Department of Computer Science and Engineering
PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Generators in Python

Prerequisites: yield keyword and Iterators

Generator in Python is a simple way of creating an iterator.

Python generators are like normal functions which have yield statements instead of a return statement. Although functions and generators are both semantically and syntactically different.

There are two terms involved when we discuss generators.

1. Generator-Function:

Creating Python generators is as simple as creating a function with yield statement instead of the return statement. Any function with yield statement instead of the return statement is termed as Python generator.

A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

Example: A generator function that yields 1 for first time,2 for second time and 3 for third time.

In [7]:
```
# Generator functions stores the last state of function and resumes from there.
def simpleGeneratorFun():
    x=1
    yield x
    y=1
    yield x+y
    z=1
    yield x+y+z

genobj = simpleGeneratorFun()     # Create a generator object

for value in genobj:  # Iterating over the generator object using for loop
    print(value)

#print(next(genobj))
#print(next(genobj))
#print(next(genobj))
```

```
1
2
3
```

In [12]:
```
# Generator functions stores the last state of function and resumes from there.
def simpleGeneratorFun():
    x=1
    yield x
    y=1
    yield x+y
    z=1
    yield x+y+z

genobj = simpleGeneratorFun()     # Create a generator object

print(next(genobj))      # Iterating over the generator object using next() fn.
print(next(genobj))      # Iterating over the generator object using next() fn.
print(next(genobj))      # Iterating over the generator object using next() fn.
```

```
1
2
3
```

In [ ]:
```
#print(simpleGeneratorFun()) # <generator object simpleGeneratorFun at 0x000000000879B7C
```

Note: The generator function remembers the function state and position of the statement following the yield. The control is transferred when the function next() is called on this generator object again.

The statements below will cause the statements in the generator function to be executed from the position of the last yield till the next yield

2. Generator-Object:

Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for in” loop.

Example: A Python program to demonstrate use of generator object with next()

In [13]:
```
# A generator function
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

genObj = simpleGeneratorFun()   # Create a generator object

# Iterating over the generator object using next
print(next(genObj));
print(next(genObj));
print(next(genObj));
```

```
1
2
3
```

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

Difference between Python Generators and Python Functions

1) Syntactically different

Though the structure is almost same for both Python generators and functions, both are syntactically different. Functions have return statement whereas generators have yield statements.

Python Function Syntax:

In [ ]:
```
def func_name():
    #statements
    return something
```

Python Generator Syntax:

In [3]:
```
def generator_name():
    #statements
    yield something
```

2) Function execution paused and saved between multiple calls

In normal functions, whenever a function is called it executes the statements and the result is returned. When the function is called again from the same program, the function starts execution from the beginning.

In generator functions, whenever the function is called it executes the statements and yields the result and the function is paused in the same state.

When the generator function is called again, the function will resume from the same state as it was in the previous call.

3) Local variables and their current state stored

In generator functions, the current state of the local variables is stored in between function calls which make it possible for pausing the function execution and resuming from the same state while called again.

4) Multiple yield statements in the same function

In normal functions, there is only one return statement. We can only return a single value from a normal function or we have to return list or tuples for returning multiple values. On the contrary, a Python generator function can have multiple yield statements which make it easy to return multiple values.

Here is an example to yield multiple fruit names from a single generator function.

In [8]:
```
def fruits():
    yield ("Apple")
    yield ("Mango")
    yield ("Orange")
```

In [10]:
```
gobj=fruits()
for i in gobj:
    print(i)
```

```
Apple
Mango
Orange
```

5) Memory efficiency

Normal functions require more memory as they operate on the whole sequence and produce all results at once, whereas in generator only one value is produced at a time. Hence, generator functions are more memory efficient.

Characteristics of generators:

```
1) The generator function has one or more yields.
2) The generator function as well the code calling next, stay in some state of
execution simultaneously. This concept is called co-routine.
3) The generator function does not execute first time it is called - instead
returns a generator object.
4) The generator function resumes from where it had left of in the earlier
```

```
    execution of call on the generator object.
    5) The generator objects are iterable.
    6) The generators are lazy.
```

Example 1: Let's create a generator to iterate between food items.

In [4]:
```python
def food_items():
    yield ("Pizza")
    yield ("Desert")
    yield ("Nuggets")

obj = food_items()
print(obj)
```

```
<generator object food_items at 0x000001F482B04BA0>
```

In [5]:
```python
# both iterators and generators have common function __next__() .
print(next(obj))
print(next(obj))
print(next(obj))
```

```
Pizza
Desert
Nuggets
```

Note:

When we access item using **next**(), it is normal to return the first item as it happens in normal functions as well, but when we again use **next**() function anywhere in the program, it will return the next object because generator functions stores the last state of function and resumes from there. Since it produces one result at a time, it requires less memory than the normal functions.

Example 2a: A simple generator to print fibonacci numbers within a range/limit.

