Generators

- There was a lot of overhead in building an iterator; we had to implement a class with __iter__() and __next__() method, keep track of internal states, raise StopIteration when there was no values to be returned etc. This is both lengthy and counter intuitive. Generator comes into rescue in such situations.
- Python generators are a simple way of creating iterators. All the overhead that we mentioned above are automatically handled by generators in Python. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Generators

- There are two type of Generators:
  - Generator functions are coded as normal def statements, but use yield statements to return results one at a time, suspending and resuming their state between each.
  - Generator expressions are similar to the list comprehensions of the prior section, but they return an object that produces results on demand instead of building a result list.

Generator functions

- It is fairly simple to create a generator in Python. It is as easy as defining a normal function with yield statement instead of a return statement. If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function. The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

Generator functions

- Here is how a generator function differs from a normal function.
  - Generator function contains one or more yield statement.
  - When called, it returns an object (iterator) but does not start execution immediately.
  - Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
  - Once the function yields, the function is paused and the control is transferred to the caller.
  - Local variables and theirs states are remembered between successive calls.
  - Finally, when the function terminates, StopIteration is raised automatically on further calls.

Generator functions

```
def my_gen():
    """a simple generator function"""
    n = 1
    print("This is printed first")
    # Generator function contains yield statements
    yield n

    n += 1
    print("This is printed second")
    yield n

    n += 1
    print("This is printed at last")
    yield n

a = my_gen()
print(next(a))
print(next(a))
print(next(a))
print(next(a))


#output:

This is printed first
1
This is printed second
2
This is printed at last
3

Traceback (most recent call last):
StopIteration
```

Generator functions

```
def my_gen():
    """a simple generator function"""
    n = 1
    print("This is printed first")
    # Generator function contains yield statements
    yield n

    n += 1
    print("This is printed second")
    yield n

    n += 1
    print("This is printed at last")
    yield n

for item in my_gen():
   print(item)



#output:

This is printed first
1
This is printed second
2
This is printed at last
3
```

Generator functions

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
        yield my_str[i]

for char in rev_str("hello"):
   print(char, end='')


#output:
olleh
```

Generator functions

```
def gensquares(N):
    for i in range(N):
        yield i ** 2

x = gensquares(5)
print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))

#output:
0
1
4
9
16
```

Generator functions

```
def gensquares(N):
    for i in range(N):
        yield i ** 2

for i in gensquares(5):      # Resume the function
    print(i, end=' : ')

#output:

0 : 1 : 4 : 9 : 16 :
```

Why Generator functions

```
def buildsquares(n):
    res = []
    for i in range(n): res.append(i ** 2)
    return res

for x in buildsquares(5):
    print(x, end=' : ')

#output:
0 : 1 : 4 : 9 : 16 :
```

Why Generator functions

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

    def __iter__(self):
        self.n = 0
        return self

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

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

Generator functions

Now we can create an iterator and iterate through it as follows.

```
>>> a = PowTwo(4)
>>> i = iter(a)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
4
>>> next(i)
8
>>> next(i)
16
>>> next(i)
Traceback (most recent call last):
...
StopIteration
```

We can also use a **for** loop to iterate over our iterator class.

```
>>> for i in PowTwo(5):
...     print(i)
...
1
2
4
8
16
32
```

Why Generator functions

```
def powerTwo(N):
    for i in range(N+1):
            yield 2 ** i

for i in powerTwo(4):
    print(i, end=' ')

print("\n")
I=powerTwo(4)
try:
    print(next(I))
    print(next(I))
    print(next(I))
    print(next(I))
    print(next(I))
    print(next(I))
except StopIteration:
    print('End')
```

```
1 2 4 8

1
2
4
8
16
End
```

Generator function send

```
def gen():
    for i in range(10):
        X = yield i
        print(X, end=', ')

G = gen()
print('1:', next(G))
print('2:', G.send(77))
print('3:', G.send(88))
print('4:', next(G))

#output:
1: 0
77, 2: 1
88, 3: 2
None, 4: 3
```

Generator Expression

- Simple generators can be easily created on the fly using generator expressions. It makes building generators easy. Same as lambda function creates an anonymous function, generator expression creates an anonymous generator function. The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.

Generator Expression

```
my_list = [1, 3, 6, 10]
a = (x**2 for x in my_list)
print(next(a))
print(next(a))
print(next(a))
print(next(a))

#output:
1
9
36
100
```

Generator vs Comprehension

- The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time. They are kind of lazy, producing items only when asked for. For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

Generator vs Comprehension

```
my_list = [1, 3, 6, 10]
a=[x**2 for x in my_list]
print(a)


#output:
[1, 9, 36, 100]
```

```
my_list = [1, 3, 6, 10]
a = (x**2 for x in my_list)
print(next(a))
print(next(a))
print(next(a))
print(next(a))

#output:
1
9
36
100
```

Generator expression inside functions

```
my_list = [1, 3, 6, 10]
print(sum(x ** 2 for x in my_list))


#output:
146
```

Nested Generators

```
res=[x * 2 for x in [abs(x) for x in (-1, -2, 3, 4)]]
print(res)

res=list(x * 2 for x in (abs(x) for x in (-1, -2, 3, 4)))
print(res)


#output:

