Dhirubhai Ambani University

(Formerly known as DA-IICT)

Topic: Iterators, Generators, Closures, and Decorators

Course: Programming Lab

Course Code- PC503

Dr. Ankit Vijayvargiya

Assistant Professor

Room No. 4205, Faculty Block 4

Email: ankit_Vijayvargiya[at]dau.ac.in

Phone: 079-68261628(O), 7877709590(M)

Dhirubhai Ambani University
Formerly known as DA-IICT

Iterators Vs Iterable

Iterable

Any object that can be looped over.

OR

In other words, an iterable is an object that can return its elements **one by one**.

List	**Tuple**	**String**	**Dictionary**	**Sets**

Dhirubhai Ambani University
Formerly known as
DA-IICT

Iterators

- An iterator is an object that allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation.

- An iterator is an object that implements the iterator protocol.

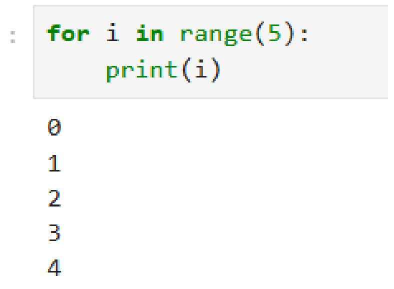
- The iterator protocol consists of two methods:
  - __iter__() method - return the iterator object
  - __next__() method - return the next element of the sequence, if no more elements, raise **stopIteration**

- Produce item one at a time.

Dhirubhai Ambani University
Formerly known as DA-IICT

Iterators

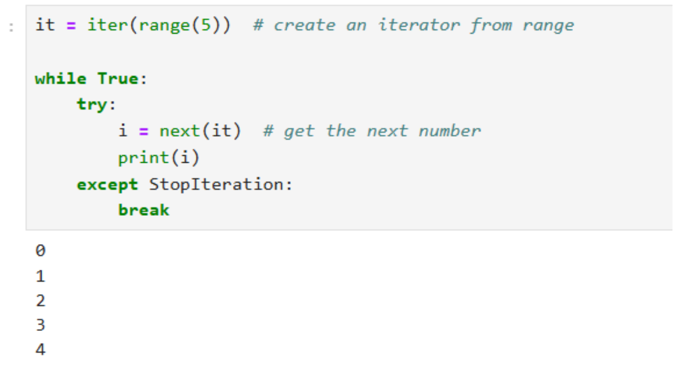
- **Built-in iterator**
  - A built-in iterator in Python is an iterator that is provided by Python itself through its built-in functions or types, so you don’t have to define your own __iter__() and __next__() methods.
  - Python provides several built-in iterators through functions or types, such as:
    - range() → produces numbers one at a time
    - enumerate() → returns index and value pairs
    - map() → applies a function to each element of an iterable
    - reverse() → reverse the elements



```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

- Calls iter(range(5)) → creates an iterator from the range object.
- Repeatedly calls next() on that iterator to get the next number.
- Stops automatically when the iterator is exhausted (i.e., StopIteration is raised).



```
it = iter(range(5))  # create an iterator from range

while True:
    try:
        i = next(it)  # get the next number
        print(i)
    except StopIteration:
        break
```

```
0
1
2
3
4
```

Iterators

- **Custom iterator**

Step 1:

- A custom iterator is usually defined with a class.
- __init__ initializes the iterator with starting and ending points.

Step 2:

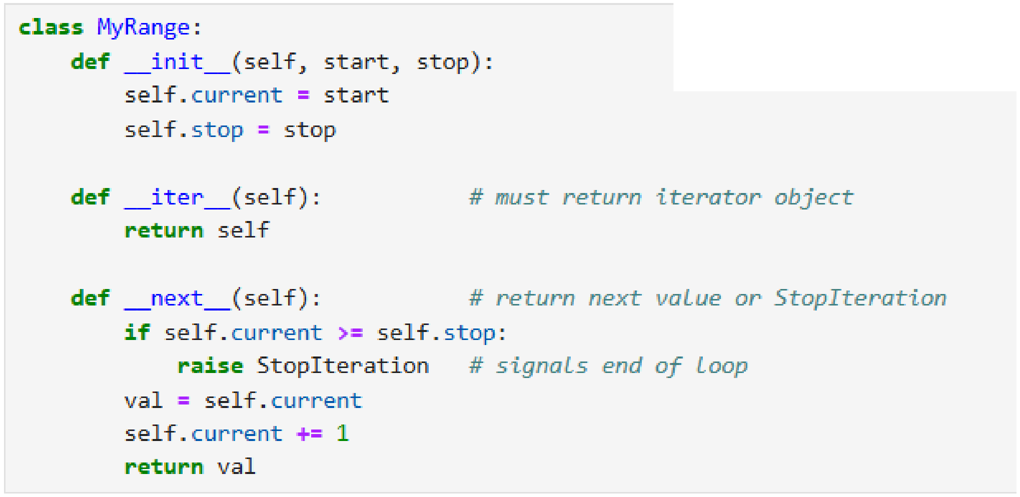
- Define __iter__ method

Step 3:

- Define __next__ method
- This method returns the next value each time it's called.

Step 4:

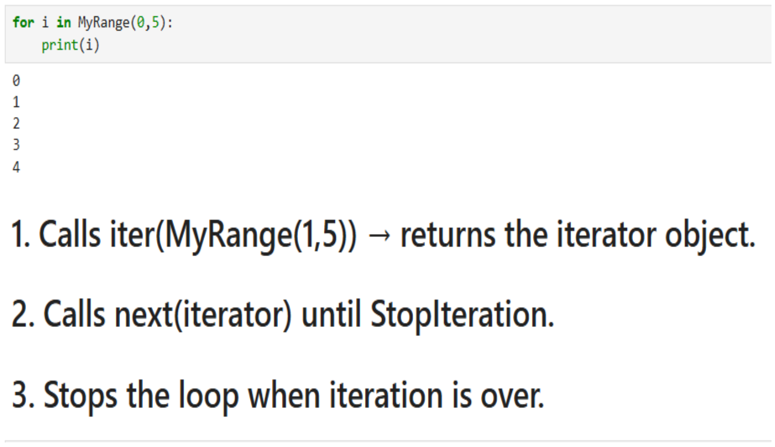
- Use the iterator



