INTERMEDIATE PYTHON

LESSON 2 |Python closures| 10-12-18

***Video tutorials*:**

https://www.youtube.com/watch?v=CHo-orWR8PI

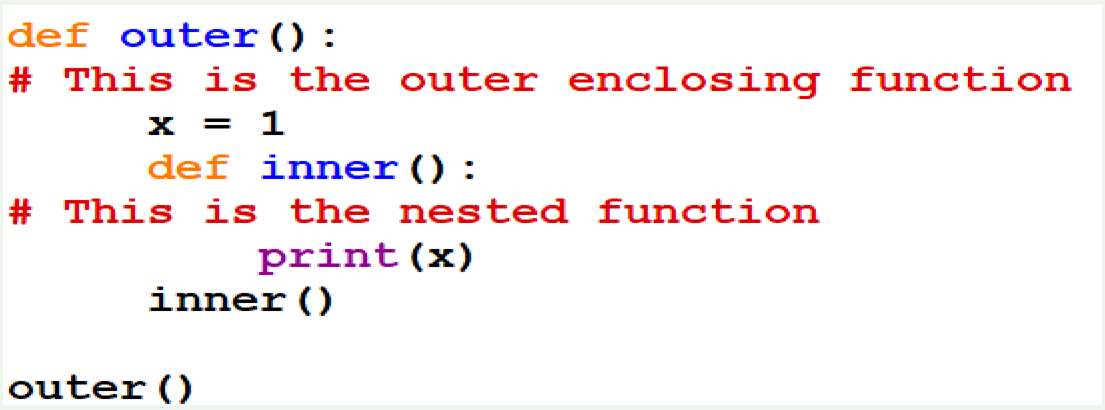
In this lesson, you will learn about python closures, understand the logic behind closures, how to create closures, **Why should we use closures,** and their significance in programming

Nonlocal variable in a nested function

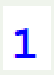
Before getting into what a closure is, we have to first understand what a nested function and nonlocal variable is.

Nested function

A function defined inside another function is called a nested function.



```
def outer():
# This is the outer enclosing function
     x = 1
     def inner():
# This is the nested function
         print(x)
     inner()

outer()
```

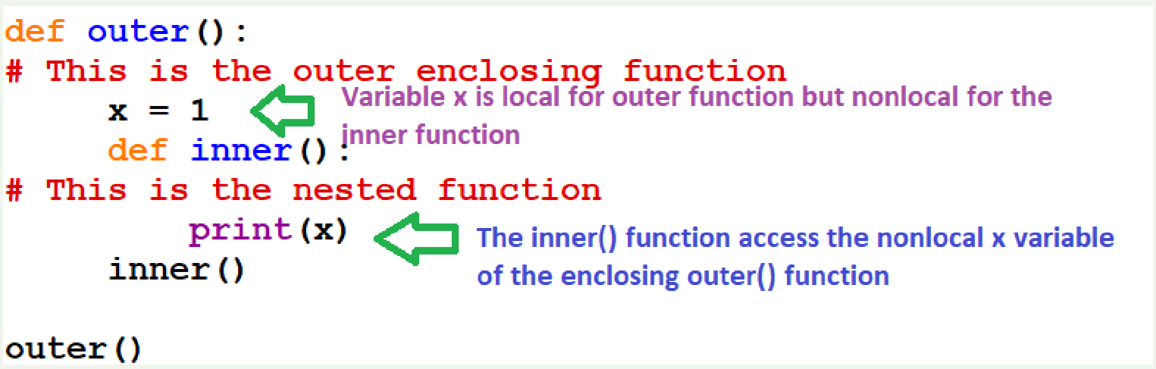


```
1
```

We can see that the nested function was able to access the non-local variable x of the enclosing function.

Nonlocal variable

Nested functions can access **variables** of the **enclosing scope**. In Python, these non-local **variables** are read only by default,



```
def outer():
# This is the outer enclosing function
    x = 1
    def inner():
# This is the nested function
        print(x)
    inner()

outer()
```