In [25]:
```python
def fib(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

x = fib(5)  # Create a generator object

# Iterating over the generator object using next
print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))
#print(next(x))

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

```
0
1
1
2
3
```

```
Using for in loop
0
1
1
2
3
```

Example 2b: A simple generator to print first-n fibonacci numbers

In [30]:
```python
def fib():
    # Initialize first two Fibonacci Numbers
    a, b = 0, 1
    # One by one yield next Fibonacci Number
    while True:
        yield a
        a, b = b, a + b

x = fib()   # Create a generator object

# Iterating over the generator object using next
#print(next(x))
#print(next(x))
#print(next(x))

# Iterating over the generator object using for in loop.
print("\nUsing for in loop")
n=20
for i in range(n):
    print(next(x), end=' ')
```

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

Note: If a function which in turn calls yield one or more times returns an object called the generator. No statement in the function is executed when called for the first time if the function contains yield statement.

In [7]:
```python
def mygen():
    print("one")
    yield 10
    print("two")
    yield 20
    print("three")
    yield 30
    print("four")

f = mygen()
```

f is a generator object. A generator object is iterable. So, we can call next on the iterable object.

In [8]:
```python
res = next(f)  # one
print(res)     # 10
```

```
one
10
```

When next(f) is called, the function mygen executes until and inclusive of the yield statement. The yield returns the control to the called of next(gen object) and returns the value of the expression in the yield statement.

In [9]:
```python
res = next(f)  # two
print(res)     # 20
```

```
two
```

```
20
```

This goes on until the end of the generator function is reached. At that point the generator function throws the exception stop iteration. As the generator object is iterable, it can be used in for loops.

Example 4: Let us have a look at an infinite sequence generator

In [15]:
```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

In [16]:
```
genobj=infinite_sequence()
print(next(genobj))

#for i in genobj:   # prints infinite sequence
#    print(i)
```

```
0
```

This looks like a typical function definition, except for the Python yield statement and the code that follows it. yield indicates where a value is sent back to the caller, but unlike return, you don’t exit the function afterward.

Instead, the state of the function is remembered. That way, when next() is called on a generator object (either explicitly or implicitly within a for loop), the previously yielded variable num is incremented, and then yielded again. Since generator functions look like other functions and act very similarly to them, you can assume that generator expressions are very similar to other comprehensions available in Python.

Example 5: Let us have a look at an first n-prime numbers sequence generator

In [21]:
```
#The function is_prime is a helper function which checks whether a given number is prime

def is_prime(m):
    i = 2
    while m % i != 0 :
        i += 1
    return i == m
```

The function below is a generator function – returns a generator object when it is called. Each time next is called it returns a number. The first time it returns 2 – then 3 – then starts from odd number 5. If the number is odd it yields it. Each time next is called, it examines whether the next odd number is prime and yields only prime numbers. This way this generator can generate an infinite sequence of prime numbers.

In [24]:
```
def prime_gen():
    yield 2
    yield 3
    m = 5
    while True:
        if is_prime(m) :
            yield m
        m += 2

g = prime_gen() # Create a generator object

# get next n primes
n = 15
```

```
for i in range(n):  #Iterating over the generator object
    print(next(g), end=' ')
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

Example 6:

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the yield statement whenever they want to return data. Each time next() is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed).

An example shows that generators can be trivially easy to create:

In [2]:
```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        #print(index, end="")
        yield data[index]

genobj = reverse('golf')

for char in genobj:
    print(char, end='')
```

```
flog
```

What makes generators so compact is that the __iter__() and __next__() methods are created automatically. Another key feature is that the local variables and execution state are automatically saved between calls.

Recursive Generators in Python

To this point, you know about creating Python generators.

Now you must be wondering if there is any way to use recursion in Python generators?

Well, obviously there is. Prior to the release of Python 3.3 we had to use loops inside a function to achieve recursion. But in version 3.3 Python allowed using yield from statement making it easy to use recursion. Here is an example to display odd numbers using recursion in Python generators.

Example of Recursive Generators in Python

In [3]:
```
#using recursion in generator function
def oddnum(start):
    yield start
    yield from oddnum(start+2)

#using for loop to print odd numbers till 10 from 1
genobj=oddnum(1)

for nums in genobj:
    if nums<100:
        print(nums, end=' ')
    else:
        break
```

```
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61
63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
```

Python Generator Expressions

Generator expression is the memory efficient generalization of list comprehensions with optimized performance.

With generators, we can evaluate the elements on demand. Though they don’t share the full power of generators, simple generators can be created on a fly using generator expressions.

Here is an example of generator expression to build a simple generator.

In [9]:
```
exp = (x ** 3 for x in range(1,6))

for i in exp:
    print(i,end=' ')
```

```
1 8 27 64 125
```

Generator expression produce one result at a time, making it memory efficient whereas list comprehensions generate a whole list.

As you can see, we can create a simple generator in a line to display first five perfect cube numbers.

They are same like list comprehensions except the fact parenthesis is used in generator expression instead of square brackets.

References:

1. 19_gen_iterator.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://www.w3schools.com/python/
3. https://docs.python.org/
4. https://www.geeksforgeeks.org/ generators-in-python/
5. http://www.trytoprogram.com/python-programming/python-iterators/
6. http://www.trytoprogram.com/python-programming/python-generators