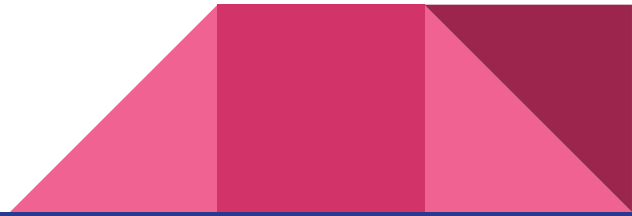
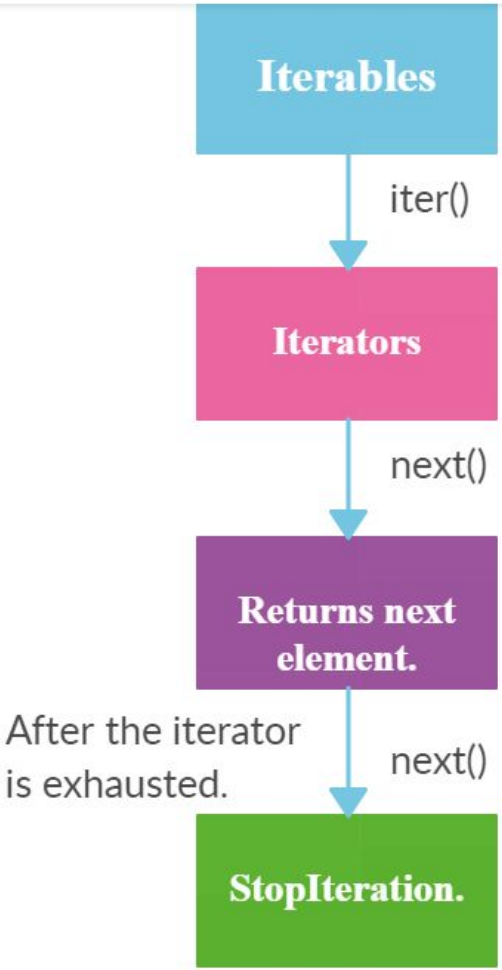
- Iterator
- Generator

Iterator in Python

n Python, an iterator is an object which implements the iterator protocol, which means it consists of the methods such as __iter__() and __next__(). An iterator is an iterable object with a state so it remembers where it is during iteration

- Iterable is an object which can be looped over or iterated over with the help of a for loop.
- Objects like lists, tuples, sets, dictionaries, strings, etc. are called iterables. In short and simpler terms, iterable is anything that you can loop over.
- In simpler words, iterable is a container that has data or values and we perform an iteration over it to get elements one by one. (Can traverse through all the given values one by one)
- Iterable has an in-built dunder method __*iter*__.




Iterables
iter()
Iterators
next()
Returns next element.
After the iterator is exhausted.
next()
StopIteration.

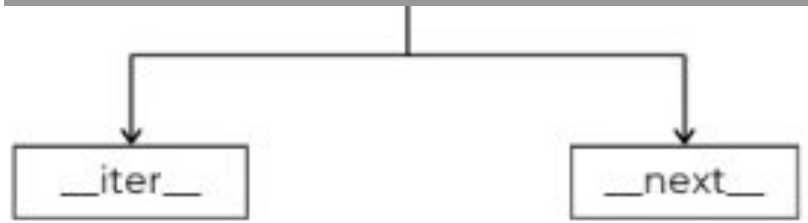
- **In Python, an iterator is an object which implements the iterator protocol, which means it consists of the methods such as *__iter__()* and *__next__()*.**

- **An iterator is an iterable object with a state so it remembers where it is during iteration. For Example, Generator**

- **These iterators give or return the data one element at a time.**

- **It performs the iteration to access the elements of the iterable one by one. As it maintains the internal state of elements, the iterator knows how to get the next value.**