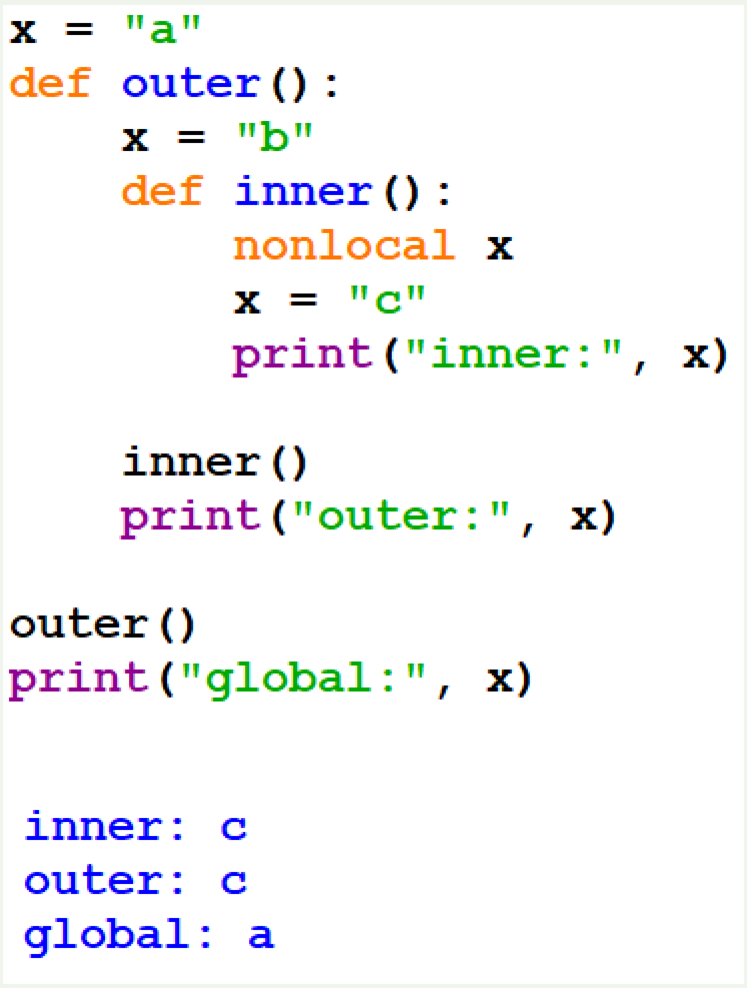
Variable x is local for outer function but nonlocal for the inner function

The inner() function access the nonlocal x variable of the enclosing outer() function

and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them. (in order to modify the variable in enclosing scope from the nested scope.

With the nonlocal keyword, you're telling python that the x in the inner() function should actually refer to the x defined in the outer() function, which is one level higher. As you can see from the result, x in both inner() and outer() is defined as "c", because it could be accessed by inner().

An example of a nested function accessing a modified non-local variable.



```
x = "a"
def outer():
    x = "b"
    def inner():
        nonlocal x
        x = "c"
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)
```

```
 inner: c
 outer: c
 global: a
```

What is a closure?

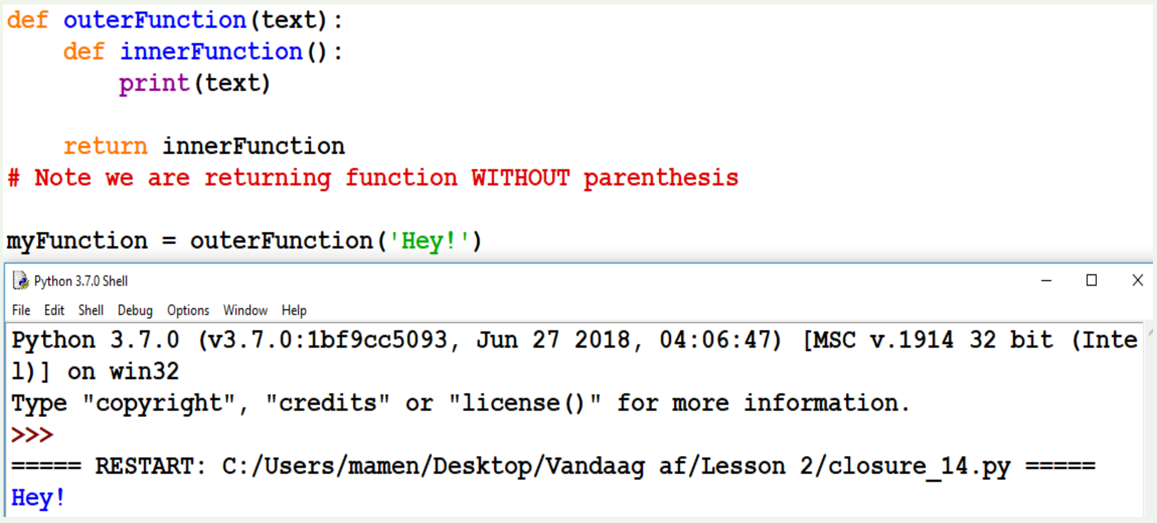
A closure is a function that has access to a variable from an enclosing scope and it's able to "remember" the value assigned to that variabkle even after the enclosing scope has finished it's execution

How to create a closure?

