எங்கள் வாழ்வும் எங்கள் வளமும்
மங்காத தமிழ் என்று சங்கே முழங்கு ... *புரட்சிக்கவி*

NOTICE

www.DataScienceInTamil.com

Day 33,34,35 and 36 - Batch 3 - Python Language

Chapter 017 Variable Scope, Binding, Local, Global

To watch the recorded Python and Data Science videos in YouTube:

Day 33- Batch 3 - Variable Scope, Binding, Local, NonLocal, Global - Part 1
https://youtu.be/3br1bEXi5XI

Day 34- Batch 3 - Variable Scope, Binding, Local, NonLocal, Global- Part 2
https://youtu.be/pFoan4laAcE

Day 35- Batch 3 - Variable Scope, Binding, Local, NonLocal, Global - Part 3
https://youtu.be/nq3PG2PGiwE

Day 36- Batch 3 - Variable Scope, Binding, Local, NonLocal, Global - Part 4

https://youtu.be/4rP27RCBRJ0

Official Website:

https://DataScienceInTamil.com/

மேலும் முக்கிய கேள்விகள் பதில்களுக்கு :

https://www.DatascienceInTamil.com/#faq

To join DataScienceInTamil Telegram group:

இந்த குழுவில் உங்கள் நண்பர்களை இணைக்க விரும்பினால் அதற்கான லிங்க்

https://t.me/joinchat/lUZEsr-zidpjZjEx

To Join the class, please fill the form :

https://forms.gle/QFpLHwAoinFaX2cE6

Join Zoom Meeting (From Sep 26 2022 to Oct 26 2022)

https://us06web.zoom.us/j/88900302653?pwd=MVBFUlhqTTE1LzFFRUVpTzZ2S1Vsdz09

Meeting ID: 889 0030 2653

Passcode: 1234

Monday through Friday 8 PM to 10 PM IST (From Sep 26 2022 to Oct 26 2022)

We support open-source products to spread Technology to the mass.

- This is completely a FREE training course to provide introduction to Python language
- All materials / contents / images/ examples and logo used in this document are owned by the respective companies / websites. We use those contents for FREE teaching purposes only.
- We take utmost care to provide credits whenever we use materials from external source/s. If we missed to acknowledge any content that we had used here, please feel free to inform us at info@DataScienceInTamil.com.

- All the programming examples in this document are for FREE teaching purposes only.

Thanks to all the open-source community and to the below websites from where we take references / content /code example, definitions, etc., please use these websites for further reading:

- Book : Python Notes For Professionals
- https://www.w3schools.com
- https://www.geeksforgeeks.org
- https://www.askpython.com
- https://docs.python.org
- https://www.programiz.com/
- https://www.openriskmanagement.com/
- https://pynative.com/python-sets/
- https://www.alphacodingskills.com/
- https://codedestine.com/
- https://appdividend.com/

- https://freecontent.manning.com/
- https://pynative.com/python-for-loop/
- https://stackoverflow.com/
- https://datagy.io/python-isdigit
- https://pynative.com/python-range-function/
- https://www.datacamp.com/community/tutorials/functions-python-tutorial
- https://data-flair.training/blogs/python-function/
- https://problemsolvingwithpython.com/07-Functions-and-Modules/07.07-Positional-and-Keyword-Arguments/
- https://www.tutorialsteacher.com/python/callable-method
- https://www.geeksforgeeks.org/python-exception-handling/
- https://www.javatpoint.com/python-exception-handling
- https://www.tutorialspoint.com/python/python_exceptions.htm
- https://www.w3schools.com/python/python_try_except.asp
- https://www.programiz.com/python-programming/exception-handling
- https://www.datacamp.com/community/tutorials/exception-handling-python
- https://www.geeksforgeeks.org/python-exception-handling/
- https://www.javatpoint.com/python-exception-handling
- https://www.tutorialspoint.com/python/python_exceptions.htm
- https://www.w3schools.com/python/python_try_except.asp
- https://www.programiz.com/python-programming/exception-handling
- https://www.datacamp.com/community/tutorials/exception-handling-python
- www.sentinelone.com/blog/python-stack-trace-understanding-using-debug/
- https://riptutorial.com/python/example/5532/re-raising-exceptions
- https://www.techbeamers.com/python-generator/
- https://nvie.com/posts/iterators-vs-generators/

Variable Scope and Binding

What to cover today

1. What is Variable Scope and Binding
2. Local variables
3. Nonlocal Variable - Nested function
4. 2 inner fns results can be returned by outer fn
5. WITHOUT USING NONLOCAL - WHAT HAPPENS?
6. WITH USING NONLOCAL
7. Local variable DOES NOT maintain the state
8. NonLocal variable DOES maintain the state
9. Each inner fn must use nonlocal token
10. Multiple varibles in outerscope
11. Return multiple nonlocal varaibles in tuple format
12. Nonlocal cannot be used in outer fn

13. Rules / Points to ponder on nonlocal token
14. which variable has what prirority - inner fn, outer fn and global scope
15. How scope works outside a class, inside a class, inside a method
16. Global key word
17. Same global variable can be used in multiple fns.
18. Global Variables
19. Inner variable shadows outter variable
20. Using nonlocal and global together in a nested fn
21. Local Variables
22. nonlocal keyword
23. Global keyword can override ‘if..false’
24. Global keyword can override ‘if..false’, but can not modify the value
25. The del command in variables

26. vars() to find class variables
27. Use global or nonlocal to del (delete) outer scope
28. How to use delattr() in class scope variables
29. example of how to delete a class variable (is possible using delattr()
30. example of how to delete a class' outer scope variable (it is possible using global token)
31. Functions skips class' scope when looking up name
32. Local vs Global Scope
33. How to access class varaibles - (using vars(className))
34. What happens if variable name clashes?
35. How to find local variables for normal and nested fns
36. Functions within functions - nested functions
37. global vs nonlocal (Python v3 only)
38. Where and how binding occur for variables

39. Where and how binding NOT occur for variables

Variable Scope and Binding

What is binding and scope?
Binding: **An association of a name with an object**. Scope: The lifetime of a binding of a name to an object. Referencing environment: The complete set of bindings in effect at a certain point in a program.

What is binding of variable?
variable binding (plural variable bindings) (programming) **the association between a variable name (identifier) and its value**

What is meant by variable scope?
In simple terms, scope of a variable is **its lifetime in the program**. This means that the scope of a variable is the block of code in the entire program where the variable is declared, used, and can be modified. In the next section

What is a variable and variable scope?
A variable's scope determines where in a program a variable is available for use. A variable's scope is defined by where the variable is initialized or created.