[2, 4, 6, 8]
[2, 4, 6, 8]
```

Extended syntax for the yield

```
def both(N):
    for i in range(N): yield i
    for i in (x ** 2 for x in range(N)): yield i
a=both(5)
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))

#output:

0
1
2
3
4
0
1
4
9
16
```

Extended syntax for the yield

```
def both(N):
    for i in range(N): yield i
    for i in (x ** 2 for x in range(N)): yield i

def both(N):
        yield from range(N)
        yield from (x ** 2 for x in range(N))

a=both(5)
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))
print(next(a))

#output:

0
1
2
3
4
0
1
4
9
16
```

Float range

```
def frange(start, stop, step):
    i = start
    while i < stop:
        yield i
        i += step


for i in frange(0.5, 1.0, 0.1):
    print(i)

#output

0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

Emulating map and zip

- **We saw the map and zip**

```
S1 = 'abc'
S2 = 'xyz123'

print(list(zip(S1, S2)))
print(list(map(abs, [-2, -1, 0, 1, 2])))
print(list(map(pow, [1, 2, 3], [2, 3, 4, 5])))

#output:

('a', 'x'), ('b', 'y'), ('c', 'z')]
[2, 1, 0, 1, 2]
[1, 8, 81]
```

Emulating map

```
print(list(map(abs, [-2, -1, 0, 1, 2])))
print(list(map(pow, [1, 2, 3], [2, 3, 4, 5])))

def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        res.append(func(*args))
    return res

print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))

def mymap(func, *seqs):
    return [func(*args) for args in zip(*seqs)]

print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))


#output:
[2, 1, 0, 1, 2]
[1, 8, 81]
[2, 1, 0, 1, 2]
[1, 8, 81]
[2, 1, 0, 1, 2]
[1, 8, 81]
```

Emulating map

```
print(list(map(abs, [-2, -1, 0, 1, 2])))
print(list(map(pow, [1, 2, 3], [2, 3, 4, 5])))

def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        yield func(*args)

print(list(mymap(abs, [-2, -1, 0, 1, 2])))
print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))


def mymap(func, *seqs):
    return (func(*args) for args in zip(*seqs))


print(list(mymap(abs, [-2, -1, 0, 1, 2])))
print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))


#output:

[2, 1, 0, 1, 2]
[1, 8, 81]
[2, 1, 0, 1, 2]
[1, 8, 81]
[2, 1, 0, 1, 2]
[1, 8, 81]
```

Emulating zip

```
S1 = 'abc'
S2 = 'xyz123'

print(list(zip(S1, S2)))

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    res  = []
    while all(seqs):
        res.append(tuple(S.pop(0) for S in seqs))
    return res

S1 = 'abc'
S2 = 'xyz123'
print(myzip(S1, S2))

#output:

[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Emulating zip

```
S1 = 'abc'
S2 = 'xyz123'

print(list(zip(S1, S2)))

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    while all(seqs):
        yield tuple(S.pop(0) for S in seqs)

S1 = 'abc'
S2 = 'xyz123'
print(list(myzip(S1, S2)))

#output:

[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Emulating zip

```
S1 = 'abc'
S2 = 'xyz123'

print(list(zip(S1, S2)))

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return [tuple(S[i] for S in seqs) for i in range(minlen)]


S1 = 'abc'
S2 = 'xyz123'
print(myzip(S1, S2))

#output:

[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Emulating zip

```
S1 = 'abc'
S2 = 'xyz123'

print(list(zip(S1, S2)))

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return (tuple(S[i] for S in seqs) for i in range(minlen))


S1 = 'abc'
S2 = 'xyz123'
print(list(myzip(S1, S2)))

#output:

[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Iterable and yield

- Technically, iteration contexts work by passing an iterable object to the iter built-in function to invoke an __iter__ method, which is expected to return an iterator object. If it's provided, Python then repeatedly calls this iterator object's __next__ method to produce items until a StopIteration exception is raised. A next built-in function is also available as a convenience for manual iterations—next(I) is the same as I.__next__().

```
class Squares:
    def __init__(self, start, stop):     # Save state when created
        self.value = start - 1
        self.stop  = stop
    def __iter__(self):                  # Get iterator object on iter
        return self
    def __next__(self):                  # Return a square on each iterat
        if self.value == self.stop:      # Also called by next built-in
            raise StopIteration ('There is no item')
        self.value += 1
        return self.value ** 2

for i in Squares(1, 5):                  # for calls iter, which calls __iter
   print(i, end=' ')                     # Each iteration calls __next__
print()
X=Squares(1, 5)
I=iter(X)
try:
    print(next(I)); print(next(I))
    print(next(I)); print(next(I))
    print(next(I)); print(next(I))
except StopIteration as err:
    print(err)

#output:
1 4 9 16 25
1
4
9
16
25
There is no item
```

```
class Squares:
    def __init__(self, start, stop):
        self.start = start
        self.stop  = stop
    def __iter__(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2

for i in Squares(1, 5):
    print(i, end=' ')

print()
X=Squares(1, 5)
I=iter(X)
try:
    print(next(I))
    print(next(I))
    print(next(I))
    print(next(I))
    print(next(I))
    print(next(I))
except StopIteration:
    print('End')
#output:

1 4 9 16 25
1
4
9
16
25
End
```