Working With Python Generator Object

- Syntax is like function but not functions
- Have a atleat 1 'yeild' statement

In [29]:
```
def normal_genertor():
    num=1
    while num<=50:
        yield num
        num+=1
        # return num

print(normal_genertor().__next__())
print(normal_genertor().__next__())
print(normal_genertor().__next__())
print(normal_genertor().__next__())
```

```
1
1
1
1
```

- Humne sab print statements me normal_generator() object hi liya hai.
- Ab function ko 1 variable me assign karake next() ka use karte hai.

In [30]:
```
number = normal_genertor()
print(number)
# Generator function humesha 1 object return karta hai

print(next(number))
print(next(number))
print(next(number))
print(next(number))
print(number.__next__())
print(number.__next__())
```

```
<generator object normal_genertor at 0x000002076C801240>
1
2
3
4
5
6
```

- Ab dekho hume pata hai generator 1 object return karta hai.
- Or kisi variable me assign karane ke baad bhi object hi return karta hai until and unless we don't use next() and __ next __() function.
- Toh ab 1 kaam karte hai function me num ko return kara kar dekhte hai.

Generator Function with return value

```
In [ ]: def normal_genertor():
            num=1
            while num<=50:
                yield num
                num+=1
                return num

        print(normal_genertor().__next__())
        print(normal_genertor().__next__())
        print(normal_genertor().__next__())
        print(normal_genertor().__next__())
```

```
1
1
1
1
```

```
In [37]: number = normal_genertor()
         print(number)
         # Generator function humesha 1 object return karta hai but is bar return state
         # Dekhte hai kya difference ayega

         print(next(number))
         print(next(number))
         print(next(number))
         print(next(number))
         print(number.__next__())
         print(number.__next__())
```

```
<generator object normal_genertor at 0x000002076C801240>
1
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[37], line 7
      3 # Generator function humesha 1 object return karta hai but is bar retur
n statement hai
      4 # Dekhte hai kya difference ayega
      6 print(next(number))
----> 7 print(next(number))
      8 print(next(number))
      9 print(next(number))

StopIteration: 2
```

Ab samjho Hume Error kyu mila-

- Ye yaad rakho 'normal_generator()' humesha object return karega.
- Ye function call karna par execute nahi hota.
- Jab 'next(number)' ye line execute hoti hai toh wo ye below code run karega -

```
In [38]: # num = 1
         # while num <= 50:
         #     yield num   # yield 1
```

- 'yield' par function 1 par pause ho jayega.
- Agla 'next(number)' ye age ka execution resume karega.
- And the Jab return encounter hoga toh generator stop ho jyega, value StopIteration error me chali jayegi. Or wahi error hume dikhega.

Kyu hua esa?

- 'return' kisi bhi iteration yaloop ko stop kar deta hai.
- Isliye error raise hua and programme execution stop ho gaya.

Ab Generator ki execution samjhte hai

- Generator ko call karne par wo value return nhi karta balki object return karta hai

```
In [42]: def normal_genertor():
             num=1
             while num<=50:
                 yield num
                 num+=1

         number = normal_genertor()
         number

Out[42]: <generator object normal_genertor at 0x000002076CB91CC0>
```

- Jab 'next()' me us object ko use karenge tab humara execution flow start hoga.

- Execution flow 'yield' tak chalega.

- 'yield' value 'next()' ko return karega then execution pause kardega, state save hogi memory me (state is num=2)

  yield -->> num -->> next() ------ execution stop at yield -> 1

```
In [43]: next(number)

Out[43]: 1
```

- Jab next time 'next()' call karega tab programme age execute hoga.

- Execution flow start hoga - num+=1 ya se.
- Then loop apna kaam karegi - while num<=50:
- Again 'yield' par execution pause ho jayega, state save hogi memory me (state is num=2).
- Fir jab next() call karega tab resume hoga.

Ese programme ka execution chalega.