What are the 3 types of scope?
JavaScript has 3 types of scope: **Block scope**. **Function scope**. **Global scope**.

Local and Global variables in python

Both these keywords are used to gain **write access** to variables which are not local to the current functions.

The global keyword declares that a name should be treated as a global variable

In Python, variables that are only referenced inside a function are implicitly local. If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.

Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring global for assigned variables provides a bar against unintended side-effects. On the other hand, if global was required for all global references, you'd be using global all the time. You'd have to declare as global every reference to a built-in function or to a component of an imported module. This clutter / mess would defeat the usefulness of the global declaration for identifying side-effects.

Local variables

```
def  add2Numbers(internal,external):
    totalMarks = internal + external
    attendance = 98
    print("Total Marks is : ", totalMarks)
    return totalMarks

result = add2Numbers(24,67)
print(result)
print(type(result))
print(id(result))

print("----------------------")
```

```
a = add2Numbers
print(a) # The prefix 0x is used in code to indicate that the number is being written in hex.
print(id(a))
print(type(a))

b = 0x012D3028
print(int, b)
print(str, b)
```

output

```
Total Marks is :  91
91
<class 'int'>
22574624
----------------------
<function add2Numbers at 0x015D3028>
22884392
<class 'function'>
<class 'int'> 19738664
<class 'str'> 19738664
```

Note:

binary uses two digits
Octal uses 8 digits
Decimal uses ten digits
hexadecimal uses sixteen digits.
==========

Nonlocal Variable

Nested function

In nested fn(function) scenario, outer fn variable can be accessed by inner fn, if we want to REASSIGN the outer fn variable – if at all we want to modify use nonlocal token

Below ex is to show outer fn varible can be accessed by inner fn (NO INCREMENT / NO MODIFICATION PROCESS IS HAPPNING HERE)

```
def outer():
    num = 0

    def inner():
```

```
        print(num)
        return num

    return inner

a = outer()
print(a())
```

OUTPUT

0

0

The above example of closure fn

========

2 inner fns example

```
def outer():
    num = 5
    def inner1():
        print("innerfn1", num + 200)
        return num

    def inner2():
        print("innerfn2", num + 300)
```

```
        return num

    return inner1(), inner2()

result1 = outer()
print(result1)
```

output

innerfn1 205

innerfn2 305

(5, 5)

2 inner fns results can be returned by outer fn

(Without using nonlocal declaration)

```
def counter():
    num = 0
    def incrementor():
```

```
        num = 10
        num = num +1
        return num
    # return incrementor()

    def testMethod():
        num = 10
        num = 10 * 10
        return num
    return incrementor(), testMethod()

a = counter()
print(a)
```

output

(11, 100)

Note, the outer fn counter has 2 inner fns (incrementor(), testMethod(),). The tasks of the 2 methods can be returned by the counter () and the **result is in tuple**

BELOW WE TRY TO INCREMENT OUTER FUNCTION VARIBLE INSIDE THE INNER FN, SO WE USE NONLOCAL

Python 3 added a new keyword called nonlocal. The **nonlocal** keyword adds a scope override to the inner scope.
One of the most common examples is to **create function that can increment:**

WITHOUT USING NONLOCAL - WHAT HAPPENS?

```
def counter():
    num = 0
    def incrementor():
        num = num + 1
        return num
    return incrementor()

counter()
```

output

```
Traceback (most recent call last):
    num = num + 1
```

UnboundLocalError: local variable 'num' referenced before assignment

If you try running this code, you will receive an UnboundLocalError because the num variable is referenced before it is assigned in the innermost function

---

WITH USING NONLOCAL

To avoid this we use ‘’nonlocal’ key word

```
def counter():
    num = 0

    def incrementor():
        nonlocal num
        num = num + 1
        print (num)
        return num
    return incrementor()

counter()
```

output

1
Notes: ie the outer fn variable is not allowed in inner fn without nonlocal TOKEN / declaration

==========

```
def counter():
    bankInitialBalance = 1000
    print('My initial bank bal is: ' ,bankInitialBalance)

    def decrementor(a):
        nonlocal bankInitialBalance
        withdrawalAmount = a
        print('withdrawal amt: ', withdrawalAmount)
        bankInitialBalance = bankInitialBalance - withdrawalAmount
        # if bankInitialBalance == 0:
        #     print('Your balance is 0')

        return  bankInitialBalance

    return decrementor

a = counter()
```

```
print("Balance after 100 withdrawal is :", a(100))
print("Balance after 200 withdrawal is :", a(200))
print("Balance after 700 withdrawal is :", a(700))
```

output

```
My initial bank bal is:  1000
withdrawal amt:  100
Balance after 100 withdrawal is : 900
withdrawal amt:  200
Balance after 200 withdrawal is : 700
withdrawal amt:  700
Balance after 700 withdrawal is : 0
```

==========

```
def bank():
    amount = 1000

    def withdraw(atm_withdraw):
        nonlocal amount
```

```
        amount = amount - atm_withdraw
        return amount

    def deposit(atm_kiosk):
        nonlocal amount
        amount = amount + atm_kiosk
        return amount
    return withdraw(100), deposit(300)

a = bank()
print(a)
```

o/p:

(900, 1200)

Local variable DOES NOT maintain the state

If we use local variable instead of nonlocal, it DOES NOT maintain the state – see below

```
def counter():
    deposit = 0
```

```
    def incrementer(a):
        # nonlocal deposit
        print("The nonlocal maintains the state ", deposit)
        balance = deposit + a
        return r"After deposit the  amount is ", balance
    return  incrementer # We have to use bracket increment here else use the bracket when call the function

c = counter()
print(c(1)) #observe: we stored the moethd in a var c, then using c, we call multiple times c()
print(c(2)) #We call using print c() ,if we don't use bracket in the above return statement
print(c(3))
```

output

```
The nonlocal maintains the state  0
('After deposit the  amount is ', 1)
The nonlocal maintains the state  0
('After deposit the  amount is ', 2)
The nonlocal maintains the state  0
```

('After deposit the amount is ', 3)

========

NonLocal variable DOES maintain the state

```
def counter():
    deposit = 0
    def incrementer(a):
        nonlocal deposit
        print("The nonlocal maintains the state ", deposit)
        deposit = deposit + a
        return r"After deposit the  amount is ", deposit
    return  incrementer # We have to use bracket increment here else use the bracket when call the function

c = counter()
print(c(1)) #observe: we stored the moethd in a var c, then using c, we call multiple times c()
print(c(2)) #We call using print c() ,if we don't use bracket in the above return statement
print(c(3))
```

