😊 Nested Statements and Scope

1. Nested Statements

Nested statements occur when one statement is placed inside another. In Python, this is commonly seen with **nested functions** or **nested loops**.

- **Nested Functions**: A function defined inside another function. The inner function can only be accessed within the outer function.
- **Nested Loops**: One loop inside another, often used in matrix operations or iterating over nested lists.

2. Scope in Python

Scope refers to the region of a program where a variable is accessible. Python follows the **LEGB rule**, which determines how variables are looked up:

1. **Local Scope (L)**: Variables defined inside a function and accessible only within that function.
2. **Enclosing Scope (E)**: Variables in the outer function when a function is nested inside another.
3. **Global Scope (G)**: Variables defined at the top level of a script or module.
4. **Built-in Scope (B)**: Predefined names in Python, such as len() or print().

😄 To begin with the Lab

1. Here we have defined a variable x = 25, then defined a function printer (), and inside the printer function we are telling the value of x is 50.
2. Then we printed the value of both x and we got the right output each time.

[1]:
```
x = 25

def printer():
    x = 50
    return x

# print(x)
# print(printer())
```

What do you imagine the output of printer() is? 25 or 50? What is the output of print x? 25 or 50?

[2]:
```
print(x)
```

```
25
```

[3]:
```
print(printer())
```

```
50
```

3. This is where the idea of scope comes in. Python has a set of rules it follows to decide what variables (such as **x** in this case) you are referencing in your code. This idea of scope in your code is very important to understand in order to properly assign and call variable names.
4. The variable x is **local** to the lambda function. **Local Scope**: The variable x is only defined inside the lambda function and cannot be accessed outside of it.
5. **Function Argument**: x is a parameter of the lambda function, which means it exists only when the function is called.

[4]:
```
# x is local here:
f = lambda x:x**2
```

6. The global variable is "This is a global name", but inside the function, a new variable "name" is set to "Sammy". The inner function uses the "name" from its enclosing function, so when greet() is called, it prints "Hello Sammy" instead of the global name.

[5]:
```
name = 'This is a global name'

def greet():
    # Enclosing function
    name = 'Sammy'

    def hello():
        print('Hello '+name)

    hello()

greet()
```

```
Hello Sammy
```

7. In Jupyter, there's a quick way to test for global variables is to see if another cell recognizes the variable

[6]:
```
print(name)
```

```
This is a global name
```

[7]:
```
len
```

[7]:
```
<function len>
```

8. This code shows that changing a parameter inside a function does **not** affect the global variable. Initially, x is set to 50 globally. Inside func, x is printed, changed to 2, and printed again. Once the function ends, the **global** x remains 50.

[8]:
```
x = 50

def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

```
x is 50
Changed local x to 2
x is still 50
```

9. This code shows how using the **global** keyword lets a function modify a variable outside its local scope. Initially, x is 50. The function declares global x, then changes x to 2. Because of the global declaration, this change **persists** after the function ends, so x becomes 2 everywhere.

[9]:
```
x = 50

def func():
    global x
    print('This function is now using the global x!')
    print('Because of global x is: ', x)
    x = 2
    print('Ran func(), changed global x to', x)

print('Before calling func(), x is: ', x)
func()
print('Value of x (outside of func()) is: ', x)
```

```
Before calling func(), x is:  50
This function is now using the global x!
Because of global x is:  50
Ran func(), changed global x to 2
Value of x (outside of func()) is:  2
```