Generators in Python:

In this tutorial, you'll learn how to create iterations easily using Python generators, how it is different from iterators and normal functions, and why you should use it.

Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.

There are two terms involved when we discuss generators.

1. Generator-Function :

A generator-function is defined like a normal function, but whenever it needs to generate a value, it does with the **yield** keyword rather than **return** keyword.

If the body of a **def contains yield** , the function automatically becomes a generator function.

Syntax:

```
def generator_name(arg):
    # statements
    yield something
```

For example:

```
# A generator function that yields 1 for first time, 2 second time and 3 third time.
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
```

2. Generator-Object :

Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or use the generator object in a “for in” loop (as shown in the above program).

Q. A Python program to demonstrate use of generator object with next() ?

```
# A generator function
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

# Creating generator object and assigns into  x  refference
x = simpleGeneratorFun()
# Iterating over the generator object using next
print(x.__next__())
print(x.__next__())
print(x.__next__())
# print(x.__next__()) #  returns StopIteration  exception
```

Output:

```
1
2
3
```

Note: If we are executing the one more time x.__next__() on object to get one more iteration value then it returns one exception like **StopIteration**.

Here , our object returns only 3 times values because of yield keyword and next no value is available.

So a generator function returns an generator object that is iterable, i.e., can be used as an Iterators.

Q. Write a generator program for displaying "Fibonacci" Numbers ?

```
# A simple generator for Fibonacci Numbers
def fibonacci(limit):
    # Initialize first two Fibonacci Numbers
    a , b  =  0 ,  1
    # One by one yield next Fibonacci Number
    while a  <  limit:
        yield a
        a , b = b , a + b
```

```
# Create a generator object
x = fibonacci(5)
# Iterating over the generator object using next
print(x.__next__())
print(x.__next__())
print(x.__next__())
print(x.__next__())
print(x.__next__())
```

Output:

```
0
1
1
2
3
```

```
# Iterating over the generator object using "for" loop.
print("\nUsing for loop")
for i in fibonacci(5):
    print(i)
```

Use of Python Generators

There are several reasons that make generators a powerful implementation.

1. Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of **2** using an iterator class.

```
class PowTwo:
    def __init__(self, max=0):
        self.n = 0
        self.max = max

    def __iter__(self):
```

```python
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result

for num  in PowTwo(5):
      print(num , end=” ”)
```

Output:

```
 1 2 4 8 16 32
```

The above program was lengthy and confusing. Now, let's do the same using a generator function.

```python
def PowTwoGen(max=0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1

for num  in PowTwoGen(5):
      print(num , end=” ”)
```

Output:

```
 1 2 4 8 16 32
```

Since generators keep track of details automatically, the implementation was concise and much cleaner.

2. Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill, if the number of items in the sequence is very large.

Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

3. Represent Infinite Stream

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

The following generator function can generate all the even numbers (at least in theory).

```python
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

4. Pipelining Generators

Multiple generators can be used to pipeline a series of operations. This is best illustrated using an example.

Suppose we have a generator that produces the numbers in the Fibonacci series. And we have another generator for squaring numbers.

If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

```python
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def square(nums):
    for num in nums:
        yield num**2
```

```
print(sum(square(fibonacci_numbers(10))))

# Output: 4895
```
Run C

This pipelining is efficient and easy to read (and yes, a lot cooler!).

What is difference between return and yield keywords ?

The difference between yield and return is that yield returns a value and pauses the execution while maintaining the internal states, whereas the return statement returns a value and terminates the execution of the function.

The following generator function includes the return keyword.

Example: return in Generator Function

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

    return

    print('Second item')
    yield 20

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

Now, execute the above function as shown below.

```
gen = mygenerator()
val = next(gen) #First item
print(val) #10

val = next(gen) #error
```

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.

Using for Loop with Generator Function

The generator function can also use the for loop.

Example: Use For Loop with Generator Function

```
def get_sequence_upto(x):
    for i in range(x):
        yield i
```

As you can see above, the get_sequence_upto function uses the yield keyword. The generator is called just like a normal function. However, its execution is paused on encountering the yield keyword. This sends the first value of the iterator stream to the calling environment. However, local variables and their states are saved internally.

The above generator function get_sequence_upto() can be called as below.

Example: Calling Generator Function

```
seq = get_sequence_upto(5)
print(next(seq)) #0
print(next(seq)) #1
print(next(seq)) #2
print(next(seq)) #3
print(next(seq)) #4
print(next(seq)) #error
```

The function resumes when next() is issued to the iterator object. The function finally terminates when next() encounters the StopIteration error.

In the following example, function square_of_sequence() acts as a generator. It yields the square of a number successively on every call of next().

Example: Generator Function with For Loop

```
gen=square_of_sequence(5)
while True:
    try:
        print ("Received on next(): ", next(gen))
    except StopIteration:
        break
```

The above script uses the try..except block to handle the StopIteration error. It will break the while loop once it catches the StopIteration error.

Output

```
Received on next(): 0
Received on next(): 1
Received on next(): 4
Received on next(): 9
Received on next(): 16
```

We can use the for loop to traverse the elements over the generator. In this case, the next() function is called implicitly and the StopIteration is also automatically taken care of.

Example: Generator with For Loop

```
squres = square_of_sequence(5)
for sqr in squres:
    print(sqr)
```

Output

```
0
1
4
9
16
```

Note:

One of the advantages of the generator over the iterator is that elements are generated dynamically. Since the next item is generated only after the first is consumed, it is more memory efficient than the iterator.