**Generator Functions in Python**

A **generator function** in Python is a special kind of function that **yields** values one at a time using the yield keyword instead of returning a single value with return. Generator functions are memory-efficient and useful for working with large datasets or infinite sequences.

**1. Defining a Generator Function**

A generator function is defined like a normal function but contains one or more yield statements.

**Example: A Simple Generator**

def count\_up\_to(n):

count = 1

while count <= n:

yield count # Pauses the function and returns the value

count += 1

# Using the generator

gen = count\_up\_to(5)

for num in gen:

print(num)

**Output:**

1

2

3

4

5

**How it Works**

* When yield is encountered, the function's state is **saved**.
* Execution resumes from the last yield when the generator is called again.

**2. Generator vs. Normal Function**

A normal function returns all values at once:

def normal\_function(n):

result = []

for i in range(n):

result.append(i)

return result

print(normal\_function(5)) # [0, 1, 2, 3, 4]

Whereas a generator **lazily** produces values one by one:

def generator\_function(n):

for i in range(n):

yield i

gen = generator\_function(5)

print(list(gen)) # [0, 1, 2, 3, 4]

**3. Generator Object and Iteration**

A generator **does not execute immediately**; instead, it returns an **iterator**.

**Manually Iterating Using next()**

gen = count\_up\_to(3)

print(next(gen)) # 1

print(next(gen)) # 2

print(next(gen)) # 3

# print(next(gen)) # Raises StopIteration

Once the generator is exhausted, calling next() raises a StopIteration exception.

**4. Infinite Generators**

Generators can create **infinite sequences** efficiently.

def infinite\_counter():

num = 1

while True:

yield num

num += 1

gen = infinite\_counter()

print(next(gen)) # 1

print(next(gen)) # 2

print(next(gen)) # 3

To prevent infinite loops, use conditions or break.

**5. Generator Expressions**

Generator expressions provide a **concise way** to create generators.

gen\_exp = (x\*\*2 for x in range(5))

print(next(gen\_exp)) # 0

print(next(gen\_exp)) # 1

print(list(gen\_exp)) # [4, 9, 16] (Remaining elements)

Similar to list comprehensions but more memory-efficient.

**6. Sending Values to Generators**

Generators can also receive input using send().

def greet():

name = yield "Enter your name: "

yield f"Hello, {name}!"

gen = greet()

print(next(gen)) # "Enter your name: "

print(gen.send("Alice")) # "Hello, Alice!"

**7. Generator Pipelines**

Generators can be chained together to process large data efficiently.

def numbers():

for i in range(10):

yield i

def squared(numbers):

for num in numbers:

yield num \*\* 2

result = squared(numbers())

print(list(result)) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

**8. When to Use Generators?**

✅ When dealing with **large datasets** (e.g., reading large files line by line).  
✅ When **streaming** data (e.g., real-time logs, sensor data).  
✅ When working with **infinite sequences**.  
✅ When aiming for **memory efficiency**.

**Conclusion**

* Generators use yield to return values lazily.
* They are **iterators** and save memory compared to lists.
* Useful for **infinite sequences, streaming data, and efficient computations**.