# Nested Statements & Scope

→ When you create a variable name in Python the name is stored in name-space.
→ They also have scope which determines the visibility of that name to other parts of the code.

For eg.

In: 
```
x=25

def printer():
        x=50
        return x
```

In: 
```
print(x)
```

O/p: 
```
25.
```

In: 
```
print(printer())
```

O/p: 
```
50
```

## Rules:

1) Name assignments would create or change local names by default

2) Four scopes,
→ local
→ enclosing functions
→ ~~built~~ in global
→ ~~g built built~~
→ built in

3) Names declared in global & non-local statements map assigned names to enclosing module & function scope.

LEGB Rule

L: Local - Names assigned in any way within a function (def or lambda), & not declared global in funcⁿ

E: Enclosing funcⁿ locals - Names in the local scope of any & all enclosing functions (def or lambda), from inner to outer.

G: Global (module) - Names assigned at the top level of a module file, or declared global in a def within a file

B : Built-in - Names preassigned in the built-in names-module : open, range, SyntaxError...

E.g.s:

Ex: Locals

In: f = lambda x: x**2

## Enclosing funcⁿ locals

This occurs in nested funcⁿs

In:
```
name = 'global'

def greet():
    # Enclosing funcⁿ

    name = 'sam'

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

    hello()

greet()
```

O/P:
```
Hello Sammy
```

## Global

In: print (name)

O/p: global

## Built-in

These are built-in function names in Python (don't overwrite).

In: len

O/p: <function len>

## Local Variables

When you declare variables inside a funcn, they aren't related in any way to other variables with the same name & outside the funcn. i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

Eg:

```
x = 50

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

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

o/p:

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

The first time that we print the value of the name x is with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the funcⁿ defⁿ.

Next, we assign the value 2 to x. The name x is local to our funcⁿ. So, when we change the value of x in the funcⁿ, the x defined in the main block remains unaffected.

With the last print st., we display the value of x as defined in the main block, thereby confirming that ~~is~~ it is actually unaffected by the local assignment within the prev. called funcⁿ.

# The global statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any scope such as functions or classes), then you have to tell Python that the name is not local but it is global. We do this using the global statement. It is impossible to assign a value to a variable defined outside a function without the global statement.

You can use ~~that~~ the values of such variables defined outside the funcⁿ (assuming there is no variable with the same name within the function). However, this is not encouraged & should be avoided since it becomes unclear to the reader of the program as to where that variable's defⁿ is.

Ex:
```
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)
```

o/p:
```
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
```