1.We have to create a nested function (a function inside another function).

2.This nested function has to refer to a variable defined inside the enclosing function.

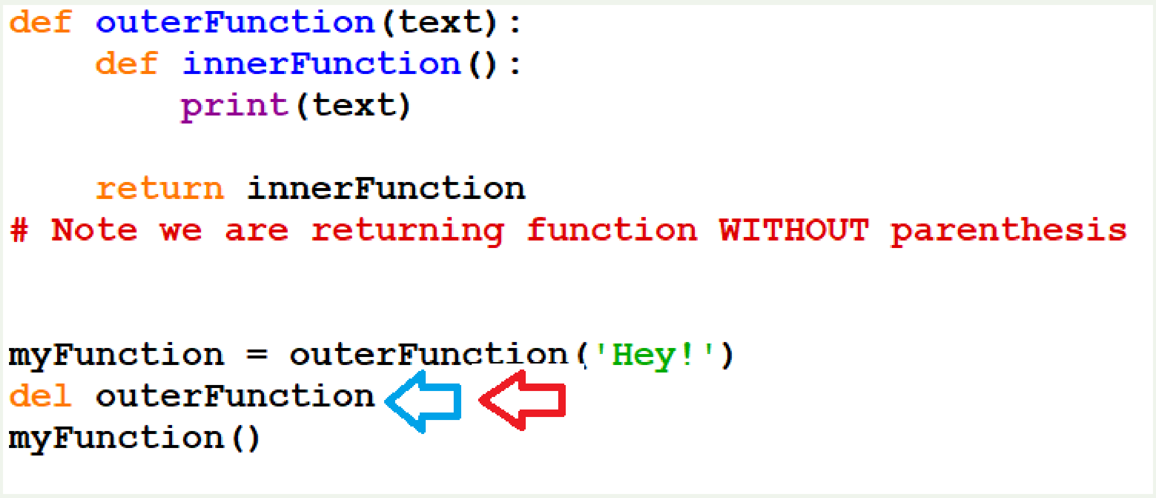
3.The enclosing function has to return(NOT CALLING) the nested function



```
def outerFunction(text):
    def innerFunction():
        print(text)

    return innerFunction
# Note we are returning function WITHOUT parenthesis

myFunction = outerFunction('Hey!')
```

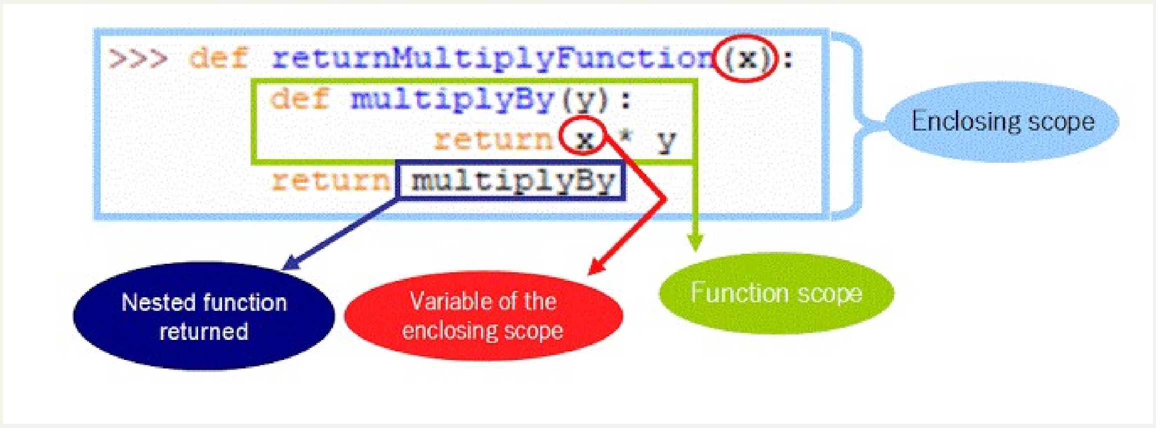
```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/mamen/Desktop/Vandaag af/Lesson 2/closure_14.py =====
Hey!
```

Another interesting fact is, that the closure remains existing even if the original creator function (in the example case it is contains factory) is deleted:



