

## Closure

Python closure is a nested function that allows us to access variables of the outer function even after the outer function is closed.

A Python closure is a nested function which has access to a variable from an enclosing function that has finished its execution.

A closure is a nested function that remembers and has access to variables from the enclosing scope even after the outer function has finished executing.

### Basic steps for developing Closures

1. Define outer function with parameters
2. Define inner function which performs operation using outer function (binding data of outer function)
3. Outer function returns reference of inner function

### Why closures are required?

Closures avoid declaring global variables (global variables are not secured).

### Example:

```
def fun1(x):
    def fun2():
        print(x)
    return fun2
```

```
f2=fun1(100)
f2()
f3=fun1(200)
f3()
f2()
```

### Output

```
100
200
100
```

Closures are helpful when inner function performs operation using data of outer function.

**Example:**

```
def power(num):
    def find_pow(p):
        return num**p
    return find_pow
```

```
p5=power(5)
res1=p5(2)
res2=p5(3)
p6=power(6)
res3=p6(2)
res4=p6(3)
print(res1,res2,res3,res4)
res5=p5(4)
print(res5)
res6=p6(4)
print(res5,res6)
```

**Output**

```
25 125 36 216
625
625 1296
```

**Example:**

```
def calculator(n1,n2):
    def calc(opr):
        match(opr):
            case '+':
                return n1+n2
            case '-':
                return n1-n2
            case '*':
                return n1*n2
            case '/':
                return n1/n2
            case _:
```

```
    return None  
return calc
```

```
c1=calculator(10,5)  
c2=calculator(50,20)  
r1=c1('*')  
r2=c2('-')  
r3=c1('+')  
r4=c2('*')  
print(r1,r2,r3,r4)
```

## **Output**

```
50 30 15 1000
```

## **Example:**

```
def draw_line(ch):  
    def draw(length):  
        for i in range(length):  
            print(ch,end="")  
        print()  
    return draw
```

```
draw_stars=draw_line('*')  
draw_dollar=draw_line('$')
```

```
draw_stars(10)  
draw_stars(30)  
draw_dollar(15)  
draw_dollar(40)
```

## **Output**

```
*****
```

```
*****
```

```
$$$$$$$$$$$$$$  
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

## **Generators**

### **What is generator?**

In Python, a generator is a function that returns an iterator. It produces a sequence of values one at a time using the `yield` keyword, instead of computing and storing all values in memory like a regular function using `return`. This makes generators memory-efficient, especially when dealing with large datasets or infinite sequences. When a generator function is called, it returns a generator object, and its code only executes when `next()` is called on the generator object.

Generate function returns value using `yield` keyword

Any function having `yield` keyword is generator function and this function returns iterator object (OR) when generator function is called, it returns iterator object.

### **What is difference between `return` and `yield`?**

<b>return</b>	<b>yield</b>
After returning value, <code>return</code> keyword terminates execution of function.	After returning value, <code>yield</code> keyword pause execution of function and when iterated again it resume back and continue
Normal function returns value using <code>return</code> keyword	Generator function returns value using <code>yield</code> keyword

#### **Example:**

```
def fun1():
    return 10
    return 20
    return 30
    return 40
```

```
def fun2():
    yield 10
    yield 20
    yield 30
    yield 40
```

```
x=fun1()
print(x)
y=fun2() # iterator object
value1=next(y)
value2=next(y)
```

```
value3=next(y)
value4=next(y)
#value5=next(y) Error
print(value1,value2,value3,value4)
```

### **Output**

```
10
10 20 30 40
```

### **Example:**

```
def sqr_gen(m,n):
    for value in range(m,n+1):
        yield value**2
```

```
s1=sqr_gen(1,10)
for x in s1:
    print(x,end=' ')
print()
s2=sqr_gen(1,10)
list1=list(s2)
print(list1)
```

### **Output**

```
1 4 9 16 25 36 49 64 81 100
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

### **Example:**

```
A=[10,20,30,40,50]
x=iter(A)
for value in x:
    print(value,end=' ')
print()
```

```
def rev_iterator(seq):
    for value in seq[::-1]:
        yield value
```

```
y=rev_iterator(A)
```

```
for value in y:  
    print(value,end=' ')
```

### **Output**

```
10 20 30 40 50  
50 40 30 20 10
```

### **Example:**

```
def inf_gen():  
    num=0  
    while True:  
        yield num  
        num=num+1
```

```
for x in inf_gen():  
    print("Hello")  
    if x==10:  
        break
```

### **Output**

```
Hello  
Hello
```

### **Example:**

```
def float_range(start,stop,step=1.0):  
    if start<stop:  
        while start<=stop:  
            yield float(start)  
            start=start+step  
    elif start>stop and step<0:
```

```
while start>=stop:  
    yield float(start)  
    start=start+step
```

```
for value in float_range(1,5):  
    print(value,end=' ')  
print()  
for value in float_range(5,1,-1):  
    print(value,end=' ')
```

## Output

```
1.0 2.0 3.0 4.0 5.0  
5.0 4.0 3.0 2.0 1.0
```

## Generator expression

Generator expression is single line statement, which returns iterator object.

variable-name=(expression for variable in iterable)

variable-name=(expression for variable in iterable if test)

Note: generator expression is used only one time to generate values

## Example:

```
sqr_gen=(value**2 for value in range(1,6))  
for value in sqr_gen:  
    print(value,end=' ')  
  
print()  
even_gen=(value for value in range(1,21) if value%2==0)  
for value in even_gen:  
    print(value,end=' ')  
print()  
odd_gen=(value for value in range(1,21) if value%2!=0)  
for value in odd_gen:  
    print(value,end=' ')
```

## Output

```
1 4 9 16 25  
2 4 6 8 10 12 14 16 18 20  
1 3 5 7 9 11 13 15 17 19
```