The nonlocal maintains the state 0
('After deposit the amount is ', 1)

```
The nonlocal maintains the state  1
('After deposit the  amount is ', 3)
The nonlocal maintains the state  3
('After deposit the  amount is ', 6)
```

==========

Each inner fn must use nonlocal token

Each inner fn must use nonlocal token, inorder to use the outer fn variable, else it is an error – see below
(But we can not call multiple times, if we have more than one inner function – if do, we get the same results)

```
def counter():
    num = 1

    def incrementor():
        nonlocal num # each inner fn needs to declare the nonlocal token
```

```
        num = num +1
        return num
    # return incrementor()

    def testMethod():
        nonlocal num # each inner fn needs to declare the nonlocal token
        num = num + 2
        return num
    return incrementor(), testMethod()

print(counter()) # we can not call multiple times – if do, we get the same results

print(counter()) # we can not call multiple times – if do, we get the same results
```

output
(2,4)
(2,4)

=========

Multiple varibles in outerscope

There can be multiple varibles in outerscope. Inner fn can use those varibles using nonlocal token – see below

```
def counter():
    bonus = 50
    salary = 0
    total = 0

    def incrementor():
        nonlocal bonus
        bonus = bonus + 100

        nonlocal salary
        salary = salary + 200

        nonlocal total
        total = bonus  + salary
        return total
    return incrementor

a = counter()
print(a())
```

```
print(a())
print(a())
```

output
350
650
950
==========

Above code can be rewritten as below

```
def counter():
    bonus = 50 # 150, 250
    print(bonus)
    salary = 0 # 200, 400, 600
    print(salary)
    total = 0

    def incrementor():
        nonlocal bonus
        bonus = bonus + 100
        print(bonus)

        nonlocal salary
        salary = salary + 200
        print(salary)
```

```
        nonlocal total
        total = bonus + salary
        return total

    return incrementor


a = counter()
print(a())
print("============")
print(a())
print("============")
print(a())
```

output

```
50
0
150
200
350
============
250
400
650
============
```

350
600
950
==========

Return multiple nonlocal varaibles in tuple format

There can be multiple varibles in outerscope. Inner fn can use those varibles using nonlocal token and return multiple nonlocal varaibles in tuple format– see below

```
def counter():
    bonus = 50
    salary = 0
    total = 0
    def incrementor():
        nonlocal bonus
        bonus = bonus + 100
```

```
        nonlocal salary
        salary = salary + 200

        nonlocal total
        total = bonus  + salary

        return bonus, salary, total
    return incrementor

a = counter()
print(a())
print(a())
print(a())
```

output
(150, 200, 350)
(250, 400, 650)
(350, 600, 950)
Note: output values are varied because of nonlocal maintain the state
==========

Nonlocal cannot be used in outer fn

Nonlocal cannot be used in outer fn as global scope, it is an error – see below

```
def counter():
    num = 1
    nonlocal num # this is NOT right. Can not be used as global scope

    def incrementor():
        # nonlocal num # this correct – nonlocal is only for inner fn
        num = num +1
        return num

    return incrementor()

print(counter())
```

output
nonlocal num

^

SyntaxError: name 'num' is assigned to before nonlocal declaration

==========

Points to ponder on nonlocal token

The below are Melcose note and you can modify as per your understanding

- nonlocal maintains the state
- if same variable name is used in outer function and inner function, we need to use nonlocal keyword in inner fn
- HO / outer function variable is constant / immutable ie, it will not be modified (it will be stored in different memory location
- Non local is applicable only for nested function
- Nonlocal cannot be used as global scope in outer function...it is an error
- If multiple variables are declared in outer function, then each variable must be declared using nonlocal in inner function, in order to access / process

- Nonlocal used to work with variables inside the nested functions, where the **variable should not belong to the inner function.**

- real time example
- RBI interest is fixed
- local banks can charge its own interest rate by overriding the RBI fixed interest rate via using nonlocal token inside the method

DataScienceMel.py

```
def counter():
    bankOpeningBalance = 1000

    def ATMWithdraw(withdrawalAmount):
        nonlocal bankOpeningBalance
        print('withdraw amount: ', withdrawalAmount)
        bankInitialBalance = bankOpeningBalance - withdrawalAmount

        if bankInitialBalance == 0:
            print('Your balance is 0')
        return bankInitialBalance

    print("Initial bank opening balance",bankOpeningBalance)
    return ATMWithdraw,bankOpeningBalance

withdrawAmount, originalAmount = counter()
print(withdrawAmount(100), originalAmount)
print(withdrawAmount(200), originalAmount)
print(withdrawAmount(300), originalAmount)
```

```
----------
def counter():
    bankOpeningBalance = 1000

    def ATMWithdraw(withdrawalAmount):
        nonlocal bankOpeningBalance
        print('withdraw amount: ', withdrawalAmount)
        bankInitialBalance = bankOpeningBalance - withdrawalAmount

        if bankInitialBalance == 0:
            print('Your balance is 0')
        return  bankInitialBalance

    print("Initial bank opening balance",bankOpeningBalance)
    return ATMWithdraw,bankOpeningBalance

withdrawAmount, originalAmount = counter()
print(withdrawAmount(100), originalAmount)
print(withdrawAmount(200), originalAmount)
print(withdrawAmount(300), originalAmount)
--------------
```

1. Nonlocal can be used only in nested function
2. Each inner fn must **return** its task, else the outer fn can not access the inner fn task
3. The outer fn must return the inner fn (method name only – **without parantheses**)
4. Nonlocal key word will be used only **in inner function** of nested function
5. It can not be used as global scope in outer fn...it is an error
6. Multiple varible can be declared in outer fn, then each variable must be declared using nonlocal in inner fn, inorder to process
7. A outer fn can have multiple inner fn, provided the last inner fn must have return statement (this return will mention all the methods of outer fn – will not be used in increment purpose), result is in tuple format
8. Nonlocal token will maintain the state , ie the previous output will be added in next output

```
a = outer()
print(a()) # this output will be added with next output
print(a())
```

9. Multiple varibales can be assigned to nonlocal token,in this case all variables will be treated as nonlocal (**nonlocal** bonus, salary, total)