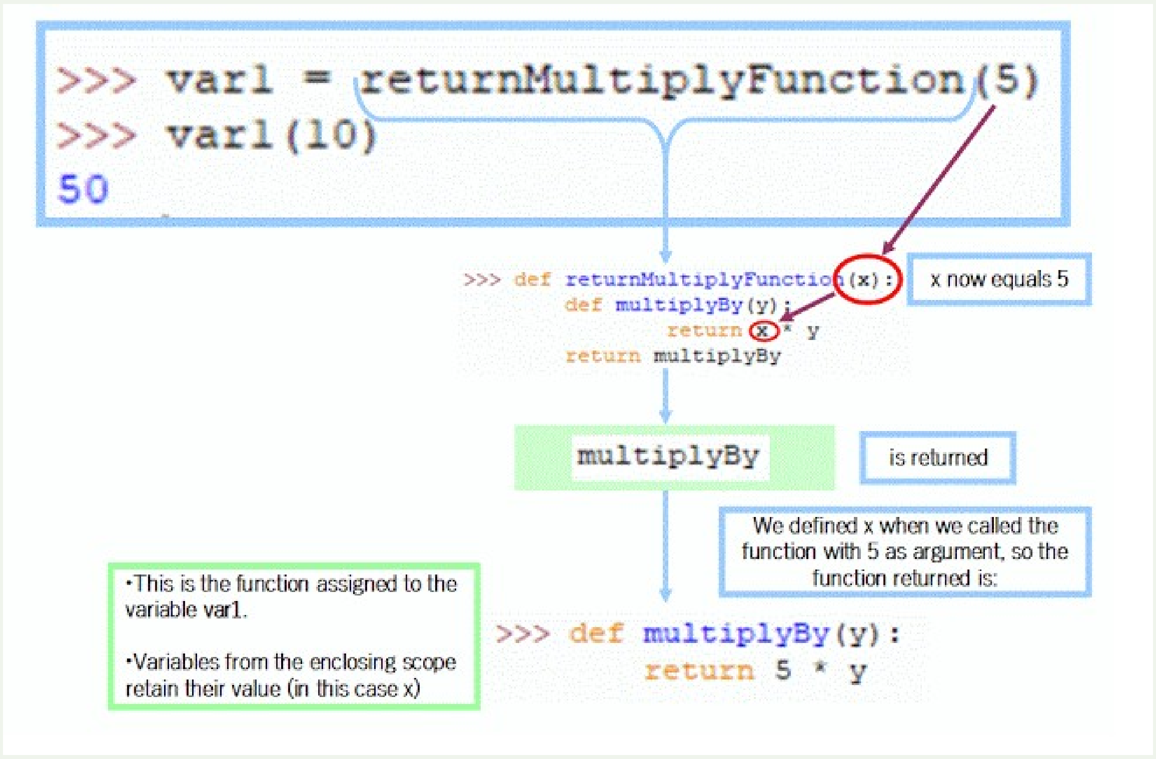
```
def outerFunction(text):
    def innerFunction():
        print(text)

    return innerFunction
# Note we are returning function WITHOUT parenthesis


myFunction = outerFunction('Hey!')
del outerFunction
myFunction()
```

The anatomy of a closure



```
>>> def returnMultiplyFunction(x):
        def multiplyBy(y):
            return x * y
        return multiplyBy
```

Using of a closure :



```
>>> var1 = returnMultiplyFunction(5)
>>> var1(10)
50
```

```
>>> def returnMultiplyFunction(x):
        def multiplyBy(y):
                return x * y
        return multiplyBy
```

x now equals 5
multiplyBy
is returned
We defined x when we called the function with 5 as argument, so the function returned is:
•This is the function assigned to the variable var1.
•Variables from the enclosing scope retain their value (in this case x)

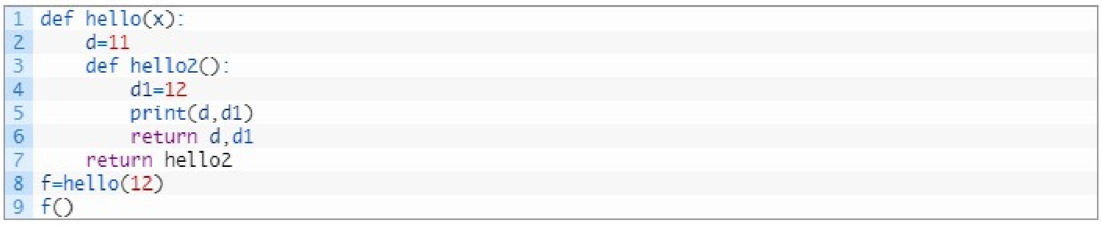
```
>>> def multiplyBy(y):
        return 5 * y
```

Why should we use closures?

