# Unit 6 – Iterators, Generators, and Comprehensions

Python provides powerful tools to **iterate over data efficiently**, and **generate new data structures** concisely.

## 6.1: Iterators and the Iterator Protocol (iter(), next())

## 1. Definition / Concept

- **Iterable** → An object that can return its members one at a time (e.g., list, tuple, string).
- **Iterator** → An object that represents a stream of data; returns one element at a time using next().
- **Iterator Protocol** → Requires two methods:
  - __iter__() → returns iterator object itself.
  - __next__() → returns next element, raises StopIteration when done.

📌 **iter(obj)** → returns an iterator.

📌 **next(iterator)** → fetches next element.

## 2. Analogy / Real-Life Connection

Think of a **music playlist**:

- The playlist (list of songs) is an **iterable**.
- The music player (pressing *next*) is the **iterator** that plays one song at a time.

## 3. Syntax

```
iterator = iter(iterable)
value = next(iterator)
```

Custom iterator:

```
class MyIterator:
  def __iter__(self):
    return self
  def __next__(self):
    # return next value or raise StopIteration
```

## 4. Step-by-Step Explanation

1. **Iterable** → any object implementing __iter__().
2. iter(iterable) returns iterator object.
3. next(iterator) gets the next element.
4. Once elements are exhausted → StopIteration.

## 5. Example Code

### (a) Basic Iterator with List

```
nums = [10, 20, 30]
it = iter(nums)

print(next(it))  # 10
print(next(it))  # 20
print(next(it))  # 30
# print(next(it))  # ❌ StopIteration
```

### (b) Iterating with Loop (Internally Uses Iterators)

```
for num in [1, 2, 3]:
    print(num)
```

Behind the scenes → Python calls iter() and next().

### (c) Custom Iterator Class

```
class Counter:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.end:
            raise StopIteration
        val = self.current
```

```
        self.current += 1
        return val

for num in Counter(1, 5):
    print(num)
```

**Output:**

```
1
2
3
4
5
```

### (d) Manual StopIteration Handling

```
nums = iter([1, 2])
try:
    print(next(nums))
    print(next(nums))
    print(next(nums))  # StopIteration
except StopIteration:
    print("No more elements")
```

**Output:**

```
1
2
No more elements
```

## 6. Diagram / Flow

📌 **Iterator Workflow**

```
Iterable → iter() → Iterator → next() → next element
                                     → StopIteration (end)
```

## 7. Output

- Shown in examples.
- Iterators allow step-by-step access to data.

## 8. Common Errors & Debugging

❌ **Error 1: Using next() on non-iterable**

next(10) # ❌ not iterable

✅ Fix: Convert to iterable (list, str, etc.).

❌ **Error 2: Forgetting StopIteration in custom iterator**

- Infinite loops if __next__ never raises StopIteration.

❌ **Error 3: Confusing iterables with iterators**

nums = [1, 2, 3] # iterable

it = iter(nums) # iterator

## 9. Interview / Industry Insight

- Interview Qs:
    - Difference between iterable and iterator?
    - What is the iterator protocol in Python?
    - How does a for loop work internally?
- Industry:
    - Iterators are key in **data streaming, file reading, DB cursors**.
    - Used in **lazy evaluation** → avoid loading entire dataset into memory.
    - Example: Iterating millions of rows in Pandas/Numpy without memory overload.

# 6.2: Creating Custom Iterators

## 1. Definition / Concept

- A **custom iterator** is a user-defined class that implements the **iterator protocol**.
- Iterator protocol requires:
    - __iter__(self) → returns the iterator object itself.
    - __next__(self) → returns the next item, raises StopIteration when done.
- Useful when you need **custom logic for iteration** (e.g., prime numbers, Fibonacci).

## 2. Analogy / Real-Life Connection

Think of a **ticket counter machine**:

- Every press of the button → gives the **next ticket number**.
- When tickets are finished → machine says **"No more tickets"** (StopIteration).

## 3. Syntax

```
class MyIterator:
    def __iter__(self):
        return self
    def __next__(self):
        # logic
        # raise StopIteration when finished
```

## 4. Step-by-Step Explanation

1. Define a class.
2. Implement __iter__() → usually returns self.
3. Implement __next__() → define logic for next value.
4. Raise StopIteration when sequence ends.
5. Use for loop or next() to iterate.

