INTRODUCTION TO PROGRAMMING USING PYTHON

Outline

- Control Flow
- List
- Tuple
- Dict
- Function
- Function
- Default parameters
- Unpacking & packing parameters
- Recursion
- Scope
- Lambda
- Exercises

If Statement

level1

```
if (x == 2):
    print("Two")
elif (x == 3):
    print("Three")
else:
    print("others")
```

level2

for …in

```
statment ='hello'
for ch in statment:
    print(ch)
```

```
h
e
l
l
o
```

Range Function

```
range([start,] end[, step])
```

Examples

```
range(5)                [0,1,2,3,4]

range(0,5,1)            [0,1,2,3,4]

range(1,10,2)           [1,3,5,7,9]
```

```
for i in range(10):
    print(i)
```

0	1	2	3	4	5	6	7	8	9

while

```
dayCount = 0
while dayCount < 4:
        print("We are learning Python")
        dayCount += 1
```

```
Output:
We are learning Python
We are learning Python
We are learning Python
We are learning Python
```

DayCount
1
2
3
4

Break Statement

```
for i in range(10):
    if (i == 5):
        break
    print(i)
```

Continue Statement

```
for i in range(10):
    if (i == 5):
        continue
    print(i)
```

Else Statement

```
for i in range(10):
    if (i == 5):
        continue
    print(i)
else:
    print(10)
```

0 1 2 3 4 6 7 8 9 10

Pass Statement

```
for i in range(10):
    if (i == 5):
        pass
    print(i)
```

input Function

```
input(prompt_message)
```

Example

```
name = input("What's your Name? ");
print(name);
```

```
Output:

What's your name? Mahmoud
Mahmoud
```

lists

```
l=[]
#To define a list:
l=[1,1.1,'one',True,[],(),{}]
print(len(l))
print(l[0],l[6])
print(l[1:3])
print(l[7])
```

```
7
1 {}
[1.1, 'one']
```

```
    print(l[7])
          ~^^^
IndexError: list index out of range
```

lists

```
fruits = ['orange', 'apple']
fruits.sort()  # sort ascending
print(fruits)
```

```
['apple', 'orange']
```

```
fruits.reverse()  # sort descending
print(fruits)
```

```
['orange', 'apple']
```

Lists

```
myList = ["C", "JavaScript", "Python", "Java", "php"];
```


```
myList.pop(4)
```

Lists

```
myList = ["C", "JavaScript", "Python", "Java", "php"];
```


```
myList.pop(4)

myList.append("go")
```

Lists

```
myList = ["C", "JavaScript", "Python", "Java", "php"];
```


```
myList.pop(4)

myList.append("go")

myList.insert(3, 'Scala')
```

Lists

```
myList = ["C", "JavaScript", "Python", "Java", "php"];
```


```
myList.pop(4)

myList.append("go")

myList.insert(3, 'Scala')

myList.remove("C")
```

Lists

```
myList = ["C", "JavaScript", "Python", "Java", "php"];
```


```
myList.pop(4)

myList.append("go")

myList.insert(3, 'Scala')

myList.remove("C")

yourList = ["Ruby", "Rust"];

myList.extend(yourList)
```

Tuples

Same as Lists but Tuples are immutable

```
newTuple = ()
```

```
t = (1, "hi", True)
t[1]
# hi
t[1] = 4
TypeError: 'tuple' object does not support item assignment
```

Dictionaries

- A dictionary: key: value comma separated elements data structure
- doesn't allow duplicates for keys

```
d={} #empty dict
#Key:value
traineedata={'ID':1,
             'Name':'aya ali',
             'branch':'smart'}
print(traineedata)
print(traineedata['ID'])
traineedata['branch']='Alex'
print(traineedata['branch'])
```

```
{'ID': 1, 'Name': 'aya ali', 'branch': 'smart'}
1
Alex
```

