17 Namespaces

"Scope it out..."

Contents

Symbol Table

- Variable names, function names and class names are in general called identifiers.
- While interpreting our program Python interpreter creates a symbol table consisting identifiers and relevant information about each identifier.
- The relevant information includes the type of the identifier, its scope level and its location in memory.
- This information is used by the interpreter to decide whether the operations performed on the identifiers in our program should be permitted or not.
- For example, suppose we have an identifier whose type has been marked as tuple in the symbol table. Later in the program if we try to modify its contents, interpreter will report an error as a tuple is immutable.

Namespace

- As the name suggests, a namespace is a space that holds names (identifiers).
- Programmatically, a namespace is a dictionary of identifiers (keys) and their corresponding objects (values).
- An identifier used in a function or a method belongs to the **local namespace**.
- An identifier used outside a function or a method belongs to the **global namespace**.
- If a local and a global identifier have the same name, the local identifier shadows out the global identifier.
- Python assumes that an identifier that is assigned a value in a function/method is a local identifier.
- If we wish to assign a value to a global identifier within a function/method, we should explicitly declare the variable as global using the **global** keyword.

```
def fun( ) :
    # name conflict. local a shadows out global a
    a = 45

    # name conflict, use global b
    global b
    b = 6.28

    # uses local a, global b and s
    # no need to define s as global, since it is not being changed
    print(a, b, s)

# global identifiers
a = 20
b = 3.14
s = 'Aabra Ka Daabra'
fun( )
print(a, b, s)     # b has changed, a and s are unchanged
```

globals( ) and *locals( )*

- Dictionary of identifiers in global and local namespaces can be obtained using built-in functions **globals( )** and **locals( )**.
- If **locals( )** is called from within a function/method, it returns a dictionary of identifiers that are accessible from that function/method.
- If **globals( )** is called from within a function/method, it returns a dictionary of global identifiers that can be accessed from that function/method.
- Following program illustrates usage of **globals( )** and **locals( )**:

```
def fun( ) :
    a = 45
    global b
    b = 6.28
    print(locals( ))
    print(globals( ))

a = 20
b = 3.14
s = 'Aabra Ka Daabra'
```

```
print(locals( ))
print(globals( ))
fun( )
```

On execution of this program, we get the following output:

```
{'a': 20, 'b': 6.28, 's': 'Aabra Ka Daabra'}
{'a': 20, 'b': 6.28, 's': 'Aabra Ka Daabra'}
{'a': 45}
{'a': 20, 'b': 6.28, 's': 'Aabra Ka Daabra'}
```

The first, second and last line above shows abridged output. At global scope **locals( )** and **globals( )** return the same dictionary of global namespace.

Inside **fun( ) locals( )** returns the local namespace, whereas **globals( )** returns global namespace as seen from the output above.

Where to use them?

- Apart from finding out what all is available in the local and global namespace, **globals( )** and **locals( )** can be used to access variables using strings. This is shown in the following program:

```
a = 20
b = 3.14
s = 'Aabra Ka Daabra'
lst = ['a', 'b', 's']
for var in lst :
    print(globals( )[var])
```

On execution it produces the following output:

```
20
3.14
Aabra Ka Daabra
```

globals( )[var] gives the current value of **var** in global namespace.

- Using the same technique we can call different functions through the same variable as shown below:

```
def fun1( ) :
    print('Inside fun1')
```

```
def fun2( ) :
    print('Inside fun2')

def fun3( ) :
    print('Inside fun3')

lst = ['fun1', 'fun2', 'fun3']
for var in lst :
    globals( )[var]( )
```

On execution it produces the following output:

```
Inside fun1
Inside fun2
Inside fun3
```

Inner Functions

- An inner function is simply a function that is defined inside another function. Following program shows how to do this:

```
# outer function
def display( ) :
    a = 500
    print ('Saving is the best thing...')

    # inner function
    def show( ) :
        print ('Especially when your parents have done it for you!')
        print(a)

    show( )
display( )
```

On executing this program, we get the following output:

```
Saving is the best thing...
Especially when your parents have done it for you!
500
```

- **show( )** being the inner function defined inside **display( )**, it can be called only from within **display( )**. In that sense, **show( )** has been encapsulated inside **display( )**.

