What is a Generator Function?

A **generator function** is a special type of function in Python that **does not return all values at once**. Instead, it produces values **one at a time** using the `yield` keyword.

Think of it like a **water tap** that gives one drop at a time **when requested**, instead of a bucket that spills all the water at once.

Example of a Generator Function

```
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()  # This creates a generator object
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
```

What is a Generator Object?

A **generator object** is what you get when you call a generator function. But unlike normal functions that return all results instantly, a generator object **remembers its last state** and continues from there.

- In the example above, `gen = my_generator()` creates a **generator object**.
- Using `next(gen)`, we ask for the next value **one at a time**.

If we keep calling `next()`, Python will **pause** after each `yield` and resume from where it left off.

What Happens If There Are No More Values?

```
print(next(gen))  # This will raise a StopIteration error
```

Once all values are "yielded," the generator is **exhausted**.

Why Use Generators?

1. **Saves Memory** – Instead of storing all values in memory, it **produces one at a time**.
2. **Faster Execution** – It **starts giving values immediately** instead of waiting to process everything.
3. **Great for Large Data** – Works well for **huge datasets** (e.g., reading large files).

Real-Life Example: Reading a Large File

If we use:

```
def read_file(filename):
    with open(filename, "r") as f:
        for line in f:
            yield line
```

- Instead of reading **all lines at once** (which is memory-intensive), this generator **gives one line at a time**.

Difference Between `return` and `yield`

Feature	`return` (Normal Function)	`yield` (Generator Function)
Execution	Stops immediately	Pauses and continues
Memory Usage	Stores all values	Produces one at a time
Usage	One-time use	Can resume from last state

****

Is `row for row in list1` a Generator?

It depends on how you write it. Let's break it down:

1️⃣ List Comprehension (NOT a Generator)

```
list1 = [[1, 2], [3, 4], [5, 6]]
result = [row for row in list1]
print(result)
```

✅ Output:

```
[[1, 2], [3, 4], [5, 6]]
```

- This **immediately creates a list** and stores all rows in memory.
- This is **NOT a generator**.

2️⃣ Generator Expression (This is a Generator!)

```
gen = (row for row in list1)
print(gen)  # Prints the generator object
print(next(gen))  # First row
print(next(gen))  # Second row
```

✅ **Output:**

```
<generator object <genexpr> at 0x7f8b1c2d5d60>
[1, 2]
[3, 4]
```

- This **does NOT create a full list immediately**.
- Instead, it **remembers where it left off** and gives **one row at a time** when `next()` is called.
- This **saves memory** because it does not store all rows at once.

How to Check If Something is a Generator?

Use `type()` or `isinstance()`:

```
gen = (row for row in list1)
print(type(gen))  # Output: <class 'generator'>
print(isinstance(gen, (list, generator)))  # True
```

Key Difference

Syntax	List Comprehension `[row for row in list1]`	Generator Expression `(row for row in list1)`
Storage	Stores **all** values in memory	Produces **one** value at a time
Speed	Faster for small lists	Better for large data
Usage	Use when memory is not a concern	Use when handling big data

****

print(*(row for row in matrix), sep="\n") what is * here??

The `*` (asterisk) in `print(*(row for row in matrix), sep="\n")` is called the **unpacking operator** in Python.

What Does `*` Do Here?

- `*` **unpacks** the generator expression `(row for row in matrix)`, meaning it extracts all the elements from it and passes them as separate arguments to `print()`.
- Normally, `print()` takes multiple arguments and prints them separated by a space by default. But here, we specify `sep="\n"`, so each row gets printed on a new line.

Step-by-Step Execution:

Let's assume:

```
matrix = [[1, 2], [3, 4], [5, 6]]
```

1. `(row for row in matrix)` is a **generator expression** that produces:

   ```
   [1, 2], [3, 4], [5, 6]
   ```

2. `*` unpacks this generator, so it's equivalent to:

   ```
   print([1, 2], [3, 4], [5, 6], sep="\n")
   ```

3. Since `sep="\n"`, the output is:

   ```
   [1, 2]
   [3, 4]
   [5, 6]
   ```

Without `*`:

If you just do:

```
print((row for row in matrix), sep="\n")
```

it prints something like:

```
<generator object <genexpr> at 0x7f8b1c2d5d60>
```

because `print()` treats the generator as a single object instead of executing it.

Alternative Equivalent:

Instead of `*`, you can use `join()`:

```
print("\n".join(str(row) for row in matrix))
```

or a simple loop:

```
for row in matrix:
    print(row)
```

****

Some more doubts:

Explanation of `print("\n".join(str(row) for row in matrix))`

1. **List Comprehension Execution:**

- The part inside `join()`, i.e., `(str(row) for row in matrix)`, is a **generator expression** that iterates over each `row` in `matrix` and converts it to a string.
- Example: If `matrix = [[1, 2], [3, 4]]`, the generator produces `"[1, 2]"` and `"[3, 4]"`.

2. **Joining Strings with Newlines:**
   - `"\n".join(...)` takes the string values generated above and joins them with `\n` (newline).
   - Output:
     ```
     [1, 2]
     [3, 4]
     ```

Why Doesn't `str(row) for row in matrix` Give Values?

If you just write:

```
str(row) for row in matrix
```

it's just a generator expression, but it does nothing on its own. Python does not automatically execute or print the values from a generator expression unless it is explicitly used in a function like `list()`, `join()`, or a loop.

To see the values, you can do:

```
print(list(str(row) for row in matrix))
```

This would print:

```
['[1, 2]', '[3, 4]']
```

because `list()` forces the generator to produce values.

Why Doesn't `print(row for row in matrix)` Print Row Values?

If you write:

```
print(row for row in matrix)
```

It's like saying:

```
Print a machine that can produce rows, but don't actually make the rows.
```

Python will just show something like:

```
<generator object <genexpr> at 0x7f8b1c2d5d60>
```

❗ The generator **exists**, but it hasn't actually **produced** any values yet!

To actually **print** each row:

```
for row in matrix:
    print(row)
```

OR

```
print(*(row for row in matrix), sep="\n")
```

OR

```
print("\n".join(str(row) for row in matrix))
```

✅ This will **force Python to print each row properly**.