😊 Iterators and Generators

Iterators

An **iterator** in Python is an object that allows traversal through a sequence (like a list or tuple) one element at a time. It follows the **Iterator Protocol**, which means it must implement two methods:

- __iter__(): Returns the iterator object itself.
- __next__(): Returns the next element in the sequence. Raises StopIteration when there are no more elements.

Use Cases of Iterators:

1. **Memory Efficiency:** Instead of loading all elements at once, iterators fetch one item at a time.
2. **Custom Iteration:** Custom classes can implement iterators to traverse objects in a specific manner.
3. **Loop Control:** Used in for loops, comprehensions, and functions like map() and filter().

Generators

A **generator** is a special type of iterator that simplifies the process of creating iterators using functions instead of classes. It allows iteration using the yield keyword instead of returning values.

Characteristics of Generators:

- Uses the yield keyword instead of return, allowing it to pause execution and resume later.
- Automatically implements the __iter__() and __next__() methods.
- More **memory efficient** than normal iterators since values are generated on demand.

Use Cases of Generators:

1. **Efficient Data Streaming:** Used for handling large datasets (like reading large files).
2. **Lazy Evaluation:** Only produces values when needed, reducing memory usage.
3. **Infinite Sequences:** Can generate endless sequences without exhausting memory.
4. **Pipeline Processing:** Useful in processing large streams of data efficiently.

Key Differences Between Iterators and Generators:

Feature	Iterator	Generator
Creation	Implemented using a class and methods	Created using a function with yield
Memory Usage	Can store all values in memory	Generates values on demand (memory efficient)
Complexity	Requires defining __iter__() and __next__()	Uses a simple function
Performance	Can be slower	Generally faster due to lazy evaluation

Generators provide a more **Pythonic** and **efficient** way of working with large data streams and complex iterations without manually implementing iterators.

😄 To begin with the Lab

1. This code defines a generator function named **gencubes** that produces the cube of each number from 0 up to (but not including) n. The yield statement returns each cube one by one as needed, making it memory efficient. The for loop iterates over the generated cubes and prints each one.

[2]:
```
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num**3
```

[4]:
```
for x in gencubes(10):
    print(x)
```

```
0
1
8
27
64
125
216
343
512
729
```

2. This code defines a generator function called genfibon that produces the Fibonacci sequence. It initializes two variables a and b to 1. In each iteration of the loop (running n times), it yields a, then updates a and b so that a takes the value of b and b becomes the sum of the old a and b. When you call genfibon(10), it prints the first 10 Fibonacci numbers.

[6]:
```
def genfibon(n):
    """
    Generate a fibonnaci sequence up to n
    """
    a = 1
    b = 1
    for i in range(n):
        yield a
        a,b = b,a+b
```

[8]:
```
for num in genfibon(10):
    print(num)
```

```
1
1
2
3
5
8
13
21
34
55
```

3. This function generates the first n numbers in the Fibonacci sequence and returns them as a list. It starts with the first two Fibonacci numbers set to 1, then iterates n times. In each iteration, it appends the current number to the list and updates the two numbers to the next ones in the sequence. Calling fibon(10) returns the first 10 Fibonacci numbers.

[10]:
```
def fibon(n):
    a = 1
    b = 1
    output = []

    for i in range(n):
        output.append(a)
        a,b = b,a+b

    return output
```

[12]:
```
fibon(10)
```

[12]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

4. This code defines a simple generator function simple_gen that yields numbers from 0 to 2. Each time you call next(g), it returns the next value. After it has yielded all three values (0, 1, and 2), calling next(g) again raises a StopIteration exception because there are no more values to generate.

[14]:
```
def simple_gen():
    for x in range(3):
        yield x
```

[16]:
```
# Assign simple_gen
g = simple_gen()
```

[18]:
```
print(next(g))
```

```
0
```

[20]:
```
print(next(g))
```

```
1
```

[22]:
```
print(next(g))
```

```
2
```

[24]:
```
print(next(g))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[24], line 1
----> 1 print(next(g))

StopIteration:
```

5. The string 'hello' is an iterable but not an iterator. The for loop internally creates an iterator to go through the characters. When you call next(s) directly, it raises an error because s itself does not support the next() method. To use next(), you must first convert the string to an iterator using iter(s).

[26]:
```
s = 'hello'

#Iterate over string
for let in s:
    print(let)
```

```
h
e
l
l
o
```

But that doesn't mean the string itself is an *iterator*! We can check this with the next() function:

[28]:
```
next(s)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[28], line 1
----> 1 next(s)

TypeError: 'str' object is not an iterator
```

6. Here, the string "hello" is converted into an iterator, which lets you access its elements one at a time. The first call to next(s_iter) returns the first character, "h", and the second call returns the next character, "e".

[30]:
```
s_iter = iter(s)
```

[32]:
```
next(s_iter)
```

[32]:
```
'h'
```

[34]:
```
next(s_iter)
```

[34]:
```
'e'
```