10. the functionality presented here is better implemented using generators

which variable has what prirority - inner fn, outer fn and global scope

How inner fn variable , outer fn variable and global scope variable are given preference

```
x = 107 # 1ST PREFERENCE IS GIVEN TO GLOBAL SCOPE VARIABLE (IF local variable is  not available  inside inner fn or outter fn)

def outerFn():
    # x = 100 # 2ND  PREFERENCE IS GIVEN TO OUTER FN VARIABLE (IF AVAILABLE)
    def innerFn():
        # x = "YYY" # 3RD PREFERENCE IS GIVEN TO INNER FN VARIABLE (IF AVAILABLE)
```

```
            print(x)
            return x
        return innerFn()

    outerFn()
```

output (based on the inner fn / outer fn / global scope

```
107
100
YYY
YYY=
===============

x="AAA"
print(x)
def change_local():
   x="BBB"
   print (x)#X BELONG TO METHOD / ACT AS PURE LOCAL VARIABLE, SEE THE TAB INTENTED-Melcose
change_local()
#print (x)
```

output
AAA
BBB
Since Python interpreter language it executes line by line.It prints x=AAA then goes to the method
========

```
x="AAA"
def change_local():
    x="BBB" #(observe, we did not print x)
# Below print statement watchs the indentetation.It is not inside the function
print (x) #X IS OUT OF SCOPE TO THE METHOD / SO IT ACCCES OUTER VALUE OF THE X “AAA”

change_local()
# print (x)
```

OUTPUT
AAA
AAA
========

How scope works outside a class, inside a class, inside a method

Declare the same variable outside the class, inside the class, inside the method – how scope works

```
salary = 100 # global scope variable
class ABC:
    bonus = 1000
    print("Bonus variable is inside of class", bonus)
    print("Salary variable is outside of the class", salary)

    def testMethod(self):
        salary = 2000
        print("able to access the salary variable which is  LOCAL ...inside method", salary)
        print("able to access the variable bonus which is  inside class ...inside method",
ABC.bonus)
        print("able to access the variable bonus which is  inside class ...inside method",
self.bonus)

a1 = ABC()
a1.testMethod()
print("outside of class", salary)
```

output
Bonus variable is inside of class 1000
Salary variable is outside of the class 100
able to access the salary variable which is LOCAL ...inside method 2000
able to access the variable bonus which is inside class ...inside method 1000
able to access the variable bonus which is inside class ...inside method 1000
outside of class 100
==========

Global key word

Declaring a name global means that, for the rest of the scope, any assignments to the name will happen at the module's top level:

Important Rule / Note: If we declare a global variable inside a method, we can access that method variable, in all the methods of the python file

```
def testMethod():
    global salary
    salary = 900
    return salary - 100

print(testMethod()) # just call the method
print(salary)
```

o/p:

800
900

Note : we declare the global token inside the method

==========

Same global value can be used in multiple fns.

Since it maintains the STATE, It takes last modified gloabal value from previous method as long as we use SAME global variable

```
def method1():
    global salary
```

```
    salary = 100
    salary = salary + 10
    return salary

def method2():
    global salary
    print("salary ", salary)

    salary= salary + 200
    # since salary is declared as global, it maintains the STATE,
    # so  it takes last modified salary value  from line4 ie 110 from previous method
    return salary

# print("print the GLOBAL VARIABLE  salary from outside, BEFORE CALLING fn",salary) #
THROWS ERROR

print(" CALLING method1", method1()) # NO  ERROR
print("print the GLOBAL VARIABLE  salary from outside, AFTER CALLING fn",salary)
print("=============")

print(" CALLING method2", method2()) # IT RETURNS METHOD2'S SALARY VALUE
```

output

CALLING method1 110

print the GLOBAL VARIABLE salary from outside, AFTER CALLING fn 110

==========

CALLING method2 310

========== ========

non local variable and gloabal variable can be used with in the nested fn, to avoid, over ride the global varaible's state and we manipulate locally

```
def method1():
    global salary
    salary = 100
    salary = salary + 10
    return salary

def method2():
    global salary # now salary becomes 110
    bonus= salary + 1000 # bonus become 1110

    def incentive():
        nonlocal bonus
        bonus = bonus *10 # bonus become 11100
        return bonus

    return incentive()

print("Call the method 1 ", method1())
print("Call the nested method ", method2())
```

output

Call the method 1 110
Call the nested method 11100

==========

Accessing Global **scope** variable and global variable

```
x="AAA"
def change_local():
    global x
    x="BBB"
    print (x)

change_local()
print (x)
```

output

```
BBB
BBB
```

```
x="AAA"
def change_local():
    global x
```

```
    x="BBB"
    # print (x)

print (x) # before calling the method x is AAA
change_local()
print (x) # After calling method the x is BBB
```

output
AAA
BBB
==========

Global Variables

In Python, variables inside functions are considered local if and only if they appear in the left side of an assignment statement, or some other binding occurrence; otherwise such a binding is looked up in enclosing functions, up to the global scope. This is true even if the assignment statement is never executed.

=========

Inner variable shadows outter variable with in the fn

Normally, an assignment inside a scope will shadow any outer variables of the same name: see below example

```
x = 'Hi'
def change_global_x():
    x = "Bye"
    print(x)

change_global_x()
```

output
Bye

==========

Overrding Global SCOPE using global variable

```
x = 'Hi'
def change_global_x():
    global x
    x = "Bye"
    print(x)

print(x) # Bfore calling the fn
change_global_x()
print(x) # now the global modifies the Hi to Bye
```

output
Hi
Bye
Bye
=======

Using nonlocal and global together in a nested fn to see how it maintains the state

```
x = 'Hi'
```

```
def outer():
    age = 10

    def inner1():
        nonlocal age
        age= age + 10

        global mark
        mark = 100
        return age # try this too return age,mark

    def inner2():
        global mark
        mark = mark + 111  # Global varaibale maintian the state
        return  mark

    return inner1(),inner2()

print(outer())
```

output

(20, 211)

Note:
global is declared only inside fn (not inside the class)
Using global , we can modify the orginal value declared in outer scope

Using global, we directly access the method variable , only after the calling the method

=======

The global keyword means that assignments will happen at the module's top level, not at the program's top level. Other modules will still need the usual dotted access to variables within the module.

Declaring a name global means that, for the rest of the scope, any assignments to the name will happen at the module's top level:

To summarize: in order to know whether a variable x is local to a function, you should read the entire function:

1.if you've found global x, then x is a global variable
2. If you've found nonlocal x, then x belongs to an enclosing function (outer fn), and is neither local nor global
3. If you've found x = 5 or **for x in range** or some other binding, then x is a local variable 5. Otherwise x belongs to some enclosing scope (function scope, global scope, or builtins)

Basically nonlocal will allow you, to assign to variables in an outer scope, but not a global scope.

