Variable Scopes

June 23, 2018

0.0.1 Python Tutorial: Variable Scope- Understanding the LEGB rule and global/nonlocal statements

LEGB - Local, Enclosing, Global, Built-in

Local - are variables defined within a function Enclosing - are variables in local scope of a enclosing function Global are the variables defined at the top level of a module or explicitly declared global using the global keyword Built-in's are the names that are pre assigned in python

Abbreviation order determined what a variable is assigned to.

Python checks in Local, then enclosing, then global and then in built-in

```
In [2]: x = 'global x' # a global string as in main body

        def test():
            y = 'local y'
            print(y)

        test() # finds a local y variable and prints it using LEGB rule
```

```
local y
```

```
In [4]: x = 'global x' # a global string as in main body

        def test():
            y = 'local y'
            #print(y)
            print(x)

        test() # finds a global x variable and prints it using LEGB rule
```

```
global x
```

```
In [6]: x = 'global x' # a global string as in main body

        def test():
            y = 'local y'
            #print(y)
```

```
        print(x)

    test() # finds a global x variable and prints it using LEGB rule
    # print(y) # gives an error
    print(x) # x is available in this scope
```

```
global x
global x
```

```
In [8]: x = 'global x' # a global string as in main body

        def test():
            x = 'local x'
            print(x)

        test() # finds a local x variable and prints it using LEGB rule
        print(x) # this prints global x as local x is not present here
```

```
local x
global x
```

setting new value of the global x variable within the test function, use the **global** keyword

```
In [9]: x = 'global x' # a global string as in main body

        def test():
            global x
            x = 'local x' # this changes the value of x
            print(x)

        test() # finds a local x variable and prints it using LEGB rule
        print(x) # this prints global x as local x is not present here
        # both values are same as x is used as global variabel in function.
```

```
local x
local x
```

even if we don't declare the variable globally, using the global keyword allows to access the local variable globally

```
In [10]: def test():
             global x
             x = 'local x' # this changes the value of x
             print(x)

         test() # finds a local x variable and prints it using LEGB rule
         print(x) # this prints global x as local x is not present here
         # both values are same as x is used as global variabel in function.
```

```
local x
local x
```

```
In [11]: # z is a local variable in the test function

         def test(z):
             x = 'local x'
             print(z)

         test('local z')
```

```
local z
```

```
In [12]: import builtins

         m = min([5, 1, 4, 2, 3])

         #print(dir(builtins))

         print(m)
```

```
1
```

Be careful with accidental overwritting of builtin this is something that python prevents us from doing

```
In [15]: import builtins

         # this is fine
         def min():
             pass

         m = min([5, 1, 4, 2, 3])

         print(m)
```

```
        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)

        <ipython-input-15-f84c3c7abee3> in <module>()
          5     pass
          6
    ----> 7 m = min([5, 1, 4, 2, 3])
          8
```

```
 9 print(m)


TypeError: min() takes 0 positional arguments but 1 was given
```

It tells us that min function **takes 0 arguments but 1 was given**. As python finds a min function in global scope before felling back to the global scope, which did'nt takes any arguments.

Enclosing Variables -

```
In [16]: def outer():
             x = 'outer x' # this is local to our outer function

             def inner():
                 x = 'inner x' # this is local to our inner function
                 print(x)
             inner()
             print(x)

         outer()

inner x
outer x
```

```
In [17]: def outer():
             x = 'outer x' # this is local to our outer function

             def inner():
                 # x = 'inner x' # this is local to our inner function
                 print(x)
             inner()
             print(x)

         outer()

outer x
outer x
```

It first checks any x variable in local and then check if there is any x in the local scope of the enclosing function if yes then uses that variable. But commenting out the outer x will give an error.

We used **global** keyword to work with a global variable in the local scope, Similarly we have a **nonlocal** keyword to work with enclosing variables inplace of local variables in case of enclosing functions.

So, **nonlocal** allows to work with local variables of enclosing functions.

```
In [1]: def outer():
            x = 'outer x' # this is local to our outer function

            def inner():
                nonlocal x
                # this is affecting local x of our enclosing function
                x = 'inner x'
                print(x)
            inner()
            print(x)

        outer()
```

```
inner x
inner x
```

The LEGB Rule Example -

```
In [3]: x = 'global x'

        def outer():
            x = 'outer x'

            def inner():
                x = 'inner x'
                print(x)
            inner()
            print(x)

        outer()

        print(x)
```

```
inner x
outer x
global x
```

```
In [4]: x = 'global x'

        def outer():
            x = 'outer x'

            def inner():
                # x = 'inner x'
                print(x)
            inner()
```

```
            print(x)

        outer()

        print(x)
```

```
outer x
outer x
global x
```

```
In [5]: x = 'global x'

        def outer():
            # x = 'outer x'

            def inner():
                # x = 'inner x'
                print(x)
            inner()
            print(x)

        outer()

        print(x)
```

```
global x
global x
global x
```