GENERATORS IN PYTHON

A concise and memory-efficient way to handle iteration

MUKESH KUMAR

What Are Generators?

- Generators are special functions in Python that return an iterator
- Unlike regular functions, they use `yield` instead of `return`
- Useful for handling large datasets without loading everything into memory at once
- Commonly used in data processing, streaming large files, and implementing infinite sequences

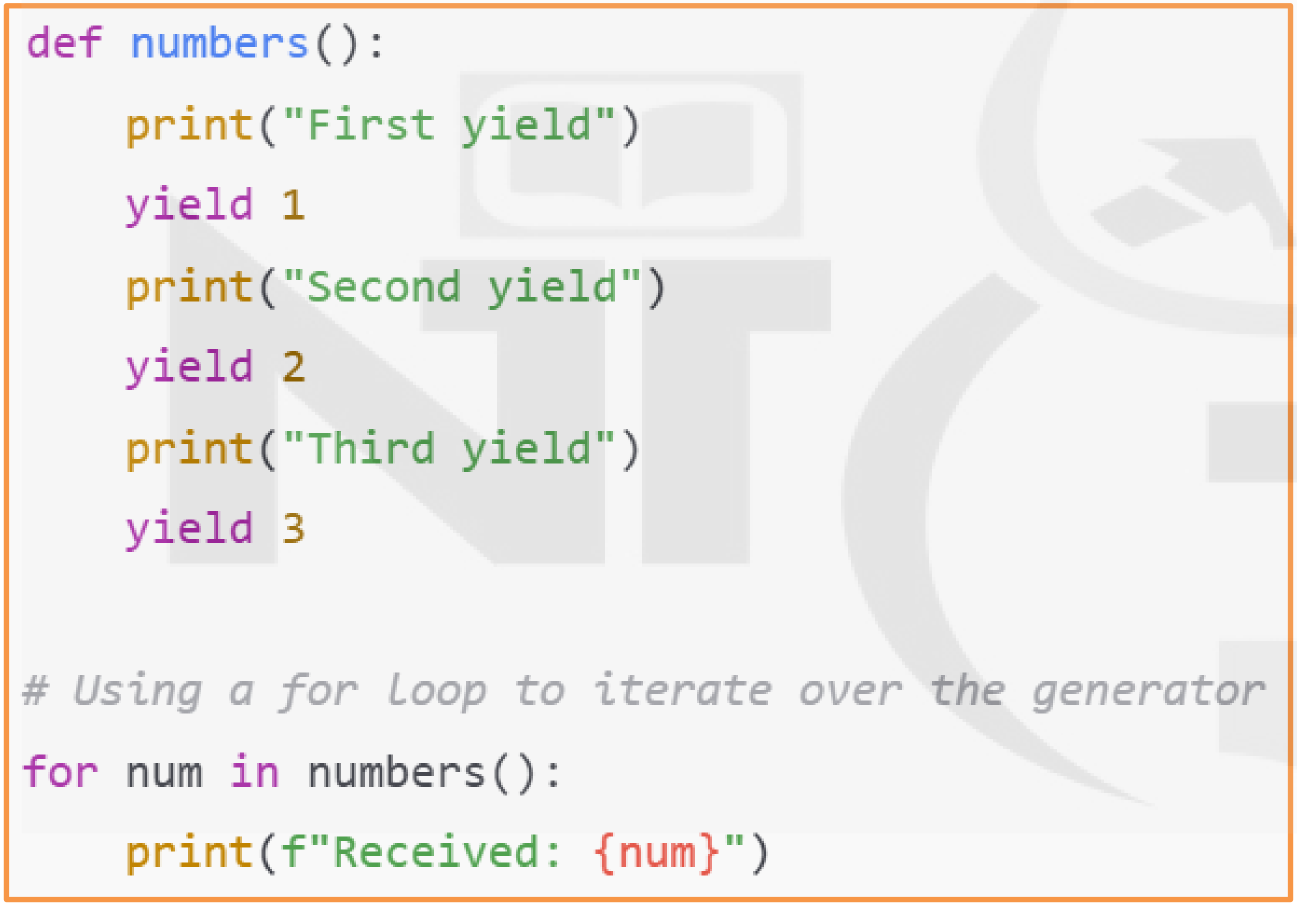
What are Generators

- A generator is a special type of iterator that is created using a function and the yield keyword.

- Unlike iterators, generators do not store all values in memory; they generate values on the fly.

- When yield is used, the function pauses execution and retains its state for the next call.

Generator Syntax



```python
def numbers():

    print("First yield")

    yield 1

    print("Second yield")

    yield 2

    print("Third yield")

    yield 3


# Using a for Loop to iterate over the generator

for num in numbers():

    print(f"Received: {num}")
```

- Show working in jupyter notebook:
  - Generator_working.ipyng

How Generators Work

- When a generator function is called, it does not execute immediately; instead, it returns a generator object
- The function’s execution is paused at each `yield` statement, resuming from the same state when called again
- Helps in efficient memory utilization by generating values lazily

Why Use Generators?