## 5. Example Code

### (a) Simple Counter Iterator

```
class Counter:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.end:
            raise StopIteration
        val = self.current
        self.current += 1
```

```
        return val

for num in Counter(1, 5):
    print(num)
```

**Output:**

```
1
2
3
4
5
```

### (b) Fibonacci Iterator

```
class Fibonacci:
    def __init__(self, n):
        self.n = n
        self.a, self.b = 0, 1
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.n:
            raise StopIteration
        val = self.a
        self.a, self.b = self.b, self.a + self.b
        self.count += 1
        return val

for num in Fibonacci(7):
    print(num, end=" ")
```

**Output:**

0 1 1 2 3 5 8

### (c) Prime Number Iterator

```
class Primes:
```

```
    def __init__(self, limit):
        self.limit = limit
        self.num = 2

    def __iter__(self):
        return self

    def __next__(self):
        while self.num <= self.limit:
            n = self.num
            self.num += 1
            for i in range(2, n):
                if n % i == 0:
                    break
            else:
                return n
        raise StopIteration

for p in Primes(10):
    print(p, end=" ")
```

**Output:**

2 3 5 7

### (d) Manual Iteration with next()

```
it = iter(Counter(1, 3))
print(next(it))
print(next(it))
print(next(it))
# next(it) → StopIteration
```

**Output:**

1

2

3

## 6. Diagram / Flow

📌 **Custom Iterator Lifecycle**

```
Object created → __iter__() → __next__() → value returned
                                           → StopIteration (end)
```

## 7. Output

- Iterators generate values one at a time.
- Stop automatically when sequence ends.

## 8. Common Errors & Debugging

❌ **Error 1: Forgetting StopIteration**

def __next__(self):

  return self.current  # ❌ infinite loop

✅ Fix: Raise StopIteration.

❌ **Error 2: Returning self in __next__**

- Must return **values**, not the iterator itself.

❌ **Error 3: Confusing __iter__ and __next__**

- __iter__() returns iterator object.
- __next__() generates values.

## 9. Interview / Industry Insight

- Interview Qs:
  - How do you create a custom iterator?
  - Difference between iterable and iterator?
  - Write a custom iterator for Fibonacci.
- Industry:
  - Custom iterators used in **streaming large data, DB cursors, network streams**.
  - Example: Pandas internally uses iterators to handle rows.
  - More efficient than loading everything into memory.

## 6.3: Generators and yield keyword

 

# 1. Definition / Concept

- A **generator** is a special type of iterator created with **functions** that use the yield keyword.
- Unlike normal functions (which return once), a generator function **pauses at yield** and **resumes later**.
- Automatically implements the iterator protocol → no need to write __iter__ and __next__.

📌 Key difference:

- return → ends function.
- yield → pauses function and remembers state.

# 2. Analogy / Real-Life Connection

Think of **watching a TV series**:

- Each **episode** is like a yield.
- You **pause at the end of one episode** and **resume with the next** later.
- Unlike a movie (normal function) which runs fully at once.

# 3. Syntax

```
def my_generator():
  yield value1
  yield value2
```

Use:

```
gen = my_generator()
next(gen)
```

# 4. Step-by-Step Explanation

1. Define a generator function with yield.
2. Calling it returns a **generator object** (not values yet).
3. Use next() or loop to fetch values one at a time.
4. When generator finishes → raises StopIteration.

# 5. Example Code

## (a) Simple Generator

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

gen = my_gen()
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
```

### (b) Using Generator in Loop

```
for val in my_gen():
    print(val)
```

**Output:**

```
1
2
3
```

### (c) Generator with State

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for num in countdown(5):
    print(num)
```

**Output:**

```
5
4
3
2
1
```

### (d) Fibonacci Generator

```
def fib(n):
```

```
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

for num in fib(7):
    print(num, end=" ")
```

**Output:**

0 1 1 2 3 5 8

### (e) return in Generators

```
def simple_gen():
    yield 1
    yield 2
    return "Done"

gen = simple_gen()
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # raises StopIteration with value "Done"
```

## 6. Diagram / Flow

📌 **Generator Execution**

Call generator → returns generator object

next() → runs till yield → returns value → pauses

next() → resumes → yield again → pauses

...

StopIteration when finished

## 7. Output

- Values produced one at a time (lazy evaluation).
- More memory-efficient than lists.

## 8. Common Errors & Debugging

❌ **Error 1: Using return instead of yield**

- Returns value once, not a sequence.

❌ **Error 2: Expecting generator to execute immediately**

```
def gen():
  yield 1
print(gen())  # ❌ generator object, not value
```

✅ Fix: Use next(gen()) or loop.

❌ **Error 3: Forgetting StopIteration**

- Normal, don't panic → just means generator is finished.

