Chapter 7: Scope of names

The scope of names in Python are maintained by ***Namespaces***, which are dictionaries that list the names of the objects (references) and the objects themselves.

As we have seen that names are not pre-defined thus Python uses the code block of the assignment of a name to associate it with a particular namespace. In other words, the place where you assign a name in your source code determines its scope of visibility.

Python uses `lexical` scoping, which means that variable scopes are determined entirely by their locations in the source code and not by function calls.

Rules for names inside **Functions** are as follows

- Names assigned inside a `def` can only be seen by the code within that `def` and cannot be referred from outside the function.
- Names assigned inside a `def` do'nt clash with variables from outside the `def`. i.e. a name assigned outside a `def` is a completely different variable from a name assigned inside that `def`.
- If a variable is assigned outside all `defs`, then it is global to the entire file and can be accessed with the help of `global` keyword inside the `def`.

Normally, the names are defined in two dictionaries, which can be accessed through the functions `locals()` and `globals()`. These dictionaries are updated dynamically at runtime.

Namespaces

Global variables can be overshadowed by local variables (because the local scope is consulted before the global scope). To avoid this, you must declare the variable as global in the local scope.

example:

```
def addlist(lists):
    """
    Add lists of lists, recursively
    the result is global
    """
    global add
#     add = 0
    for item in lists:
#         print(item, "=>", add)
        if isinstance(item, list): # If item type is list
            addlist(item)
        else:
            add += item # add = add + item

add = 0
addlist([[1, 2], [3, 4, 5], 6])

print(add)
```

```
21
```

```
# add = 10

def addlist(lists):
    """
    Add lists of lists, recursively
    the result is global
    """
    global add2

    for item in lists:
        if isinstance(item, list): # If item type is list
            addlist(item)
        else:
            if 'add2' in globals():
                add2 += item
            else:
                print("Creating add")
                add2 = 1

addlist([[1, 2], [3, 4, 5], 6])

print(add2)
```

```
41
```

Using global variables is not considered a good development practice, as they make the system harder to understand, so it is better to avoid their use. The same applies to overshadowing variables.

```
#add = 10

def addlist(lists):
    """
    Add lists of lists, recursively
    the result is global
    """
    global add

    for item in lists:
        if isinstance(item, list): # If item type is list
            addlist(item)
            x = 100
        else:
            add += item
        print(x)

addlist([[1, 2], [3, 4, 5], 6])

print(add)
```

```
---------------------------------------------------------------------------

UnboundLocalError                         Traceback (most recent call last)

<ipython-input-3-0cd1c571bef2> in <module>()
     17         print(x)
     18 
---> 19 addlist([[1, 2], [3, 4, 5], 6])
     20 
     21 print(add)


<ipython-input-3-0cd1c571bef2> in addlist(lists)
     10     for item in lists:
     11         if isinstance(item, list): # If item type is list
---> 12             addlist(item)
     13             x = 100
     14         else:


<ipython-input-3-0cd1c571bef2> in addlist(lists)
     15             add += item
     16 
---> 17         print(x)
     18 
     19 addlist([[1, 2], [3, 4, 5], 6])


UnboundLocalError: local variable 'x' referenced before assignment
```

```
def outer():
    a = 0
    b = 1

    def inner():
        print(a)
        print(b)
        # b = 4

    inner()

outer()
```

```
0
1
```

NOTE: - A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects.

```
def outer():
    a = 0
    b = 1

    def inner():
        print(a)
        print(b)
        b = 4

    inner()

outer()
```

```
0



---------------------------------------------------------------------------

UnboundLocalError                         Traceback (most recent call last)

<ipython-input-5-7d23a109e025> in <module>()
     10     inner()
     11 
---> 12 outer()


<ipython-input-5-7d23a109e025> in outer()
      8         b = 4
      9 
---> 10     inner()
     11 
     12 outer()


<ipython-input-5-7d23a109e025> in inner()
      5     def inner():
      6         print(a)
----> 7         print(b)
      8         b = 4
      9 
```

```
UnboundLocalError: local variable 'b' referenced before assignment
```

```python
# todo - Copy to scope section
def List_fun(l, a=[]):
    """function takes 2 parameters list having values and empty list."""

    for i in l:
        #checking whether the values are list or not
        if isinstance(i, list):
            List_fun(i, a)
        else:
            a.append(i)
#          print(a)
    return a
b=[]
l2=List_fun([[1,2],[3,[4,5]],6,7], b)
# print(l2)
print(b)
```

```
[1, 2, 3, 4, 5, 6, 7]
```