- Reduce memory usage by yielding values one at a time
- Improve performance for large data sets
- Enable lazy evaluation
- Useful for handling infinite sequences

Understanding Iterators

- Any object that implements the `__iter__()` and `__next__()` methods
- Used to iterate over sequences like lists, tuples, and dictionaries
- Requires storing all data in memory (unless a custom iterator is implemented)

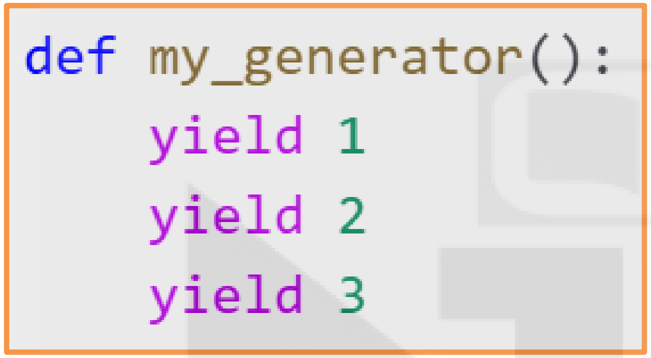
What Makes Generators Different?

- A special type of iterator created using a function with `yield`
- Automatically handles state persistence between iterations
- More memory-efficient, as it generates values lazily

Key Differences Between Generators and Iterators

- Generators are easier to implement and require less code
- Generators pause execution and resume from the last `yield`, while iterators fetch the next element explicitly
- Generators do not store all elements in memory, whereas iterators might (depending on the implementation)

Creating a Generator Function



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

- Uses `yield` to return values lazily
- Suspends state between calls

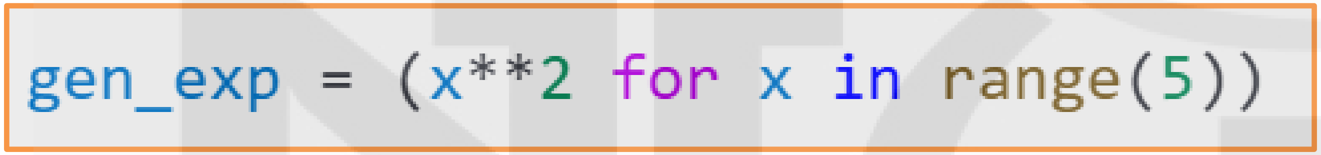
Show Practice problems notebook

Using `yield` in Generators

- `yield` pauses function execution
- State is remembered between calls
- Execution resumes from the last yield statement

Generator Expressions

- Similar to list comprehensions but use parentheses `()`



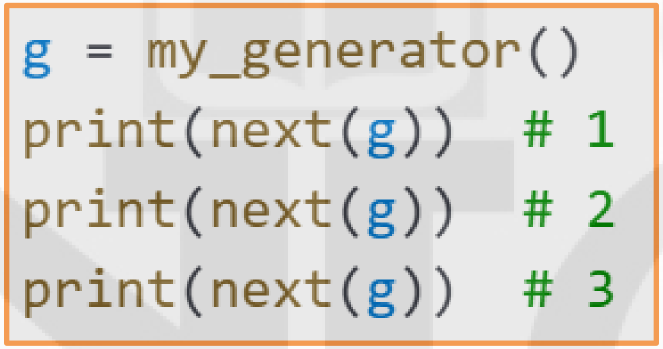
```
gen_exp = (x**2 for x in range(5))
```

- More memory efficient than list comprehensions

Generator Expression Examples

- Show notebook:
  - Additonal_Practice_Questions_Generators_Comprehension.ipynb

Using `next()` and `for` Loop with Generators



```
g = my_generator()
print(next(g))  # 1
print(next(g))  # 2
print(next(g))  # 3
```

- `next()` fetches the next item from the generator
- `for` loop automatically handles StopIteration

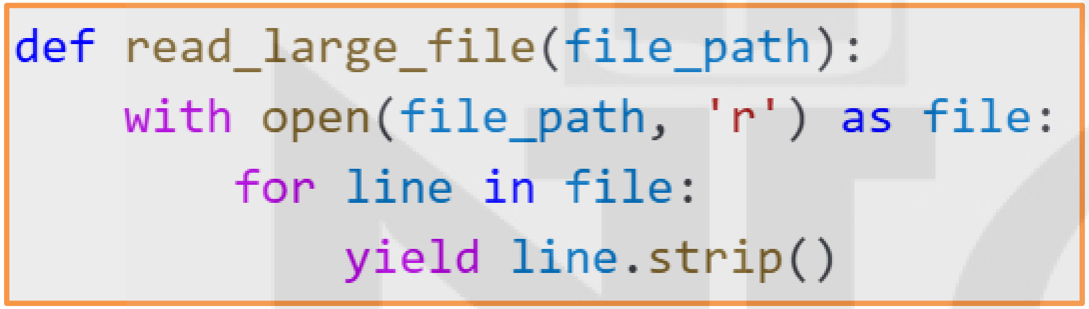
StopIteration

- See Notebook:
- StopIteration_in_Generators.ipynb

Advantages of Generators

- Saves memory
- Improves performance
- Suitable for large data processing
- Supports infinite sequences

Real-World Examples



```python
def read_large_file(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()
```

- Reading large files

Summary and Q&A

- Generators provide memory-efficient iteration
- Use `yield` instead of `return`
- Generator expressions offer a compact syntax
- Suitable for large data sets and real-time streaming
- Questions?