Dictionaries

```
infoDict = {'track': 'OS', 'name': 'Ahmed', 'age': 17}

infoDict.keys() # dict_keys(['track', 'name', 'age'])

'name' in infoDict # True

infoDict.items()

# dict_items([('track', 'OS'), ('name', 'Ahmed'), ('age', 17)])

addInfoDict = {'track': 'SD', 'branch': "Smart"}

infoDict.update(addInfoDict)

#{'track': 'SD', 'name': 'Ahmed', 'age': 17 , 'branch': "Smart"}
```

Functions

- Is reusable block of code
- Run when you call it
- Accept parameters to deal with it
- Can return data after job finished
- Types of functions

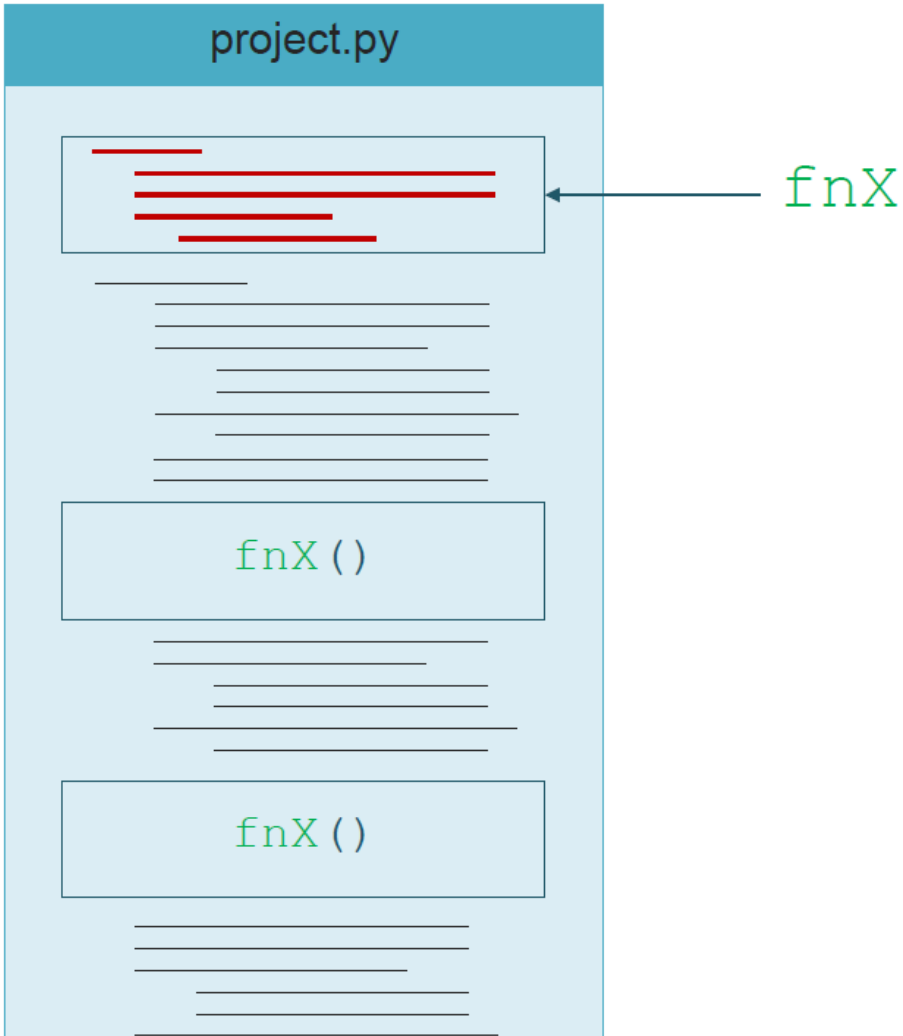
return

Built in

User defined

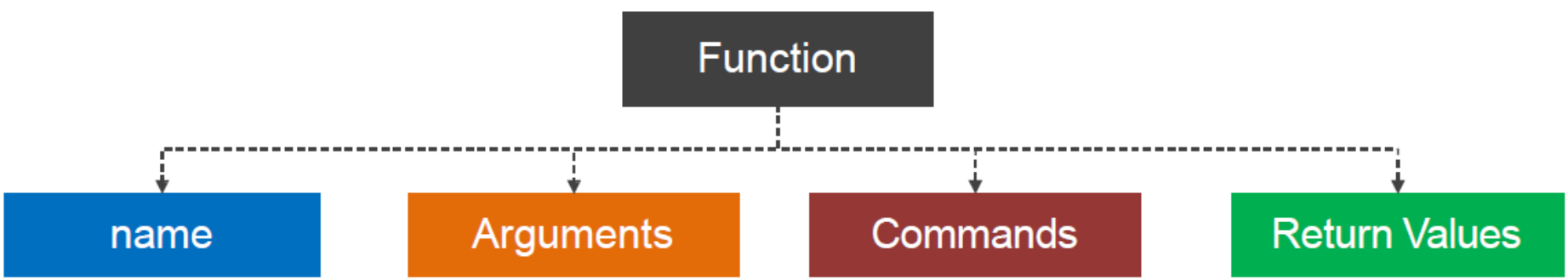
Make your code more generic

Functions



Function (Defining)

```
def fnName:
    pass
```




```
def measureTemp ( temp ):
    if temp < 37:
        return "Too Cold"
    elif temp > 37:
        return "Too Hot"
    return "Normal"
```

Defining example

Calling

```
measureTemp(37)
