# Python Generators

In Python, a generator is a function that returns an iterator that produces a sequence of values when iterated over.

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.

---

## Create Python Generator

In Python, similar to defining a normal function, we can define a generator function using the `def` keyword, but instead of the `return` statement we use the `yield` statement.

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

Here, the `yield` keyword is used to produce a value from the generator.

When the generator function is called, it does not execute the function body immediately. Instead, it returns a generator object that can be iterated over to produce the values.

---

## Example: Python Generator

Here's an example of a generator function that produces a sequence of numbers,

```
def my_generator(n):

    # initialize counter
    value = 0

    # loop until counter is less than n
    while value < n:

        # produce the current value of the counter
        yield value

        # increment the counter
        value += 1

# iterate over the generator object produced by my_generator
for value in my_generator(3):

    # print each value produced by generator
    print(value)
```

**Output**

```
0
1
2
```

In the above example, the `my_generator()` generator function takes an integer `n` as an argument and produces a sequence of numbers from **0** to `n-1` using while loop.

The `yield` keyword is used to produce a value from the generator and pause the generator function's execution until the next value is requested.

The `for` loop iterates over the generator object produced by `my_generator()`, and the print statement prints each value produced by the generator.

We can also create a generator object from the generator function by calling the function like we would any other function as,

```
generator = my_range(3)
print(next(generator))  # 0
print(next(generator))  # 1
print(next(generator))  # 2
```

**Note**: To learn more, visit range() and for loop().

---

# Python Generator Expression

In Python, a generator expression is a concise way to create a generator object.

It is similar to a list comprehension, but instead of creating a list, it creates a generator object that can be iterated over to produce the values in the generator.

## Generator Expression Syntax

A generator expression has the following syntax,

```
(expression for item in iterable)
```

Here, `expression` is a value that will be returned for each item in the `iterable`.

The generator expression creates a generator object that produces the values of `expression` for each item in the `iterable`, one at a time, when iterated over.

---

# Example 2: Python Generator Expression

```
# create the generator object
squares_generator = (i * i for i in range(5))

# iterate over the generator and print the values
for i in squares_generator:
    print(i)
```

**Output**

```
0
1
4
9
16
```

Here, we have created the generator object that will produce the squares of the numbers **0** through **4** when iterated over.

And then, to iterate over the generator and get the values, we have used the `for` loop.

---

# 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):
        return self

    def __next__(self):
```

```
        if self.n > self.max:
            raise StopIteration

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

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

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

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).

```
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.

```
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
```

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

## Table of Contents