```python
class MyRange:
    def __init__(self, start, stop):
        self.current = start
        self.stop = stop

    def __iter__(self):          # must return iterator object
        return self

    def __next__(self):          # return next value or StopIteration
        if self.current >= self.stop:
            raise StopIteration   # signals end of loop
        val = self.current
        self.current += 1
        return val
```



```python
for i in MyRange(0,5):
    print(i)
```

```
0
1
2
3
4
```

1. Calls iter(MyRange(1,5)) → returns the iterator object.

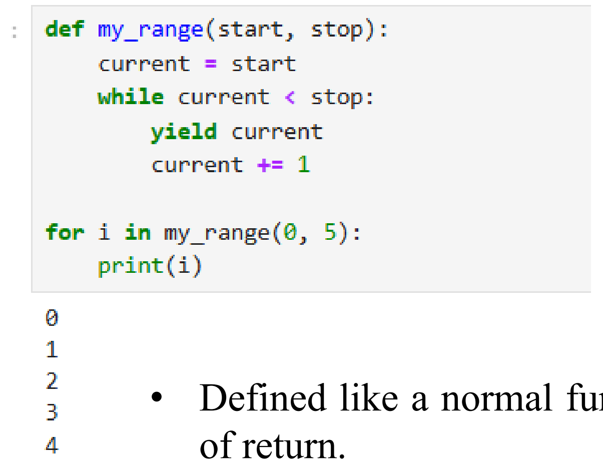
2. Calls next(iterator) until StopIteration.

3. Stops the loop when iteration is over.

Generators

- A generator is just a way to create an iterator without writing the full __iter__ and __next__ methods.

Generator Function



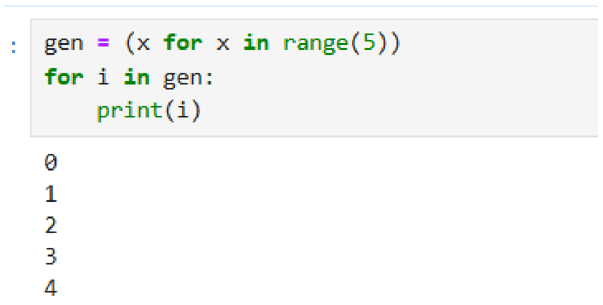
```
def my_range(start, stop):
    current = start
    while current < stop:
        yield current
        current += 1

for i in my_range(0, 5):
    print(i)
```

```
0
1
2
3
4
```

- Defined like a normal function, but uses yield instead of return.
- Every time Python sees yield, it pauses and remembers the function’s state.
- When the next value is needed, it resumes from where it left off.
- Automatically creates an iterator object behind the scenes.

Generator Expression



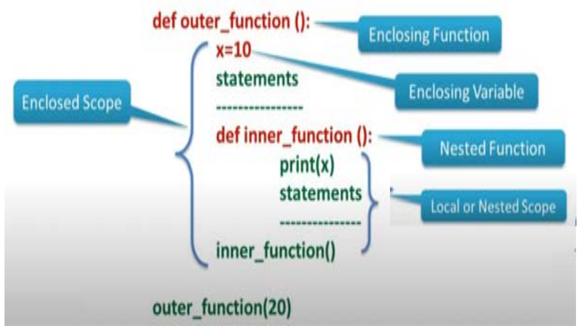
```
gen = (x for x in range(5))
for i in gen:
    print(i)
```

```
0
1
2
3
4
```

- Looks like a list comprehension but with parentheses ().
- Also returns a generator object.
- No function definition required here

Closures and Decorators

- Before starting the closures and Decorators, we should know about:
  - Functions
  - Nested Function
  - Name space
  - Function as a parameter
  - Function as an Object



```
def outer_function ():
    x=10
    statements
    ---------------
    def inner_function ():
        print(x)
        statements
        --------------
    inner_function()

outer_function(20)
```

Closures:

A closure happens when:

1. An inner function is defined inside an outer function.
2. The inner function remembers variables from the outer function's scope, even after the outer function has finished executing.
3. The inner function itself is returned (or passed around) without being executed immediately.

Importance of Closures:

- Data Hiding (Encapsulation without class)
- Maintaining state across function calls
  - Normally, function variables disappear after the function ends. Closures remember values even after the outer function has finished.
- Avoiding global variables
  - Instead of using messy global variables, closures let you store data locally inside a function scope.
- Used in decorators
  - Closures are the backbone of decorators.

Decorators:

A decorator in Python is a function that takes another function as input, adds extra functionality to it, and returns a new function.