PYTHON

LECTURE 22

Today's Agenda

- **User Defined Functions-III**
  - Variable Scope
  - Local Scope
  - Global Scope
  - Argument Passing

Variable Scopes

- The ***scope*** of a variable refers to the places from where we can see or access a variable.

- In Python , there are 4 types of scopes:
  - **Local : Inside a function body**
  - **Enclosing: Inside an outer function's body . We will discuss it later**
  - **Global: At the module level**
  - **Built In: At the interpreter level**

- In short we pronounce it as **LEGB**

Global Variable

- **GLOBAL VARIABLE**
  - A variable which is defined in the main body of a file is called a ***global*** variable.
  - It will be visible throughout the file

Local Variable

- **LOCAL VARIABLE**
  - A variable which is defined inside a function is **local** to that function.
  - It is accessible from the point at which it is defined until the end of the function.
  - It exists for as long as the function is executing.
  - Even the **parameter** in the function definition behave like **local variables**
  - **When we use the assignment operator (=) inside a function, it's default behaviour is to create a new local variable – unless a variable with the same name is already defined in the local scope.**

Example

```
s = "I love Python"
def f():
    print(s)
f()
```

Since the variable **s** is **global** , we can access it from anywhere in our code

Output:

I love Python

Example

```
s = "I love Python"
def f():
    print(s)
```

Output:

Since we have not called the function f( ) , so the statement print(s) will never get a chance to run

Example


```
def f():
    print(s)
s = "I love Python"
f()
```

Output:

I love Python

Even though the variable **s** has been declared after the function **f( )** , still it is considered to be **global** and can be accessed from anywhere in our code

Example

```
def f():
    print(s)
f()
s="I love Python"
```

Output:

NameError !

Since we have called the function f( ) , before declaring variable s , so we get NameError!

Example

```
def f():
    s="I love Python"
    print(s)
f()
```

Output:

I love Python

The variable s now becomes a local variable and a function can easily access all the local variables inside it's definition

Example

```
def f():
    s="I love Python"
    print(s)
f()
print(s)
```

Output:

I love Python

NameError!

The variable s is local and cannot be accessed from outside it's function's definition

Example

```
s="I love Python"
def f():
    s="I love C"
    print(s)
f()
print(s)
```

Output:

```
I love C
I love Python
```

If a variable with same name is defined inside the scope of function as well then Python creates a new variable in local scope of the function and uses it

Example

What if we want to use the same global variable inside the function also ?

```
s="I love Python"
def f():
    global s
    s="I love C"
    print(s)
f()
print(s)
```

To do this , we need a special keyword in Python called global. This keyword tells Python , not to create any new variable , rather use the variable from global scope

Output:

```
I love C
I love C
```

Guess The Output ?

```
s="I love Python"
def f():
    print(s)
    s="I love C"
    print(s)
f()
print(s)
```

Now , this is a special case! . In Python any variable which is changed or created inside of a function is **local**, if it hasn't been declared as a **global** variable. To tell Python, that we want to use the **global** variable, we have to explicitly state this by using the keyword "**global**"

Output:

UnboundLocalError!:

Local variable s referenced before assignment

Guess The Output ?

```
s="I love Python"
def f():
    global s
    print(s)
    s="I love C"
    print(s)
f()
print(s)
```

Output:

I love Python
I love C
I love C

Guess The Output ?

```
a=1
def f():
   print ('Inside f() : ', a)
def g():
   a = 2
   print ('Inside g() : ',a)
def h():
   global a
   a = 3
   print ('Inside h() : ',a)

print ('global : ',a)
f()
print ('global : ',a)
g()
print ('global : ',a)
h()
print ('global : ',a)
```

Output:

```
global : 1
inside f( ):1
global: 1
inside g( ): 2
global : 1
inside h( ): 3
global : 3
```

Guess The Output ?


```
a=0
if a == 0:
   b = 1
def my_function(c):
   d = 3
   print(c)
    print(d)
my_function(7)
print(a)
print(b)
print(c)
print(d)
```

Output:

7
3
0
1
NameError!

Guess The Output ?


```
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)

a, b, x, y = 1, 15, 3,4
foo(17, 4)
print(a, b, x, y)
```

Output:

42 17 4 17
42 15 3 4

Argument Passing

- There are **two** ways to pass arguments/parameters to function calls in **C programming**:
  - **Call by value**
  - **Call by reference.**

Call By Value

- In **Call by value**, **original value is not modified**.
- In **Call by value**, the value being passed to the function is locally stored by the function parameter as **formal argument**
- So , if we change the value of **formal argument**, it is changed for the **current function** only.
- These changes are not reflected in the **actual argument's** value

Call By Reference

- In **Call by reference** , the location (address) of **actual argument** is passed to **formal arguments**, hence any change made to formal arguments will also reflect in actual arguments.

