countdown(n):

while n > 0:

yield n

n -= 1

for num in countdown(5):

print(num) # Output: 5, 4, 3, 2, 1

**Conclusion**

Iterators in Python allow efficient looping and memory management, especially when dealing with large datasets. The itertools module provides powerful iterator functions for different use cases. You can also create your own iterators using classes or generators.