# "Normal"
```

Function (Default Arguments)

```
def doSum(x, y = 2,z = 3):
    sum = x + y + z
    print(sum)
```

Calling It

```
doSum(2)          # output: 7
doSum(2,4)        # output: 9
doSum(2,4,10)     # output: 16
```

Function (*arguments)

```
def doSum(*args):
        sum = 0
        for i in args:
                sum += i;
        print(sum)
```

Calling It

```
doSum(2,6)          # output: 8

doSum(2,4,5,15)     # output: 26
```

Function (**keywords)

```
def doSum(**kwargs):
    for k in kwargs:
        print(kwargs[k])
```

Calling It

```
doSum(x = 2, y = 26)     # output: 2
                                  26
```

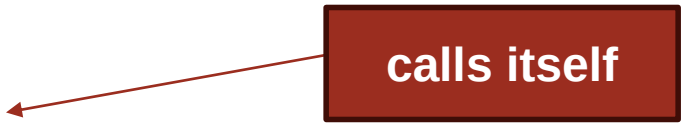
Recursion

- function calls itself in order to solve a problem

2 usages

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

num = 5
result = factorial(num)
print(f"The factorial of {num} is {result}")
```

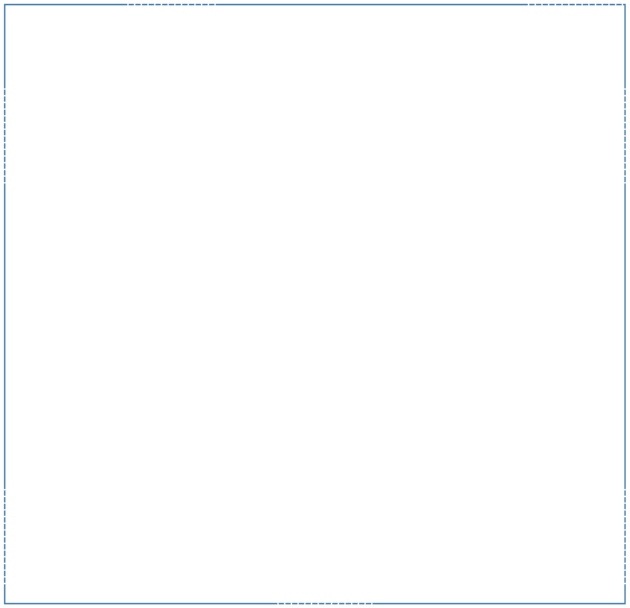