Iterator

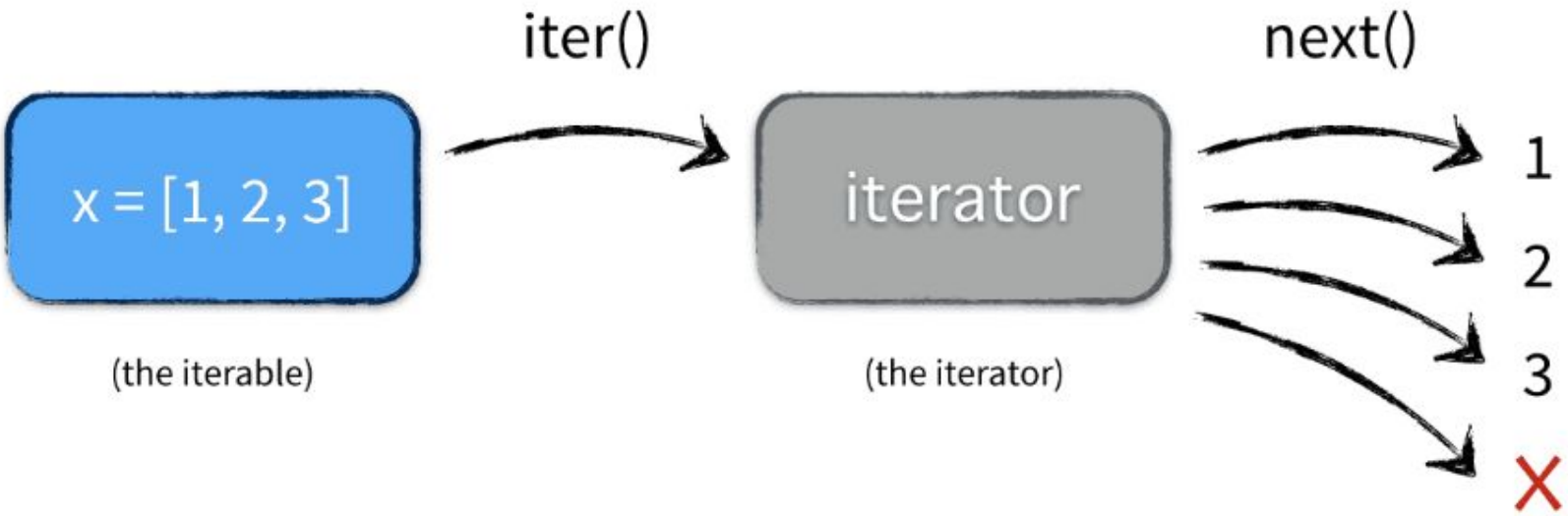


iter() function →

iter() function is used to iterate through sequence
Belongs to iterator method
used on iterable data-type

next() function →

next() function is used to reach the next element of sequence
Belongs to iterator method
used on iterable data-type


iter()
next()
x = [1, 2, 3]
iterator
1
2
3
(the iterable)
(the iterator)

Iterables	Iterators
1.Can be iterated using for loop	Can be iterated using for loop
2. Iterables supports iter() function.	Iterators supports iter() and next() function.
3. Iterables are not iterators.	Iterators are also iterables.

Now, let's see the limits of the iterators:

- **We can only go forward in an iterator.**
- **We can't make a copy of it.**
- **No way to get the previous element.**
- **We can't reset the iterator.**

Input

```
number_iterator = iter([1, 2, 3, 4, 5])
print(type(number_iterator))
print(next(number_iterator))
print(next(number_iterator))
print(next(number_iterator))
print(next(number_iterator))
print(next(number_iterator))
# Once the iterator is exhausted, next() function raise StopIteration.
print(next(number_iterator))
```

Output:

```
<class 'list_iterator'>
1
2
3
4
5

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-5-32f956cadd50> in <module>
      6 print(next(number_iterator))
      7 print(next(number_iterator))
----> 8 print(next(number_iterator))

StopIteration:
```

- Python provides a generator to create your own iterator function.
- A generator is a special type of function which does not return a single value, instead, it returns an iterator object with a sequence of values.
- In a generator function, a `yield` statement is used rather than a return statement.

Example: Generator Function

```
def mygenerator():
    print('First item')
    yield 10

    print('Second item')
    yield 20

    print('Last item')
    yield 30
```

In the above example, the `mygenerator()` function is a generator function. It uses `yield` instead of return keyword. So, this will return the value against the `yield` keyword each time it is called. However, you need to create an iterator for this function, as shown below.

Example: return in Generator Function

```
def mygenerator():
    print('First item')
    yield 10

    return

    print('Second item')
    yield 20

    print('Last item')
    yield 30
```

Example: Generator Function

Copy

```
>>> gen = mygenerator()
>>> next(gen)
First item
10
>>> next(gen)
Traceback (most recent call last):
        File "<pyshell#13>", line 1, in <module>

                it.__next__()
                StopIteration
```

As you can see, the above generator stops executing after getting the first item because the return keyword is used after `yield`ing the first item.