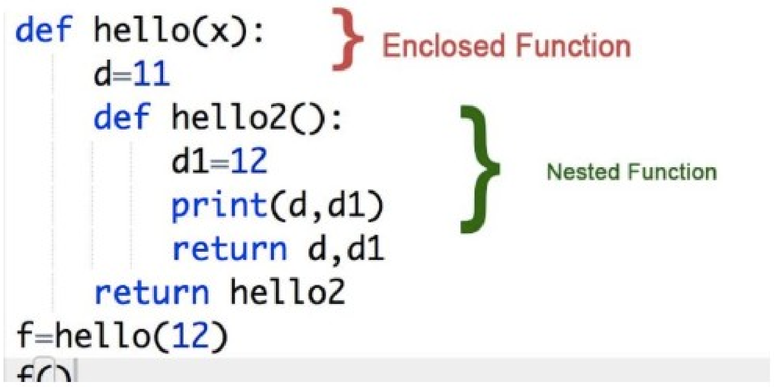
1. Closures provide some sort of data hiding as they are used as callback functions. This helps us to reduce the use of global variables.
2. Useful for replacing hard-coded constants
3. Closures prove to be efficient way when we have few functions in our code.

Examples

Example 1:

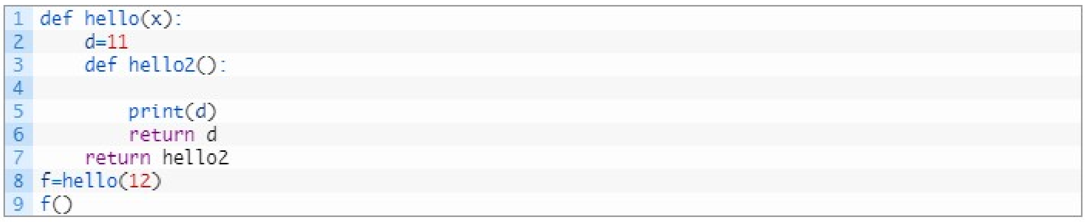


```
def hello(x):
    d=11
    def hello2():
        d1=12
        print(d,d1)
        return d,d1
    return hello2
f=hello(12)
f()
```



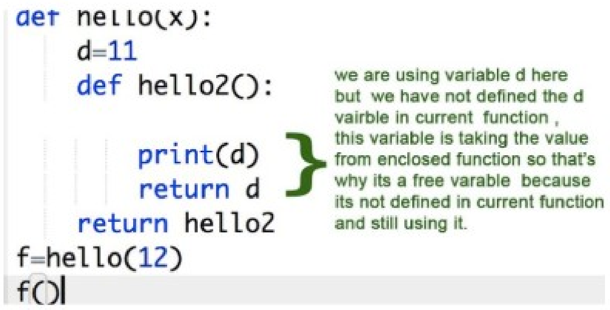
```
def hello(x):
    d=11
    def hello2():
        d1=12
        print(d,d1)
        return d,d1
    return hello2
f=hello(12)
f()
```

Example 2:



```
def hello(x):
    d=11
    def hello2():

        print(d)
        return d
    return hello2
f=hello(12)
f()
```

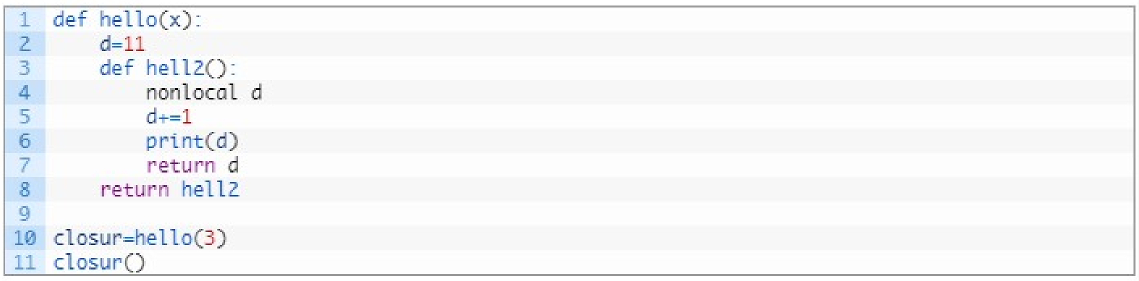
In this example you can see :



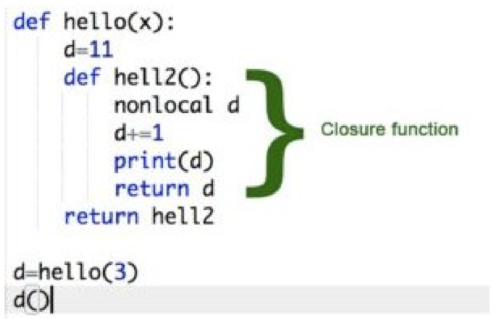
```
def hello(x):
    d=11
    def hello2():

        print(d)
        return d
    return hello2
f=hello(12)
f()
```

we are using variable d here but we have not defined the d vairble in current function , this variable is taking the value from enclosed function so that's why its a free varable because its not defined in current function and still using it.