Note that the functionality presented here is better implemented using generators.
==========

Local Variables

If a name is bound inside a function, it is by default accessible only within the function:

```
def foo():
    a = 5
    print(a)

foo()

print(a)
```

output

```
5
Traceback (most recent call last):
  File "C:\Users\Melcose\PycharmProjects\PythonDSIT\DataScienceInTamil.py", line 6, in
<module>
    print(a)
```

NameError: name 'a' is not defined

==========

Control flow constructs have no impact on the scope (with the exception of except), but accessing variable that was not assigned yet is an error:

```
def foo():
    if True:
        a = 5
        print(a)

foo()
```

output

5

==========

```
b=3
def foo():
    if False:
        b = 5
        print(b)

foo()
```

output

check the output and discuss ..

==========

```
sal = 100
def M():
    if False:
        sal = 101
    print(sal)

print(M())
```

output is error

```
    print(sal)
```

UnboundLocalError: local variable 'sal' referenced before assignmentCommon binding

Note: we try to reassign the value to the variable sal from 100 to 101 inside the fn. But inside the fn, we have False in ‘if’ Statement. So, execution control flow will NOT go into False. That is why we get error. To avoid , inside the ‘if’ statement give True

---

Global scope variable can be accessed in side a fn’

```
‘sal = 100
def M():
    return sal

print(M())
```

Ouput
100
====

Global keyword can override ‘if..false’

Global keyword can override ‘if..false’ statement and able to access outer scope variable
Even if the ‘if’ statement has False, if we want the control goes into ‘if’ statement, give ‘global’ token, this global token will access outside variable (sal) value – see below

```
sal = 100
def M():
    if False:
        global sal # even if the assignment statement is never executed, global keyword tells the interpreter that ‘sal’ is referring to glable scope variable
        sal = 101
```

```
    print(sal)

print(M())
```

output
100

Note: since we declare 'global' inside the 'If ...False', still it is possible to access the outerscope / global scope variable

There are 2 tasks are done by interpretor

1. Read the code
2. Execute the code

Above we have If false, here the interpreter read the if and false but not EXECUTING(reason is FALSE)

==========

Global keyword can override 'if..false', but can not modify the value

Global keyword can override 'if..false' statement and able to ACESS outerscope variable – BUT NOT MODIFY THE OUTERSCOPE VARIABLE in IF FALSE

Now, since we declare 'global' inside the 'If ...False', still it is able to access the outerscope variable, BUT WE **CAN NOT REASSIGN** A NEW VALUE TO OUTERSCOPE VARIABLE(VAL) USING GLOBAL KEYWORD

```
sal = 100
def M():
    if False:
        global sal
        sal = 1000 # trying to reassign new val to varible (but not possible)
        sal = sal + 500 # modifying the global varible INSIDE the IF FALSE  is NOT POSSIBLE
    sal = sal + 111 # modifying the global varible INSIDE the FN is POSSIBLE
    print(sal)

print(M())
print (sal)
```

output

211

None

211

If the GLOBAL was given, inside if ---TRUE block, then the global scope varible could be reassigned – see the below code

```
sal = 100
def M():
    if True:
        global sal
        sal = 1000 # trying to REASSIGN new val to varible (is possible)
        sal = 10111 # trying to REASSIGN new val to varible (is possible)
    print(sal)

print(M())
```

output
10111

Note1:If we declare outer scope variable as nonlocal inside the function, the variable can be reassigned / writable.

Note 2: if we use the global variable inside the if...False we can override the if..else statement(control still can go inside if..False), But we cannot reassign.

=======

Non local can not be used in side IF FALSE

```
def outer():
    num = 100
    def inner():
        if False:
            nonlocal num
            num = num + 10
        return num
    return inner

a = outer()
print(a())
```

o/P:
100

Note: if True , the output will be 110.
======

Non local is declared before the IF FALSE condition and it works as normal if else loop

```
def outer():
    num = 100
```

```
    def inner():
        nonlocal num
        if False:
            num = num + 10
        else:
            num = num -10
        return num

    return inner

a = outer()
print(a())
```

output
90

operations are assignments, for loops, and augmented assignments such as a += 5
==========

Below is for deep discussion inorder to understand the scope

```
x = 10
y = 20

def outer():
    z = 30
    def inner():
        global x
        x = x + 30
        print(f'x is {x}') # from global scope variable is modified
        print(f'y is {y}') # from global variable is accessed as it is
        print(f'z is {z}') # from outer fn

    inner()
outer()

print("==========")
print(f'x is {x}')
print(f'y is {y}')
# print(f'z is {z}') # not able to access as it bellongs to outter fn
```

ouput

```
x is 40

y is 20

z is 30

==========
```

x is 40

y is 20

==========

The del command in variables

This command has several related yet distinct forms.
del v
If v is a variable, the command del v removes the variable from its scope. For example:

```
v = 5
print(v) # output 5
del(v)
print(v) # this is error
```

output

```
    print(v)
NameError: name 'v' is not defined
5
```

==========

Note that del is a binding occurrence, which means that unless explicitly stated otherwise (using nonlocal or global), del v will make **v local to the current scope.** If you intend to delete v in an outer scope, use nonlocal v or global v in the same scope of the del v statement.

We can not del a variable inside the nested loop with out globale or nonlocal declaration

```
def outer():
    num = 100
    age = 34
    def inner():
        del num # num can not be deleted as num does not belong the inner()
        return num

    return inner

a = outer()
print(a())
```

output is error

```
    del num
UnboundLocalError: local variable 'num' referenced before assignment
```

---

del will work if we declare as non local

```
def outer():
    num = 100
    age = 34
    def inner():
        nonlocal num
        del num # num varibale CAN BE delete, cos it became non local
        return num # hence we dont have num variable we can not return

    return inner()

a = outer()
print(a())
```

output is an error

```
    return num
NameError: free variable 'num' referenced before assignment in enclosing scope.
Did you mean: 'sum'
```

Use global to apply del (delete) outer scope

```
sal = 100
def delMethod():
    global sal
    sal = 200

    del sal          # del is used to remove the global outer scope
     #print(sal)            # not possible
     # return sal       # not possible

a = delMethod()
print(a)
print(sal)
```

output is error

```
    return sal
NameError: name 'sal' is not defined
----------
```