## 9. Interview / Industry Insight

- Interview Qs:
  - Difference between yield and return.
  - Why use generators over lists?
  - Write a generator for Fibonacci.
- Industry:
  - Generators widely used in **streaming APIs, data pipelines, log processing**.
  - Efficient for **big data** since they don't load everything into memory.
  - Example: Reading a large CSV file line by line using generator.

## 6.4: Generator Expressions

## 1. Definition / Concept

- A **generator expression** is a concise way to create a generator, similar to list comprehensions but using **parentheses ()** instead of square brackets [].
- It produces values **lazily** (one at a time), instead of storing the entire list in memory.

📌 Key difference:

- **List comprehension** → [x*x for x in range(5)] → stores all results.
- **Generator expression** → (x*x for x in range(5)) → generates on demand.

## 2. Analogy / Real-Life Connection

- **List comprehension** → Like buying **all groceries at once** and storing them at home (uses more space).
- **Generator expression** → Like **ordering groceries on-demand** when you need them (saves space).

## 3. Syntax

gen = (expression for item in iterable if condition)

## 4. Step-by-Step Explanation

1. Use parentheses () instead of [].
2. Returns a **generator object**.
3. Fetch values using next() or a loop.
4. More memory-efficient than list comprehensions.

## 5. Example Code

### (a) Basic Generator Expression

```
gen = (x*x for x in range(5))
print(gen)       # generator object
print(next(gen))  # 0
print(next(gen))  # 1
print(next(gen))  # 4
```

### (b) Using Generator Expression in Loop

```
for val in (x*x for x in range(5)):
   print(val)
```

**Output:**

```
0
1
4
9
16
```

### (c) With Condition

```
gen = (x for x in range(10) if x % 2 == 0)
for val in gen:
   print(val, end=" ")
```

**Output:**

0 2 4 6 8

### (d) Memory Efficiency Demo

```
import sys

lst = [x*x for x in range(1000)]
gen = (x*x for x in range(1000))

print("List size:", sys.getsizeof(lst))
print("Generator size:", sys.getsizeof(gen))
```

**Output (example):**

List size: 8856

Generator size: 112

### (e) Using Generator Expression in Functions

```
nums = (x for x in range(1, 6))
print(sum(nums))   # works directly
```

**Output:**

15

## 6. Diagram / Flow

📌 **List vs Generator Expression**

List comprehension:

[x*x for x in range(5)]

→ Creates full list [0, 1, 4, 9, 16]

Generator expression:

(x*x for x in range(5))

→ Creates generator object → produces one value at a time

## 7. Output

- Generators save memory.
- Useful for large datasets or infinite streams.

## 8. Common Errors & Debugging

❌ **Error 1: Forgetting parentheses**

gen = x*x for x in range(5)  # ❌ SyntaxError

✅ Fix: Use (x*x for x in range(5)).

❌ **Error 2: Reusing exhausted generator**

gen = (x for x in range(3))

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

print(list(gen))  # [] (already exhausted)

✅ Fix: Recreate generator if needed again.

❌ **Error 3: Expecting list methods on generator**

gen = (x*x for x in range(5))

print(len(gen))  # ❌ TypeError

✅ Fix: Convert to list → len(list(gen)).

## 9. Interview / Industry Insight

- Interview Qs:
  - Difference between list comprehension and generator expression?
  - Why are generator expressions memory-efficient?
  - Can you use generator expressions directly in functions like sum() or max()?
- Industry:
  - Used in **data pipelines, big data, streaming logs**.
  - Very common in **data science** to avoid memory overload.
  - Example: (line for line in open("huge_file.txt")) → processes file lazily.

## 6.5: List, Set, and Dict Comprehensions

# 1. Definition / Concept

- **Comprehensions** are concise ways to create **lists, sets, or dictionaries** from iterables in a single line.
- Types:
    - **List Comprehension** → []
    - **Set Comprehension** → {}
    - **Dict Comprehension** → {key: value}

📌 Key benefit → shorter, more readable, often faster.

# 2. Analogy / Real-Life Connection

Think of a **factory assembly line**:

- Raw materials (iterables) enter.
- A **processing rule** (expression) is applied.
- The factory outputs **products** (lists, sets, dicts) in one step.

# 3. Syntax

```
# List comprehension
[expression for item in iterable if condition]

# Set comprehension
{expression for item in iterable if condition}

# Dict comprehension
{key: value for item in iterable if condition}
```

# 4. Step-by-Step Explanation

1. Start with an iterable.
2. Apply an **expression** to each element.
3. Optionally filter with if condition.
4. Collect into list/set/dict.