```
The factorial of 5 is 120
```

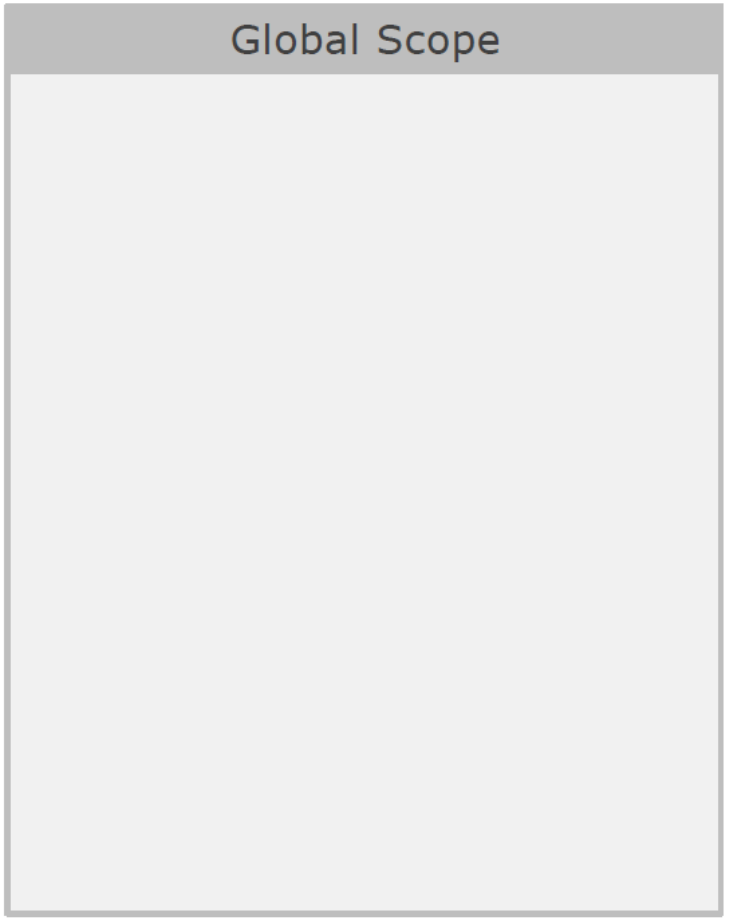
Scope

To know your limits

Lexical Scope



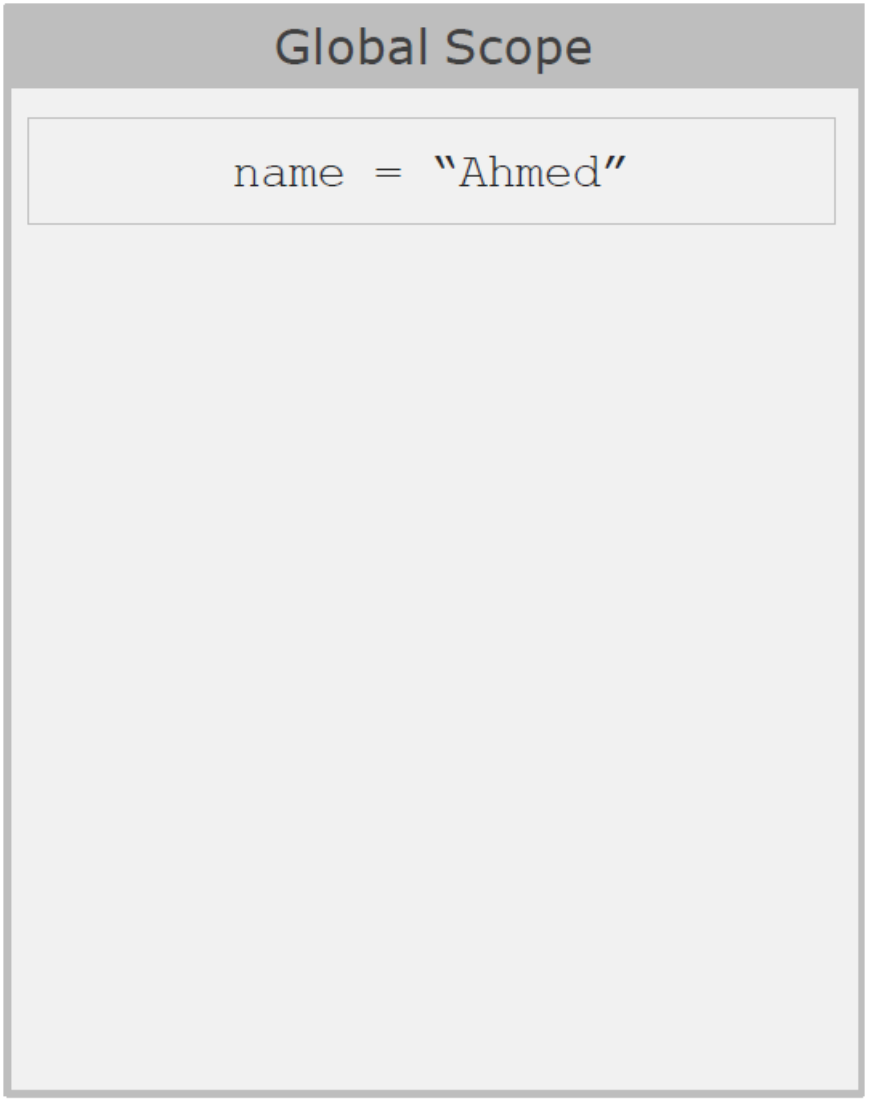
```
Output:
```



Lexical Scope

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
                print(name)
        innerFn()
```