Use nonlocal to del (delete) outer scope

```
def delOuter():
    sal = 100

    def delInner():
        nonlocal sal
        del sal       # del is used to remove the nonlocal outer scope
        # sal = sal + 100  #not possible
        return sal    # del is used to remove the global outer scope

    return delInner

a = delOuter()
print(a())
```

output is error

sal = sal + 100

NameError: free variable 'sal' referenced before assignment in enclosing scope

---

Non local variable can be deleted and re cretated and the newly created variable act as new non local variable – inside the fn

```
def delOuter():
    sal = 100
    def delInner():
        nonlocal sal
        print("ID of sal BEFORE del ", id(sal))
        del sal       # del is used to remove the nonlocal outer scope

        # print("ID of sal AFTER  del ", id(sal)) # IT THROWS ERROR, COS SAL VARIABLE DELTED SO NO ID / MEMORY LOCATION

        sal = 1000 # here again non local sal is reassinged with new value 1000
        print("ID of sal AFTER REASSIGN del ", id(sal))
        return sal     # del is used to remove the global outer scope

    def inner2():
        nonlocal sal
        return sal
    return delInner, inner2

a, b = delOuter()
print(a())
print(b())
```

output
ID of sal BEFORE del 2515655265616
ID of sal AFTER REASSIGN del 2515656300816

1000
1000

How to use del in class scope variables
Mel: Note, teach this after the 'class concept)

Teach below section after 'class' concepts
Teach below section after 'List comprehension concepts
Teach below section after 'Set comprehension concepts
Teach below section after 'Dict comprehension concepts
Teach below section after 'tuple comprehension / generator expression concepts
Teach below section after 'lambda ' / anonymous function
Teach below section after 'lambda globals(), locals()

example of how to delete a class variable (is possible using delattr()

```
class ABC():
    marks = 90
    age = 20

    def method1(self):
        print("before del ,  marks ", self.marks)
        print("class variable  marks  before deleting", self.marks)
        delattr(ABC, 'marks')
        return  " "

    def method2(self):
        print(" from method Global scope name AFTER deleting ", self.age)

a = ABC()
a.method1()
a.method2()
```

==========

example of how to delete a class' outer scope variable (it is possible using global token)

```
name = "Linda"
class ABC():
    marks = 90
    age = 20

    def method1(self):
        global name
        name = "Mel"
        print("before del the marks ", self.marks)
        print("Global scope name before deleting", name)
        del name
        return  " "

    def method2(self):
        global name
        name = "Cose"
        print(" from method Global scope name AFTER deleting ", name)


print(name) # we can acess this global scope variable 'name' before calling method1()

a = ABC()
a.method1()
# print(name) # we NOT  acess this global scope variable 'name' before calling method1

a.method2()
print(name)
```

output

Linda
before del the marks 90
Global scope name before deleting Mel
 from method Global scope name AFTER deleting Cose
Cose

==========

Functions skips class' scope when looking up name – teach afte the 'class'

Classes have a local scope during definition, but functions inside the class do not use that scope when looking up names. Because lambdas are functions, and comprehensions are implemented using function scope, this can lead to some surprising behavior

```
a = 'global'

class Fred:
    a = 'class'  # class scope
    b = (a for i in range(10))  # function scope
    c = [a for i in range(10)]  # function scope
    d = a  # class scope
    e = lambda: a  # function scope
    f = lambda a=a: a  # default argument uses class scope

    @staticmethod  # or @classmethod, or regular instance method
    def g():  # function scope
        return a

print(Fred.a)  # class
print(next(Fred.b))  # global
print(Fred.c[0])  # class in Python 2, global in Python 3
print(Fred.d)  # class
print(Fred.e())  # global
print(Fred.f())  # class
print(Fred.g()) # global
```

Users unfamiliar with how this scope works might expect `b`, `c`, and `e` to print `class`.

```
a= 'global scope'
class Fred:
    a='class level scope'
    b= (a for i in range(10))
    c= [a for i in range(10)]
    d = a
    e = lambda :a
    f= lambda a = a: a

    @staticmethod
    def g():
        return a

print (Fred.a)
print (next(Fred.b))
print (Fred.c[0]) # print(Fred.c[0)

print (Fred.d)
print (Fred.e())
print (Fred.f())
print (Fred.g())
```

output
class level scope
global scope
global scope
class level scope
global scope
class level scope
global scope

Names in class scope are not accessible. Names are resolved in the innermost enclosing function scope. If a class definition occurs in a chain of nested scopes, the resolution process skips class definitions.

The scope of names defined in a class block is limited to the class block; **it does not extend to the code blocks of methods** – this includes comprehensions, generator expressions , lamda and methods, since they are implemented using a function scope. This means that the following will fail:

```
# a =  43 #this global scope can be accessed by genereator expression
class ABC:
    a = 42 #this class scope can NOT be accessed by genereator expression
```

```
    b = list(i+a for i in range (10))
    print(b)
# print(next(b))
```

output
NameError: name 'a' is not defined

Note: i+ABC.a for i in range (10)) also will fail. The reason in a=42 is declared under the class ABC. If a fn wants to access a class variable, it can be done in 2 ways
(1.classname.valriable / ABC.a.
(2 self.variable name / self.a)

But comprehension is NOT a fn. So it can not access the class varible
Generator expression also not fn. So it can not access the class varible

Note: ie **comprehension** and **generator expressins** are functions scope but these are NOT typical functions / methods. That is why these 2 are NOT able to access class' variable using className.varaible or self.variable but able to access global scope varible (ie the variable declared before class / top level)
==========

Local vs Global Scope - teach after the class

What are local and global scope?
All Python variables which are accessible at some point in code are either in local scope or in global scope.
The explanation is that local scope includes all variables defined in the current function and global scope includes variables defined outside of the current function/outside of the class.

How To find the global scope and local scope variables

Note: Globals() gives all the default and user defined global SCOPE variables in a given fn

(outside of the current function/outside of the class) variables use
globals().key()
globals().values()
globals().items()

locals().key()
locals().values()
locals().items()

see the pgm

```
name, age = "Murugan", 19
class Student():
    regNo = 5652
    grade = "First"

    def testFunc(self, mark):
        year = 2022
        print("Marks is ", mark)
        print("==========")
        print("globals.keys gives outter class'globale scope variables' KEYS", globals().keys())
        print("globals.values gives outter class'globale scope variables'
VALUES",globals().values())
        print("globals.items gives outter class'globale scope variables' KEYS +
VALUES",globals().items())
        print("============")

        print(locals().keys())
        print(locals().values())
        print(locals().items())
        return ""

obj = Student()
result = obj.testFunc(100)
print(result)
```

output

Marks is 100
==========
globals.keys gives outter class'globale scope variables' KEYS
dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__annotations__', '__builtins__', '__file__', '__cached__', 'name', 'age', 'Student', 'obj'])

globals.values gives outter class'globale scope variables' VALUES
dict_values(['__main__', None, None, <_frozen_importlib_external.SourceFileLoader object at 0x00000191C7E307F0>, None, {}, <module 'builtins' (built-in)>, 'C:\\Users\\Melcose\\PycharmProjects\\pythonProject\\Sudha_Linda.py', None, 'Murugan', 19, <class '__main__.Student'>, <__main__.Student object at 0x00000191C7ED8430>])

globals.items gives outter class'globale scope variables' KEYS + VALUES
dict_items([('__name__', '__main__'), ('__doc__', None), ('__package__', None), ('__loader__', <_frozen_importlib_external.SourceFileLoader object at 0x00000191C7E307F0>), ('__spec__', None), ('__annotations__', {}),