# 5. Example Code

## (a) List Comprehension

```
squares = [x*x for x in range(5)]
```

```
print(squares)
```

**Output:**

```
[0, 1, 4, 9, 16]
```

### (b) List Comprehension with Condition

```
evens = [x for x in range(10) if x % 2 == 0]
print(evens)
```

**Output:**

```
[0, 2, 4, 6, 8]
```

### (c) Nested List Comprehension

```
matrix = [[i*j for j in range(1, 4)] for i in range(1, 4)]
print(matrix)
```

**Output:**

```
[[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

### (d) Set Comprehension (removes duplicates)

```
nums = [1, 2, 2, 3, 4, 4]
unique = {x for x in nums}
print(unique)
```

**Output:**

```
{1, 2, 3, 4}
```

### (e) Dict Comprehension

```
nums = [1, 2, 3, 4]
squares = {x: x*x for x in nums}
print(squares)
```

**Output:**

```
{1: 1, 2: 4, 3: 9, 4: 16}
```

### (f) Dict Comprehension with Condition

```
words = ["apple", "banana", "cherry"]
lengths = {w: len(w) for w in words if len(w) > 5}
print(lengths)
```

**Output:**

```
{'banana': 6, 'cherry': 6}
```

## 6. Diagram / Flow

📌 **List Comprehension Workflow**

for x in iterable:
    if condition:
        expression → output list

📌 Example:

[x*x for x in range(5)]
→ [0, 1, 4, 9, 16]

## 7. Output

- List → maintains order, allows duplicates.
- Set → removes duplicates.
- Dict → key-value pairs.

## 8. Common Errors & Debugging

❌ **Error 1: Forgetting brackets**

x*x for x in range(5)  # ❌ SyntaxError

✅ Fix: Use [], {}, or {k:v}.

❌ **Error 2: Nested comprehension confusion**

- Overuse can reduce readability.

  ✅ Fix: Break into normal loops if logic is complex.

❌ **Error 3: Assuming set comprehension preserves order**

- Sets are unordered → order is not guaranteed.

## 9. Interview / Industry Insight

- Interview Qs:
  - Difference between list comprehension and generator expression?
  - How do set comprehensions handle duplicates?

- Write a dict comprehension for word length mapping.

- Industry:
  - Widely used in **data preprocessing, filtering, transformations**.
  - Preferred in **data science pipelines** for clean and concise code.
  - Example: Extracting columns, filtering records, converting data formats.

## 6.6: Performance Benefits of Generators vs Lists

## 1. Definition / Concept

- **Lists** → Store all elements in memory at once.
- **Generators** → Produce values **on demand (lazy evaluation)**.
- Generators are **more memory-efficient** and sometimes **faster** for large datasets.

📌 **Lists** → Good for random access, repeated use.

📌 **Generators** → Best for one-time, sequential access of large/streaming data.

## 2. Analogy / Real-Life Connection

- **List** → Buying groceries for the entire month at once (needs more storage space, some may go unused).
- **Generator** → Ordering groceries **only when you need them** (saves space, avoids waste).

## 3. Syntax

```
# List comprehension
lst = [x*x for x in range(1000000)]

# Generator expression
gen = (x*x for x in range(1000000))
```

## 4. Step-by-Step Explanation

1. Lists → allocate memory for all elements immediately.
2. Generators → don't compute values until requested.
3. Lists → allow indexing & random access.
4. Generators → can only be iterated once.

## 5. Example Code

### (a) Memory Comparison

```
import sys

lst = [x*x for x in range(1000000)]
gen = (x*x for x in range(1000000))

print("List size:", sys.getsizeof(lst))
print("Generator size:", sys.getsizeof(gen))
```

**Output (example):**

List size: 8697464

Generator size: 112

### (b) Speed Demo

```
import time

# List comprehension
start = time.time()
sum([x*x for x in range(1000000)])
end = time.time()
print("List time:", end - start)

# Generator expression
start = time.time()
sum((x*x for x in range(1000000)))
end = time.time()
print("Generator time:", end - start)
```

**Output (example):**

List time: 0.45 sec

Generator time: 0.40 sec

### (c) Generator for Streaming Data

```
def stream_numbers(n):
    for i in range(1, n+1):
        yield i
```

```
for num in stream_numbers(5):
    print(num)
```

**Output:**

1

2

3

4

5

## 6. Diagram / Flow

📌 **List vs Generator**

List:
[1, 2, 3, 4, 5]
Stores all values in memory.

Generator:
(x for x in range(1,6))
Produces values one at a time.

## 7. Output

- Generators → small memory footprint.
- Lists → higher memory usage, but allow direct indexing.

## 8. Common Errors & Debugging

❌ **Error 1: Expecting random access from generator**