Example 3:



```
def hello(x):
    d=11
    def hell2():
        nonlocal d
        d+=1
        print(d)
        return d
    return hell2

closur=hello(3)
closur()
```



```
def hello(x):
    d=11
    def hell2():
        nonlocal d
        d+=1
        print(d)
        return d
    return hell2

d=hello(3)
d()
```

Example 4:

```
def pop(lst):
    def get_last_item(mylst):
        return mylst[len(mylst)-1]
    lst.remove(get_last_item(lst))
    return lst

a = [1,2,3,4,6]
print(pop(a))
```

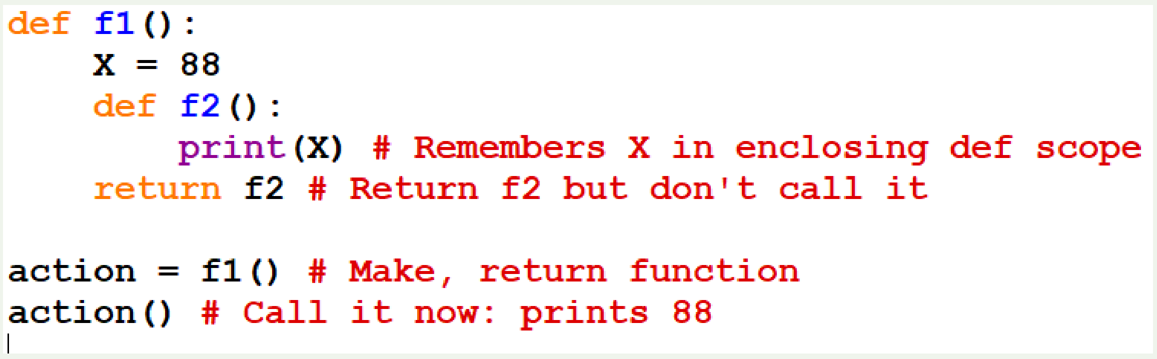
Example 5:

```
def nth_power(exponent):
    def power_of(base):
        return pow(base,exponent)
    return power_of


square = nth_power(2)
print("square of %d is %d"%(3,square(3)))

cube = nth_power(3)

print("cube of %d is %d"%(3,cube(3)))
```

Example 6:

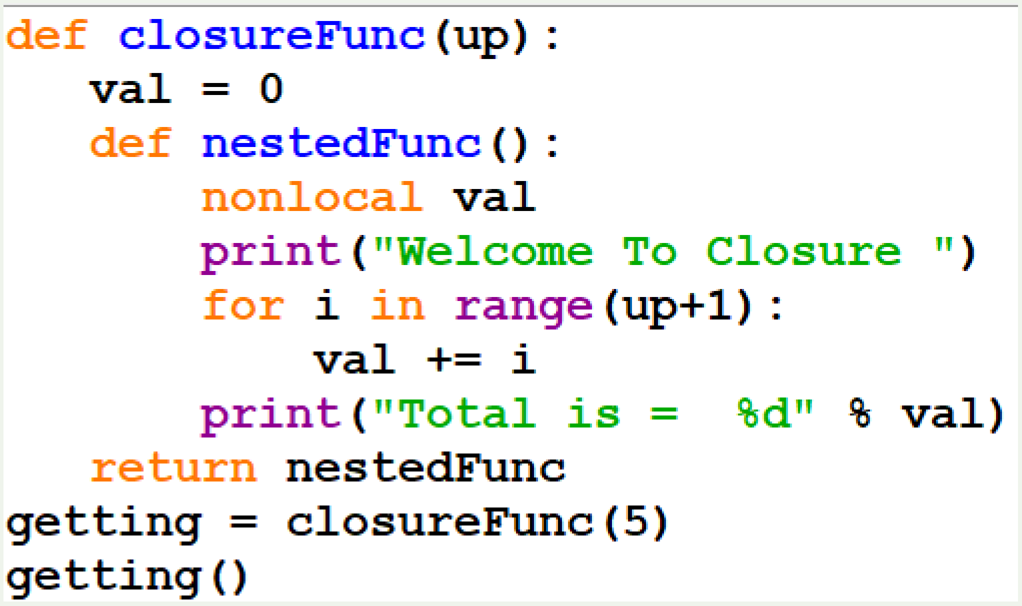
```
def is_div_2(num):
    return num%2==0
def make_is_divisble(den):
    def is_divisible(num):
        return num%den ==0
    return is_divisible

is_div_2= make_is_divisble(2)
print(is_div_2(1254))
```