('__builtins__', <module 'builtins' (built-in)>), ('__file__', 'C:\\Users\\Melcose\\PycharmProjects\\pythonProject\\Sudha_Linda.py'), ('__cached__', None), ('name', 'Murugan'), ('age', 19), ('Student', <class '__main__.Student'>), ('obj', <__main__.Student object at 0x00000191C7ED8430>)])
==========
dict_keys(['self', 'mark', 'year'])
dict_values([<__main__.Student object at 0x00000191C7ED8430>, 100, 2022])
dict_items([('self', <__main__.Student object at 0x00000191C7ED8430>), ('mark', 100), ('year', 2022)])
==========

How to access class varaibles – teach after the class

(using vars(className))

```
class Student():
    regNo = 5652
    grade = "First"

obj = Student()
print(vars(Student))
```

output

{'__module__': '__main__', 'regNo': 5652, 'grade': 'First', '__dict__': <attribute '__dict__' of 'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' objects>, '__doc__': None}

```
foo = 1  # global

def func():
    bar = 2  # local
    print(foo)  # prints variable foo from global scope
    print(bar)  # prints variable bar from local scope
```

One can inspect which variables are in which scope. Built-in functions `locals()` and `globals()` return the whole scopes as dictionaries.

```
foo = 1

def func():
    bar = 2
    print(globals().keys())  # prints all variable names in global scope
    print(locals().keys())  # prints all variable names in local scope
```

==========

What happens with name clashes?

(# global scope variable name and local variable name both have same name)

==========

Preference is given to pure local – by default

```
foo =1
def func():
    foo = 2
    print(foo) #prints 2
    print(foo) #print 2

func()
```

If want access pure local and global scope use local() and global()

```
foo =1
def func():
    foo = 2
    print(locals()['foo']) #prints 2
    print(globals()['foo']) #print 1

func()
```

==========

another ex

```
foo =1
def func():
    foo = 2
    print(foo)

print(foo) # before fn calls

func()
print(foo) # after  fn calls
```

output

1

2

1

==========

```
foo =1
def func():
    foo = 2
    print(foo)
```

```
    print(locals()) # it takes only inside fn varaible
print(globals()) # it takes global scope variable)

func()
```

output

{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x0000028349B2E7C8>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'C:/Users/ABC/PycharmProjects/pythonProject/kkkk.py', '__cached__': None, 'foo': 1, 'func': <function func at 0x0000028349B9FF78>}

2

{'foo': 2}

```
foo =1
def func():
    global foo
    foo = 10
    name= “ABC”
    print(foo) # prints 10
```

```
    print(locals().items()) # it is empty, since the fn does not have any LOCAL variable
    print(globals()['foo']) # 1 from global scope has been modifed in side this fn so gives 10
    print(locals()['name']) # gives the value of the lcal variable


func()
```

output

```
10
dict_items([])
10
ABC

==========
```

How to find local variables for normal and nested fns

```
salary = 100 # this variable is global SCOPE (not local) CAN NOT  be part of locals()
BONUS = 50 # # this variable is global SCOPE (not local)  CAN NOT be part of locals()
```

```
def testLocal():
    global OVERTIME
    OVERTIME = 30.55 # this is global variable, so it will NOT be part of locals ()
    name = "AAA" ## this is fn's local variable so it  will be PART of locals ()
    salary = 45000.50

    print("locals() gives the ITEMS  in DICT format")
    print(locals())

def outer():
    bonus = 300
    festivalLoan = 200
    localVariableForOuter = 1000

    def inner():
        # nonlocal bonus # if we use nonlocal, it becomes LOCAL variable to inner()
        # nonlocal localVariableForOuter
        name = "Tamil"
        initial = "is ours"
        print("Local variables from LOCAL inner fn", locals())  # it shows local variables of
inner()
        return ''
    print("These are local variables from OUTER fn", locals()) # it shows local variables of
outer()

    return inner()

    print("locals() gives the ITEMS  in DICT format")
    print(locals())
```

```
outer()
print("=" * 40)
testLocal()
```

output

```
These are local variables from OUTER fn {'bonus': 300, 'festivalLoan': 200, 'localVariableForOuter': 1000, 'inner': <function outer.<locals>.inner at 0x000001B0D42FC670>}
Local variables from LOCAL inner fn {'name': 'Lara', 'initial': 'M'}
========================================
locals() gives the ITEMS  in DICT format
{'name': 'AAA', 'salary': 45000.5}
====================
```

Functions within functions

There may be many levels of functions nested within functions, but within any one function there is only one local scope for that function and the global scope. There are no intermediate scopes

Fn inside fn insisde another fn – ex

Notes: If you want **fn1() result ,** below is the code.

```
bonus = 1000
def fn3():
    sal =  100
    # print("inside fn3", sal)

    def fn2():
        sal = 200
        # print("inside fn2", sal)

        def fn1():
            sal = 300
            # print("inside fn1", sal)
            return sal

        return fn1()      # observe the indentation of return

    return  fn2()       #observe the indentation of return

a = fn3
print(a())
```

output

300

Fn inside fn insisde another fn – ex

Notes: If you **want fn2()** result , below is the code.

```
bonus = 1000
def fn3():
    sal =  100
    # print("inside fn3", sal)

    def fn2():
        sal = 200
        # print("inside fn2", sal)


        def fn1():
            sal = 300
            # print("inside fn1", sal)
            return sal
        return sal      # fn2 returns sal, now the results comes from fn2
    return  fn2()       #observe the indentation of return

a = fn3()
print(a)
```

o/p:

200

Fn inside fn insisde another fn – ex

Notes: If you want fn3() result , below is the code.

```
bonus = 1000
def fn3():
    sal =  100
    # print("inside fn3", sal)

    def fn2():
        sal = 200
        # print("inside fn2", sal)


        def fn1():
            sal = 300
            # print("inside fn1", sal)
            return sal
        return sal      # observe the indentation of return
    return  sal       #observe the indentation of return

a = fn3()
print(a)
```

o/p:

100

---

global vs nonlocal (Python 3 and above only)

Both these keywords are used to gain **write access (re-assign)** to variables which are not local to the current functions.