- The inner function has access to variables of the enclosing function, but it cannot change the value of the variable. Had we done **a = 600** in **show( )**, a new local **a** would have been created and set, and not the one belonging to **display( )**.

Scope and LEGB Rule

- Scope of an identifier indicates where it is available for use.
- Scope can be Local (L), Enclosing (E), Global (G), Built-in (B). Scope becomes more and more liberal from Local to Built-in. This can be best understood though the program given below.

```
def fun1( ) :
    y = 20
    print(x, y)
    print(len(str(x)))

    def fun2( ) :
        z = 30
        print(x, y, z)
        print(len(str(x)))

    fun2( )

x = 10
print(len(str(x)))
fun1( )
```

Output of the program is given below:

```
2
10 20
2
10 20 30
2
```

- **len**, **str**, **print** can be used anywhere in the program without importing any module. So they have a built-in scope.
- Variable **x** is created outside all functions, so it has a global scope. It is available to **fun1( )** as well as **fun2( )**.
- **fun2( )** is nested inside **fun1( )**. So identifier **y** created in **fun1( )** is available to **fun2( )**. When we attempt to print **y** in **fun2( )**, it is not

found in **fun2( )**, hence the search is continued in the enclosing function **fun1( )**. Here it is found hence its value 20 gets printed. This is an example of enclosing scope.

- Identifier **z** is local to **fun2( )**. So it is available only to statements within **fun2( )**. Thus it has a local scope.

P</> Programs

Problem 17.1

Write a program that nests function **fun2( )** inside function **fun1( )**. Create two variables by the name **a** in each function. Prove that they are two different variables.

Program

```
def fun1( ) :
    a = 45
    print(a)
    print(id(a))

    def fun2( ) :
        a = 90
        print(a)
        print(id(a))

    fun2( )

fun1( )
```

Output

```
45
11067296
90
11068736
```

Tips

- Function **id( )** gives the address stored in a variable. Since the addresses in the output are different, it means that the two **a**'s are referring to two different values

---

Problem 17.2

Write a program that proves that the dictionary returned by **globals( )** can be used to manipulate values of variables in it.

Program

```
a = 10
b = 20
c = 30
globals( )['a'] = 25
globals( )['b'] = 50
globals( )['c'] = 75
print(a, b, c)
```

Output

```
25 50 75
```

Tips

- **globals( )** returns a dictionary of identifiers and their values. From this dictionary specific identifier can be accessed by using the identifier as the key.
- From the output it is evident that we are able to manipulate variables **a**, **b**, **c**.

Problem 17.3

Write a program that proves that if the dictionary returned by **locals( )** is manipulated, the values of original variables don't change.

Program

```
def fun( ) :
    a = 10
    b = 20
    c = 30
    locals( )['a'] = 25
    locals( )['b'] = 50
    locals( )['c'] = 75
    print(a, b, c)

fun( )
```

Output

```
10 20 30
```

Tips

- **locals( )** returns a 'copy' of dictionary of identifiers that can be accessed from **fun( )** and their values. From this dictionary specific identifier can be accessed by using the identifier as the key.
- From the output it is evident that though we do not get any error, the manipulation of variables **a**, **b**, **c** does not become effective as we are manipulating the copy.

[A] State whether the following statements are True or False:

(a) Symbol table consists of information about each identifier used in our program.

(b) An identifier with global scope can be used anywhere in the program.

(c) It is possible to define a function within another function.

(d) If a function is nested inside another function then variables defined in outer function are available to inner function.

(e) If a nested function creates a variable with same name as the one in the outer function, then the two variables are treated as same variable.

(f) An inner function can be called from outside the outer function.

(g) If a function creates a variable by the same name as the one that exists in global scope, then the function's variable will shadow out the global variable.

(h) Variables defined at global scope are available to all the functions defined in the program.

[B] Answer the following questions:

(a) What is the difference between the function **locals( )** & **globals( )**?

(b) Would the output of the following print statements be same or different?

```
a = 20
b = 40
print(globals( ))
print(locals( ))
```

(c) Which different scopes can an identifier have?

(d) Which is the most liberal scope that an identifier can have?