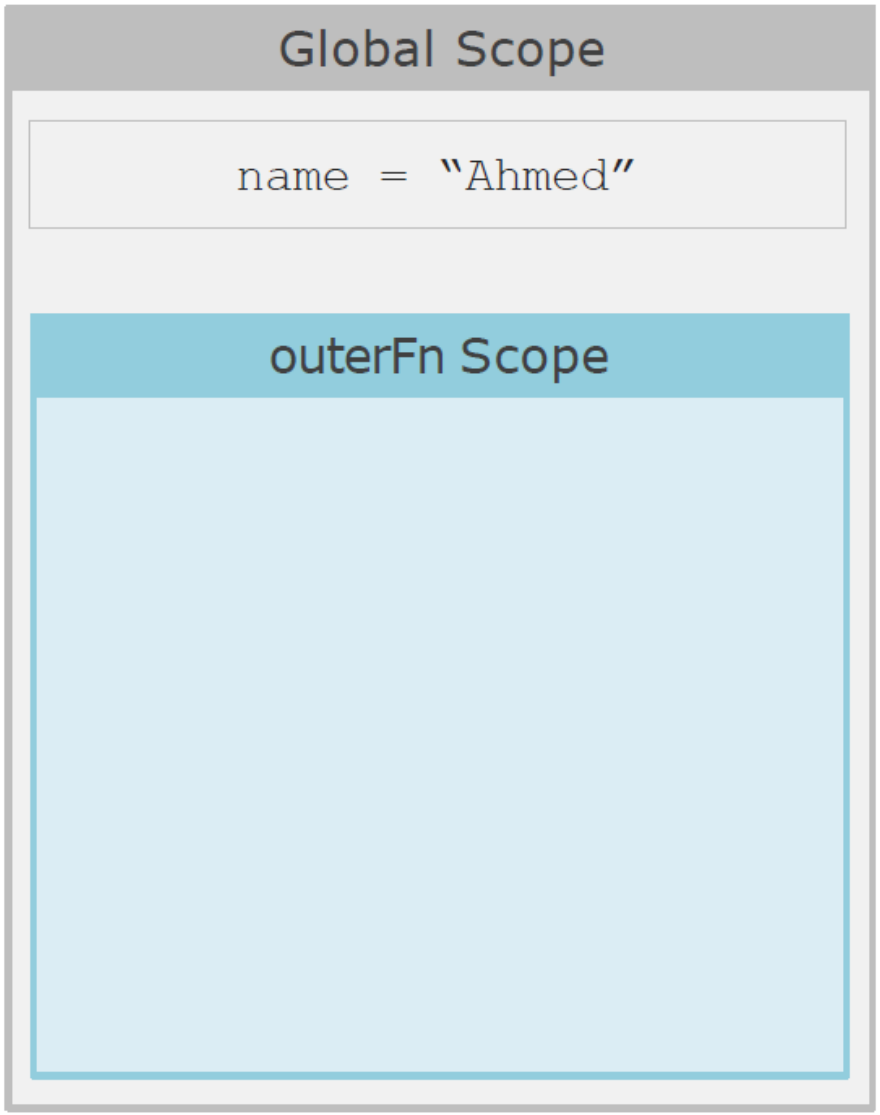
Output:



Lexical Scope

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
                print(name)
        innerFn()
outerFn()
```

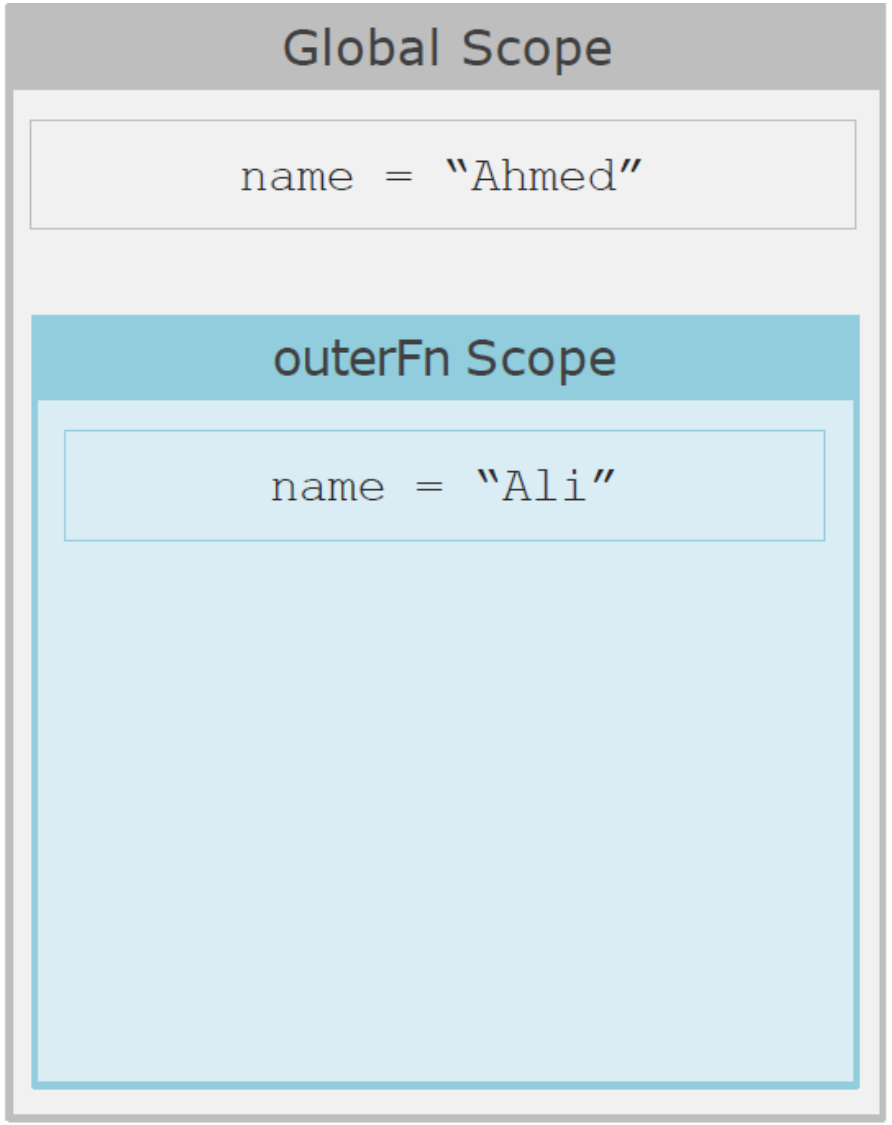
Output:



Lexical Scope