The global keyword declares that a name should be treated as a global variable

Example:

```
foo = 0 # Gloable scope

def f1():
    foo = 1 # a new local foo for fn1


    def f2():
        foo = 2 # a new local foo for fn2
```

```
        def f3():
            foo = 3 # a new local foo for fn3
            # print(foo) # ans 3
            foo = 30 # modifies the local foo of fn3 only
            return foo

        def f4():
            global foo
            # print(foo) # ans 0
            foo = 100 # reassign the global scope foo to 100
            return foo
        return foo
    return f2()

a = f1
print(a())
```

output

2

Notes: Based on which function result we want, modify the return statement at the end of the program(before calling the method).

=======

Another example:

```
foo = 5
def f1():
    foo = 1
    print("fn1 foo", foo)

    def f2():
        foo = 2
        print("fn2 foo", foo)

        def fn3():
            global foo
            foo = 10
            print("fn3 foo", foo)

            def fn4():
                foo = 4
                print("fn4 foo", foo)
                # return foo #
            return fn4()
```

```
        return fn3()
    return f2()
print("The global scope value before calling the fn ", foo)
f1()
print("The global scope value AFTER calling the fn ", foo)
```

output

```
The global scope value before calling the fn  5
fn1 foo 1
fn2 foo 2
fn3 foo 10
fn4 foo 4
The global scope value AFTER calling the fn  10
```

==========

Nonlocal takes a local variable from an enclosing scope (outer Scope)

On the other hand, nonlocal (see Nonlocal Variables), available in Python 3, takes a local variable from an enclosing scope (outer Scope) into the local scope of current function.

The nonlocal statement causes the listed identifiers to refer to **previously bound variables in the nearest enclosing scope** excluding globals.

==========

```
def f1():

    def f2():
        foo = 2

        def f3():
            nonlocal foo # foo from f2, which is the nearest enclosing scope
            print("Accessing the foo, before assiging new value to foo", foo)
            foo = 20 # modifies the foo from f2()
            print("Accessing the foo, after  assigned new value to foo", foo)
        return f3()
    return f2()

print(f1())
```

output

```
Accessing the foo, before assiging new value to foo 2
Accessing the foo, after  assigned new value to foo 20
None
==========
```

Where and how binding occurrences for variables

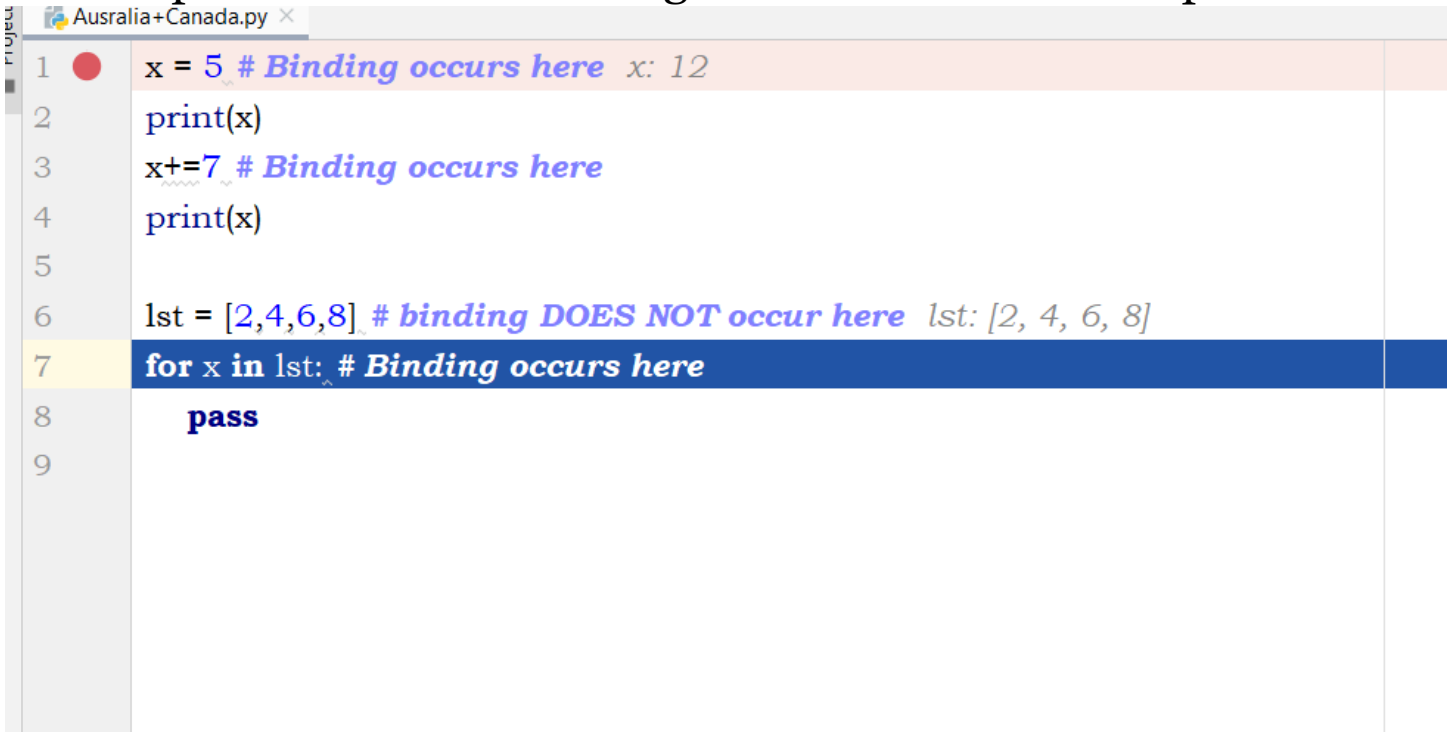
```
x = 5 # Binding occurs here
print(x)
x+=7 # Binding occurs here
print(x)

lst = [2,4,6,8] # binding  DOES NOT occurs here
for x in lst: # Binding occurs here
    pass

==========
```

Each of the above statements is a binding occurrence - x become bound to the object denoted by 5. If this statement appears inside a function, then x will be function's -local by default. See the "Syntax" section for a list of binding statements.

How to prove where the binding occurance – use break point



==========

The end