In [45]:
```
next(number)
```

Out[45]:
```
3
```

Fibonacci Printer in Generator

In [126…
```
# Infinite
def fiboancci_generator():
    a = 0
    b = 1
    # while True:
    for i in range(20):
        yield a
        c = a + b
        a,b = b,c


fibonacci = fiboancci_generator()
for i in fibonacci:
    print(i)
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
```

```
In [93]: def details_generator():
             yield "Name: Arman Khan"
             yield "Qualification: BCA"
             yield "Skills: Python,SQL,etc."

         details=details_generator()
         details.__next__()

Out[93]: 'Name: Arman Khan'
```

```
In [94]: details.__next__()

Out[94]: 'Qualification: BCA'
```

```
In [95]: details.__next__()

Out[95]: 'Skills: Python,SQL,etc.'
```

Generators and List()

- Sometimes we need to store generated values in list.
- For this python have given us a solution rather then appending each element with loop

```
In [108… def list_generator():
             for i in range(10,-1,-1):
                 yield i

         generated_list = list(list_generator())
         print(generated_list)

        [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

We have one more method for creating a Generator -

Instead of creating a function. We can create a list comprehension but use '()' parentheses instead of '[]'.

```
In [112… generator_comprehension = (n for n in range(100))
         print(generator_comprehension)
         print(next(generator_comprehension))
         print(next(generator_comprehension))
         print(next(generator_comprehension))

        <generator object <genexpr> at 0x000002076D25F640>
        0
        1
        2
```

Memory Utilization with Generator -

- Every iterable object in python occupy memory for its entire size.
- Python generator object yield and store 1 value at a time. So, generator occupy 1 value memory at a time whether it generating infinite values.

'next()' Function -

- Built-in function
- Works on iterable objects
- Generator is also a iterator
- Iterators have '__ next __()' method

In [117…
```
obj = [12, 'Arman',43,'8934']
# print(next(obj)) # Ye error dega because lists ke pass __next__() nahi hota

itr = iter(obj)

print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
```

```
12
Arman
43
8934
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[117], line 10
      8 print(next(itr))
      9 print(next(itr))
---> 10 print(next(itr))

StopIteration: 
```

- Jab values khatam ho jati hai tab StopIteration Error raise kardeta hai

In [119…
```
list.__dict__
```

```
Out[119…  mappingproxy({'__new__': <function list.__new__(*args, **kwargs)>,
                        '__repr__': <slot wrapper '__repr__' of 'list' objects>,
                        '__hash__': None,
                        '__getattribute__': <slot wrapper '__getattribute__' of 'list'
          objects>,
                        '__lt__': <slot wrapper '__lt__' of 'list' objects>,
                        '__le__': <slot wrapper '__le__' of 'list' objects>,
                        '__eq__': <slot wrapper '__eq__' of 'list' objects>,
                        '__ne__': <slot wrapper '__ne__' of 'list' objects>,
                        '__gt__': <slot wrapper '__gt__' of 'list' objects>,
                        '__ge__': <slot wrapper '__ge__' of 'list' objects>,
                        '__iter__': <slot wrapper '__iter__' of 'list' objects>,
                        '__init__': <slot wrapper '__init__' of 'list' objects>,
                        '__len__': <slot wrapper '__len__' of 'list' objects>,
                        '__getitem__': <method '__getitem__' of 'list' objects>,
                        '__setitem__': <slot wrapper '__setitem__' of 'list' objects>,
                        '__delitem__': <slot wrapper '__delitem__' of 'list' objects>,
                        '__add__': <slot wrapper '__add__' of 'list' objects>,
                        '__mul__': <slot wrapper '__mul__' of 'list' objects>,
                        '__rmul__': <slot wrapper '__rmul__' of 'list' objects>,
                        '__contains__': <slot wrapper '__contains__' of 'list' object
          s>,
                        '__iadd__': <slot wrapper '__iadd__' of 'list' objects>,
                        '__imul__': <slot wrapper '__imul__' of 'list' objects>,
                        '__reversed__': <method '__reversed__' of 'list' objects>,
                        '__sizeof__': <method '__sizeof__' of 'list' objects>,
                        'clear': <method 'clear' of 'list' objects>,
                        'copy': <method 'copy' of 'list' objects>,
                        'append': <method 'append' of 'list' objects>,
                        'insert': <method 'insert' of 'list' objects>,
                        'extend': <method 'extend' of 'list' objects>,
                        'pop': <method 'pop' of 'list' objects>,
                        'remove': <method 'remove' of 'list' objects>,
                        'index': <method 'index' of 'list' objects>,
                        'count': <method 'count' of 'list' objects>,
                        'reverse': <method 'reverse' of 'list' objects>,
                        'sort': <method 'sort' of 'list' objects>,
                        '__class_getitem__': <method '__class_getitem__' of 'list' obje
          cts>,
                        '__doc__': 'Built-in mutable sequence.\n\nIf no argument is giv
          en, the constructor creates a new empty list.\nThe argument must be an iterab
          le if specified.'})
```

```
In [ ]:
```