
DECODE
AI

Iterator & Generator

→ **Eager Evaluation** ← default behaviour

① A programming concept where expressions are evaluated immediately as they are encountered.

② There is no delay in computing the result

③ Example: List Comprehension

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

↳ All values [0, 1, 4, 9, 16] are calculated and stored immediately.
↳ Memory is used up-front.

* **Pros:**
  ↳ Faster access to result when needed immediately.
  ↳ Useful when full result required right away
* **Cons:**
  ↳ Consume more memory.
  ↳ Can be wasteful if not all values are actually used

→ **Lazy Evaluation**

① Expressions are not evaluated until needed.

② values are calculated one by one, only when requested.

③ Saves memory for large data.

④ Can be implemented using iterator and generator.

⑤ Example : Generator Expression

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

↳ squares holds a generator object, which just knows how to produce values when asked.

↳ Execution happens only when you iterate.

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

→ Space Complexity Comparison:-

$n = 10^6$

① Eager evaluation

↳ int in Cpython takes ~28 B

↳ Total space = 28×10^6 B = 28 MB (for integer) + list overhead (~4-8 MB)

= ~30-35 MB

② Lazy evaluation

↳ only 1 value is in memory at a time during iteration

↳ Total space = 200-400 B

↳ for generator object and states

☑ Real world problem

→ Imagine you are working with a very large dataset – say a file with 10M lines.

→ Generate an Infinite sequence (like fibonacci no or prime number)

☑ Lazy evaluation is the Solution

→ Implemented using

① Iterator

② Generator

Iterator

→ An iterator is an object which implements two methods.

- ·) `__iter__()` ← returns the iterator object itself.
- ·) `__next__()` ← returns the next item or raises StopIterator exception.

→ `my_list = [1, 2, 3]`

NOTE :-
List, string → Iterable

```
it1 = iter(my_list)   <- iterator
it2 = iter(my_string)
```

```
next(it1) #1
next(it1) #2
next(it1) #3
next(it1) → Raise StopIteration.
```

NOTE

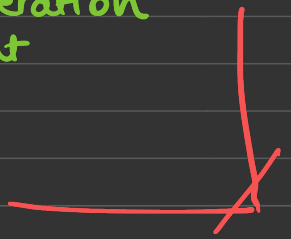
`iter(my_list) ≡ my_list.__iter__()`

`next(my_list) ≡ my_list.__next__()`

```
class CountUpTo:
    def __init__(self, max):
        self.max = max
        self.current = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.max:
            raise StopIteration
        value = self.current
        self.current += 1
        return value
```



Usage

```
counter = CounterUpTo(5)
for num in counter:
    print(num)      → # 0 1 2 3 4
```

→ The above code is too long.

→ writing Iterator manually is a Verbose.

Soln ↓

Generator

→ It helps us write clean readable code in few lines.

→ Special kind of iterator that is defined using a function with a Yield Keyword.

```
def count_up_to(max):
    current = 1
    while current <= max:
        yield current
        current += 1
```

⇒

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