**Module 8: Generators and iterators**

In Python, generators and iterators are tools that allow you to iterate over data efficiently, especially when dealing with large datasets or streams of data. They help in lazy evaluation, meaning they produce items one at a time and only when required, instead of generating all items at once.

**1. Iterators**

An **iterator** is an object that allows you to traverse through all the elements of a collection (like lists, tuples, or strings) one element at a time, without needing to know the underlying structure.

**Key Points about Iterators:**

* Implements two methods:
  + \_\_iter\_\_() – Returns the iterator object itself.
  + \_\_next\_\_() – Returns the next element; raises StopIteration when there are no more elements.
* Created from an iterable (like a list) using iter() function.

**1.Example of an Iterator:**

# Creating an iterator from a list

numbers = [1, 2, 3, 4]

it = iter(numbers) # Get an iterator

print(next(it)) # Output: 1

print(next(it)) # Output: 2

print(next(it)) # Output: 3

print(next(it)) # Output: 4

# print(next(it)) # Raises StopIteration

2. Example

**ReverseIterator**

nums = [1, 2, 3, 4]

for n in reversed(nums):

    print(n)

# Output: 4 3 2 1

**2. Generators**

A **generator** is a special type of iterator that is defined using a **function with yield keyword** instead of return.

**Key Points about Generators:**

* Automatically implements the iterator protocol (\_\_iter\_\_ and \_\_next\_\_).
* State of the function is saved between yield calls.
* They are **memory efficient** because they generate values on the fly.

**1. Example of a Generator:**

def my\_generator():

yield 1

yield 2

yield 3

gen = my\_generator()

print(next(gen)) # Output: 1

print(next(gen)) # Output: 2

print(next(gen)) # Output: 3

**2.Example**

Even Number Generator

def even\_numbers(limit):

for num in range(2, limit + 1, 2):

yield num

# Use the generator

for num in even\_numbers(10):

print(num, end=" ")# Output: 0 2 4 6 8 10

**3.The Functions any and all**

In Python, any() and all() are built-in functions that are particularly useful when working with generators and iterators because they consume elements lazily without creating intermediate lists.

**1. any() Function**

* **Definition:**  
  any(iterable) returns True if **at least one element** in the iterable is True.  
  It stops checking as soon as it finds a True value (short-circuit evaluation).
* **Usage with Generators:**  
  Generators are evaluated lazily. Using any() with a generator expression is **memory efficient** because elements are checked one by one without building a list.

**Example:**

numbers = [1, 6, 3, 2]

result = any(num > 5 for num in numbers) # Generator expression

print(result) # Output: True

Here, any() stops as soon as it finds num > 5 is True.

**2. all() Function**

* **Definition:**  
  all(iterable) returns True if **all elements** in the iterable are True.  
  It stops checking as soon as it finds a False value.
* **Usage with Generators:**  
  Similar to any(), it works efficiently with generators.

**Example:**

numbers = [2, 4, 6, 8]

result = all(num % 2 == 0 for num in numbers) # Generator expression

print(result) # Output: True

**3. Why Use any() and all() with Generators?**

* **Memory Efficient:** Unlike lists, generators don’t store all values in memory.
* **Short-Circuiting:** Stops early when the result is determined, saving computation.

**Comparison**

| **Function** | **Returns True When** | **Stops Checking When** |
| --- | --- | --- |
| any() | At least one True | Finds a True |
| all() | All values are True | Finds a False |

**Using with in Generators**

A generator can include a with statement inside its body when dealing with resources. For example, if a generator reads from a file, you can use with open() to ensure the file closes automatically.

**Example**

def read\_file\_line\_by\_line(filename):

    with open(filename, 'r') as f:

        for line in f:

            yield line.strip()

for line in read\_file\_line\_by\_line('example2.txt'):

    print(line)

**Explanation:**

* The generator read\_file\_line\_by\_line uses a with statement to open a file.
* When the generator finishes or is stopped, the file is automatically closed due to the context manager.

**Using contextlib.contextmanager with Generators**

Python’s contextlib.contextmanager decorator allows you to **write context managers using generators** instead of creating a full class with \_\_enter\_\_ and \_\_exit\_\_ methods.

**Example**

from contextlib import contextmanager

@contextmanager

def open\_file(filename, mode):

f = open(filename, mode)

try:

yield f

finally:

f.close()

# Using with statement

with open\_file(‘example2.txt', 'w') as file:

file.write("Hello Python!")

**Example: Iterator with “with”**

class FileIterator:

def \_\_init\_\_(self, filename):

self.filename = filename

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

with open(self.filename, 'r') as f:

for line in f:

yield line.strip()

# Using the iterator

for line in FileIterator('sample.txt'):

print(line)

**Data Compression**

Data compression in the context of generators and iterators means selecting specific elements from an iterable based on certain criteria, such as a condition or a boolean mask, without creating unnecessary intermediate data structures.

In Python, data compression can be efficiently implemented using:

1. itertools.compress()
2. Generator Expressions with Conditions

**1. itertools.compress()**

The compress() function from the itertools module filters elements from an iterable based on a selector list of Boolean values.

Syntax:

itertools.compress(data, selectors)

data: The iterable you want to filter.

selectors: An iterable of boolean values, where True means the element is included and False means it is excluded.

**Example:**

from itertools import compress

data = ['A', 'B', 'C', 'D', 'E']

selectors = [1, 0, 1, 0, 1] # Only include elements where the selector is 1 (True)

result = compress(data, selectors)

print(list(result)) # Output: ['A', 'C', 'E']

1. **Using Generators for Compression**

Instead of using compress(), you can create a generator that yields elements based on a condition or boolean mask.

Example:

data = ['A', 'B', 'C', 'D', 'E']

selectors = [1, 0, 1, 0, 1]

compressed\_gen = (item for item, sel in zip(data, selectors) if sel)

print(list(compressed\_gen)) # Output: ['A', 'C', 'E']

This method is memory-efficient because it doesn't store the whole filtered list in memory; it yields items one by one.

**3.Conditional Data Compression with Generators**

You can filter data directly using conditions in a generator expression.

Example:

numbers = [10, 20, 30, 40, 50]

# Compress data where numbers > 25

compressed\_numbers = (num for num in numbers if num > 25)

print(list(compressed\_numbers)) # Output: [30, 40, 50]

**4. Combining Iterators and Compress**

You can combine iterators with compress() for advanced filtering.

**Example:**

from itertools import compress

letters = 'ABCDE'

selectors = [True, False, True, True, False]

compressed = compress(letters, selectors)

for item in compressed:

print(item)