- In **Call by reference**, **original value is modified** because we pass reference (address).

What About Python ?

- When asked whether **Python** function calling model is "**call-by-value**" or "**call-by-reference**", the correct answer is: **neither**.

- What Python uses , is actually called "**call-by-object-reference**"

A Quick Recap Of Variables

- We know that everything in **Python** is an **object**.
- All **numbers** , **strings** , **lists** , **tuples** etc in **Python** are objects.
- Now , recall , what happens when we write the following statement in **Python**:

  x=10

- An **object** is created in **heap** , storing the value **10** and **x** becomes the reference to that **object**.

A Quick Recap Of Variables

- Also we must recall that in **Python** we have 2 types of data : **mutable** and **immutable**.

- **Mutable types** are those which do not allow modification in object's data and examples are **int** , **float** , **string** ,**tuple** etc

- **Immutable types** are those which allow us to modify object's data and examples are **list** and **dictionary**

What Is Call By Object Reference ?

- Now , when we pass **immutable** arguments like **integers**, **strings** or **tuples** to a function, the passing acts like **call-by-value.**

- The ***object reference is passed*** to the function parameters.

- They can't be changed within the function, because they can't be changed at all, i.e. they are **immutable**.

What Is Call By Object Reference ?

- It's different, if we pass **mutable arguments**.
- They are also **passed by object reference**, but they can be **changed in place** in the function.
- If we pass a **list** to a function, elements of that **list** can be changed in place, i.e. the **list** will be changed even in the caller's scope.

Guess The Output ?

```
def show(a):
    print("Inside show , a is",a," It's id is",id(a))

a=10
print("Outside show, a i
show(a)
```

Since Python uses Pass by object reference , so when we passed a , Python passed the address of the object pointed by a and this address was received by the formal variable a in the function's argument list. So both the references are pointing to the same object

Output:

```
Outside show, a is 10   It's id is 8791162737984
Inside show , a is 10   It's id is 8791162737984
```

Guess The Output ?

```
def show(mynumbers):
    print("Inside show , mynumbers is",mynumbers)
    mynumbers.append(40)
    print("Inside show , mynumbe
mynumbers=[10,20,30]
print("Before calling show, m
show(mynumbers)
print("After calling show, my
```

Since list is a mutable type , so any change made in the formal reference a does not create a new object in memory . Rather it changes the data stored in original list

Output:

```
Before calling show, mynumbers is [10, 20, 30]
Inside show , mynumbers is [10, 20, 30]
Inside show , mynumbers is [10, 20, 30, 40]
After calling show, mynumbers is [10, 20, 30, 40]
```

Guess The Output ?

```
def show(mynumbers):
    mynumbers=[50,60,70]
    print("Inside show , mynumbers is",mynu
mynumbers=[10,20,30]
print("Before calling show, mynumb
show(mynumbers)
print("After calling show, mynu
```

If we create a new object inside the function , then Python will make the formal reference mynumbers refer to that new object but the actual argument mynumbers , will still be refering to the actual object

Output:

```
Before calling show, mynumbers is [10, 20, 30]
Inside show , mynumbers is [50, 60, 70]
After calling show, mynumbers is [10, 20, 30]
```

Guess The Output ?

```
def increment(a):
    a=a+1
a=10
increment(a)
print(a)
```

Output:

10

When we pass **n** to **increment(n)**, the function has the local variable **n** referring to the same object. Since integer is **immutable**, so Python is not able to **modify** the object's value to **11** in place and thus it created a new object. But the original variable **n** is still referring to the **same object** with the value **10**

Guess The Output ?

```
def swap(a,b):
    a,b=b,a
a=10
b=20
swap(a,b)
print(a)
print(b)
```

Output:

```
10
20
```

Guess The Output ?

```
def changetoupper(s):
    s=s.upper()
s="bhopal"
changetoupper(s)
print(s)
```

Output:

bhopal

Guess The Output ?

```
def changetoupper(s):
    s=s.upper()
    return s
s="bhopal"
s=changetoupper(s)
print(s)
```

Output:

BHOPAL