Example 7:



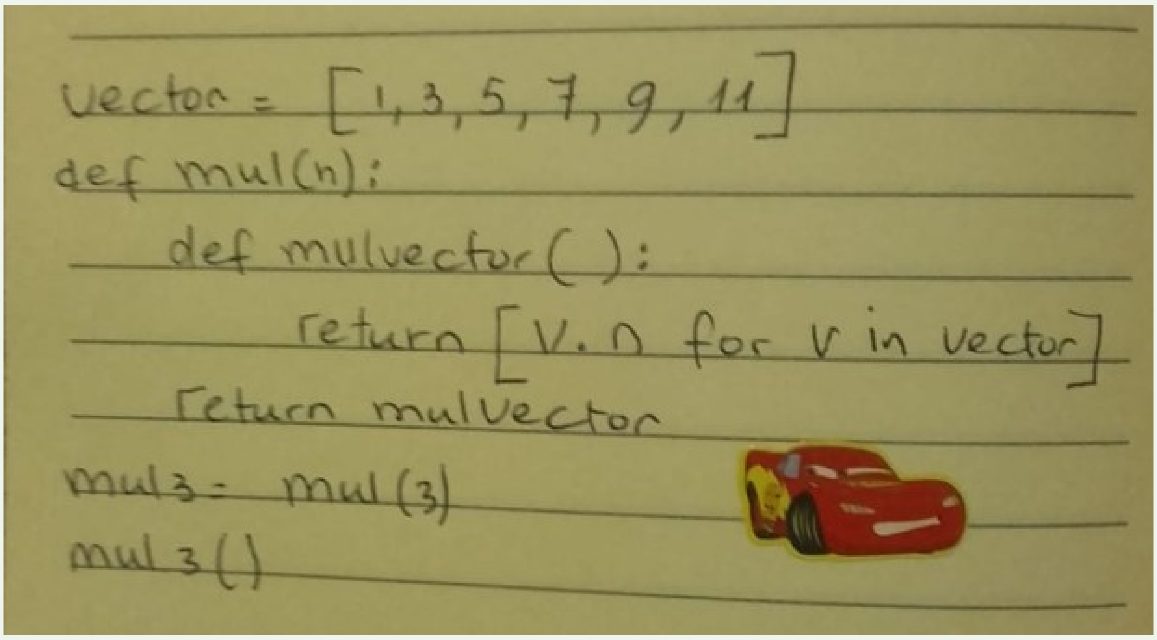
```python
def f1():
    X = 88
    def f2():
        print(X) # Remembers X in enclosing def scope
    return f2 # Return f2 but don't call it

action = f1() # Make, return function
action() # Call it now: prints 88
```

Example 8:



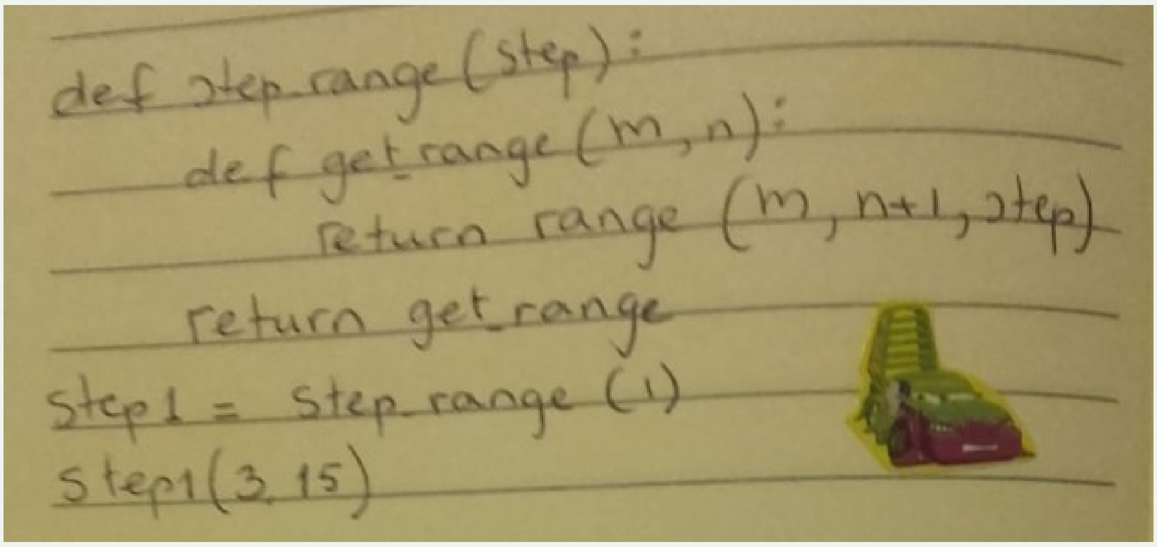
```python
def closureFunc(up):
    val = 0
    def nestedFunc():
        nonlocal val
        print("Welcome To Closure ")
        for i in range(up+1):
            val += i
        print("Total is =  %d" % val)
    return nestedFunc
getting = closureFunc(5)
getting()
```

