```
"""
Iterable:
1. A Collection of elements that is capable of traversing.
2. Implements __iter__ method.
3. Ex: list, tuple, dict


Iterators:
1. Iterators are objects that are capable of returning
one element at a time.
When all the elements in an iterator are traversed,
they raise a StopIteration Exception.
2. Iterators can transform iterables to iterators.
3. They implement two methods.
   __iter__, __next__
   __iter__ => returns itself.
   __next__ => Logic for returning the next element.
   When it reaches the end, will raise a StopIteration Exception.


Iterable vs Iterator:
1. Object: An iterable is any object that can return an iterator,
          while an iterator is the actual object that performs
          iteration one element at a time.
2. Methods Implementation:
   Iterables implement the __iter__ method only whereas the
   iterators implement __iter__ and __next__ methods.
3. State: Iterables doesn't maintain any state
    whereas the iterators maintain the current state
4. Return Value: We have to manually use indices to
   retrieve the values from an iterable. Iterators are capable
   of returning one value at a time.
5. Relationship: All the iterators are iterables.
   But, All the iterables are not iterators.


Uses of Iterators:
1. Memory Efficient (Lazy Evaluation):
   In the case of large datasets, storing all the elements in
   memory is expensive. Instead, We use iterators to
   retrieve one value at a time.


Iterators in Python are fundamental for efficient and flexible data processing. Their primary uses include:

1. Sequential Traversal of Collections: Iterators allow you to access elements of iterable objects (like lists, tuples, dictionaries, sets, and strings) one at a time,
    my_list = [1, 2, 3, 4]
    my_iterator = iter(my_list)
    print(next(my_iterator))  # Output: 1
    print(next(my_iterator))  # Output: 2
2. Memory-Efficient Processing of Large Datasets: Iterators implement "lazy evaluation," meaning they generate and provide elements only when requested. This is crucia
3. Creating Infinite Sequences: Iterators can be designed to produce an endless stream of values, which is useful in scenarios where you need to continuously generate

Here are some key benefits:

Lazy Evaluation: Processes items only when needed, saving memory.
Generator Integration: Pairs well with generators and functional tools.
Stateful Traversal: Keeps track of where it left off.
Uniform Looping: Same for loop works for lists, strings and more.
Composable Logic: Easily build complex pipelines using tools like itertools.
"""
l = [1, 2, 3, 4]
print(l)
# print(next(l))

l_iterator = iter(l)
print(l_iterator)
print(next(l_iterator))
print(next(l_iterator))
print(next(l_iterator))
print(next(l_iterator))
# print(next(l_iterator))


print("Using For Loop...")
l_iterator = iter(l)
for item in l_iterator:
    print(item)


# Implementing a Custom Iterator
# Iterator of n natural numbers
print("Custom Iterator...")
class NNaturalNumbers:
    def __init__(self, max_element):
        self.cur_value = 0
        self.max_element = max_element

    def __iter__(self):
        return self

    def __next__(self):
        if self.cur_value < self.max_element:
            self.cur_value += 1
            return self.cur_value
        else:
            raise StopIteration


numbers_5 = NNaturalNumbers(5)
# print(next(numbers_5))
print(next(numbers_5))
print(next(numbers_5))
print(next(numbers_5))
for i in numbers_5:
    print(i, end=' ')
for i in numbers_5:
    print(i, end=' ')
print()
for i in NNaturalNumbers(5):
    print(i, end=' ')
print()


print("Iterable...")
l = [1, 2, 3, 4, 5]
for i in l:
    print(i, end=' ')

for i in l:
    print(i, end=' ')
print()
```

```python
print("Iterator....")
l = [1, 2, 3, 4, 5]
l_iterator = iter(l)
for i in l_iterator:
    print(i, end=' ')

for i in l_iterator:
    print(i, end=' ')
```