```
gen = (x for x in range(5))
print(gen[2])  # ❌ TypeError
```

✅ Fix: Convert to list if indexing is needed.

❌ **Error 2: Reusing generator after exhaustion**

```
gen = (x for x in range(3))
```

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

print(list(gen))  # [] → already consumed

✅ Fix: Recreate generator if needed again.

❌ **Error 3: Assuming generators are always faster**

- Generators save memory, but lists can be faster for small datasets since values are precomputed.

## 9. Interview / Industry Insight

- Interview Qs:
    - Why are generators more memory-efficient than lists?
    - Can you index a generator? Why not?
    - When would you prefer a list over a generator?
- Industry:
    - **Generators** → used in big data, logs, streaming APIs, infinite data sources.
    - **Lists** → used when data needs to be accessed multiple times.
    - Example: ETL pipelines → generators stream millions of rows without RAM overload.

## 6.7: Iteration using NumPy Arrays

## 1. Definition / Concept

- **NumPy arrays** are more efficient than Python lists for numerical computations.
- Iteration works differently in NumPy:
    - Simple iteration → loops through rows (for multi-D arrays).
    - **nditer** → advanced iterator for efficient element-wise iteration.
    - **Vectorized operations** → often eliminate the need for explicit loops.

📌 Advantage: NumPy iterations are faster & memory-efficient compared to native Python lists.

## 2. Analogy / Real-Life Connection

- **Python list iteration** → like writing marks of each student manually one by one.
- **NumPy iteration** → like scanning the entire marks sheet with a computer → faster and structured.

## 3. Syntax

```
import numpy as np

arr = np.array([...])

# Simple iteration
for x in arr: ...

# Using nditer
for x in np.nditer(arr): ...
```

## 4. Step-by-Step Explanation

1. 1D NumPy arrays → behave like lists.
2. 2D+ arrays → simple for loops iterate row by row.
3. Use **np.nditer()** for element-wise iteration across dimensions.
4. Can also iterate with conditions, transformations, or in flat order.

## 5. Example Code

### (a) Iterating 1D Array

```
import numpy as np

arr = np.array([10, 20, 30])
for x in arr:
    print(x)
```

**Output:**

```
10
20
30
```

### (b) Iterating 2D Array (Row by Row)

```
arr = np.array([[1, 2], [3, 4], [5, 6]])
for row in arr:
    print(row)
```

**Output:**

[1 2]

[3 4]

[5 6]

### (c) Element-wise Iteration with nditer

```
arr = np.array([[1, 2], [3, 4]])
for x in np.nditer(arr):
    print(x, end=" ")
```

**Output:**

1 2 3 4

### (d) Iterating with Index (ndenumerate)

```
arr = np.array([[10, 20], [30, 40]])
for idx, val in np.ndenumerate(arr):
    print(idx, val)
```

**Output:**

(0, 0) 10

(0, 1) 20

(1, 0) 30

(1, 1) 40

### (e) Vectorized Alternative (Faster)

```
arr = np.array([1, 2, 3, 4])
print(arr * 2)  # no loop needed
```

**Output:**

[2 4 6 8]

## 6. Diagram / Flow

📌 **Iteration in NumPy**

```
1D Array:  [1, 2, 3]   → element by element
2D Array:  [[1,2],[3,4]]
    ├ Row iteration → [1,2], [3,4]
    ├ nditer → 1,2,3,4
    └ ndenumerate → ((i,j), value)
```

## 7. Output

- for row in arr → row-wise iteration.
- np.nditer(arr) → element-wise.
- Vectorized ops → avoid loops altogether.

## 8. Common Errors & Debugging

❌ **Error 1: Expecting element-wise iteration with simple loop on 2D array**

```
for x in np.array([[1,2],[3,4]]):
    print(x)  # ❌ prints row, not element
```

✅ Fix: Use nditer.

❌ **Error 2: Forgetting vectorized alternatives**

- Loops in NumPy are slower than vectorized operations.

✅ Fix: Prefer broadcasting (arr * 2, arr + 5).

❌ **Error 3: Confusing nditer with iter**

- iter() is for Python objects, nditer() is for NumPy arrays.

## 9. Interview / Industry Insight

- Interview Qs:
  - How do you iterate over elements in a 2D NumPy array?
  - What's the difference between simple iteration and nditer?
  - Why are vectorized operations preferred over iteration in NumPy?
- Industry:
  - Iteration used in **numerical simulations, ML preprocessing, image processing**.
  - But industry prefers **vectorized solutions** for performance.
  - Example: Instead of looping pixel-by-pixel in an image, use vectorized NumPy operations.