Example 9:



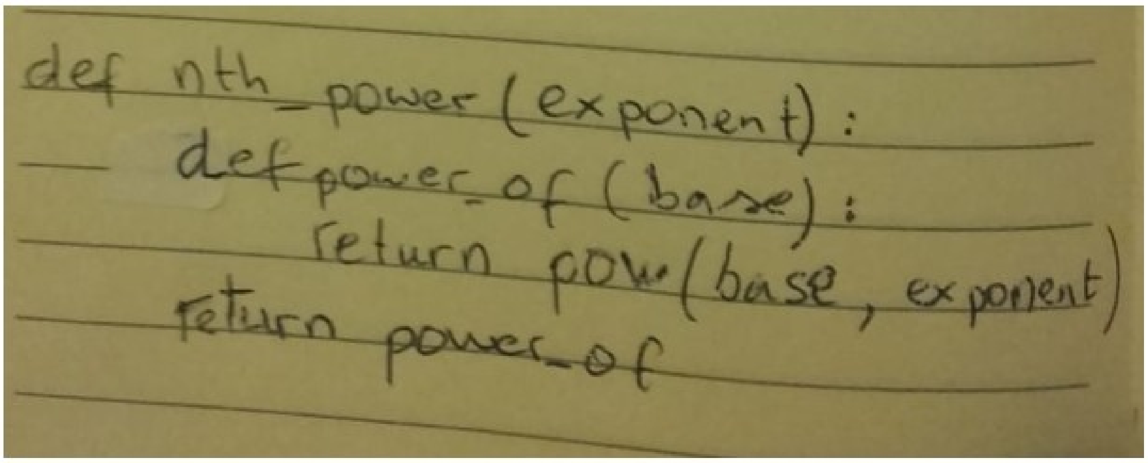
```
vector = [1, 3, 5, 7, 9, 11]
def mul(n):
    def mulvector( ):
        return [V.n for v in vector]
    return mulVector
mul3 = mul(3)
mul3()
```

Example 10:

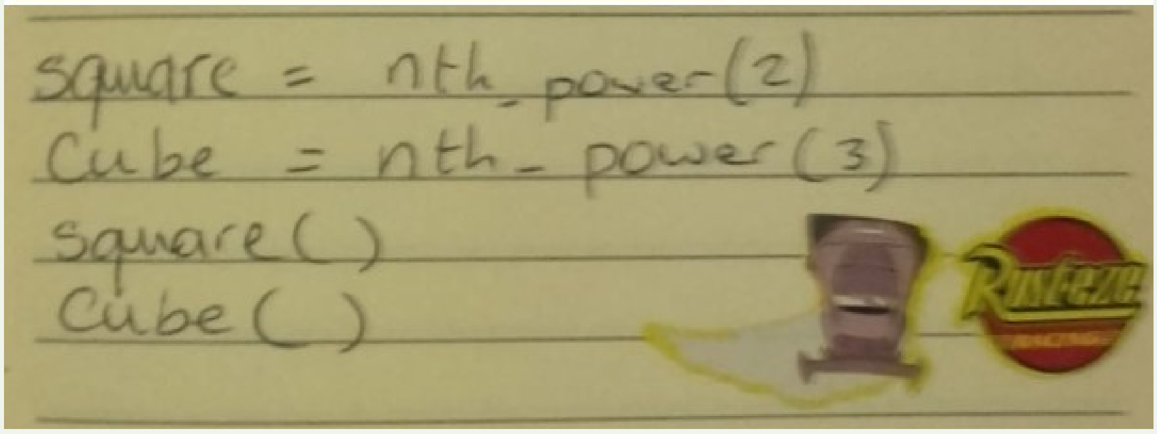


```
def step_range(step):
    def get_range(m, n):
        return range (m, n+1, step)
    return get_range
step1 = step_range (1)
step1(3, 15)
```

Example 11:



```
def nth_power(exponent):
    def power_of(base):
        return pow(base, exponent)
    return power_of
```



```
square = nth_power(2)
Cube = nth_power(3)
square()
Cube()
```