```
name = "Ahmed"
def outerFn():
    name = "Ali"
    def innerFn():
        print(name)
    innerFn()
outerFn()
```

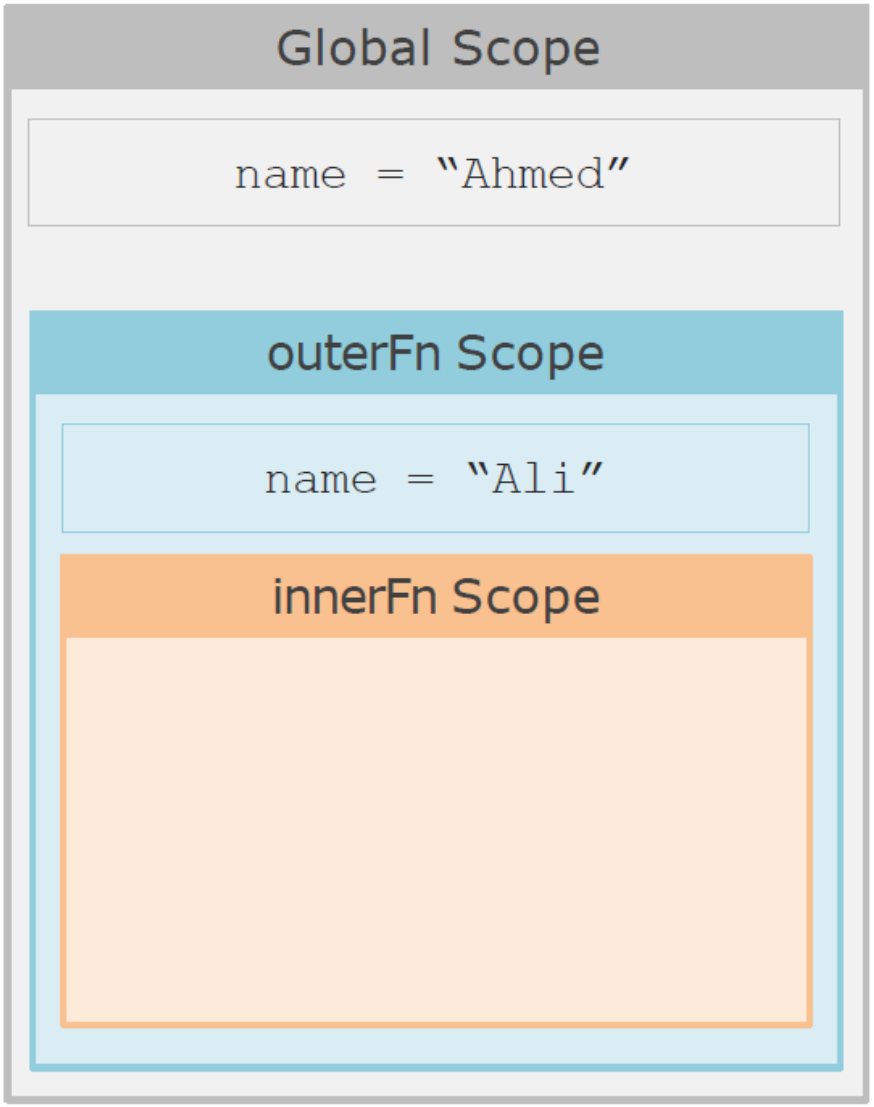
Output:



Lexical Scope

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
    →           print(name)
        innerFn()
outerFn()
```

Output:

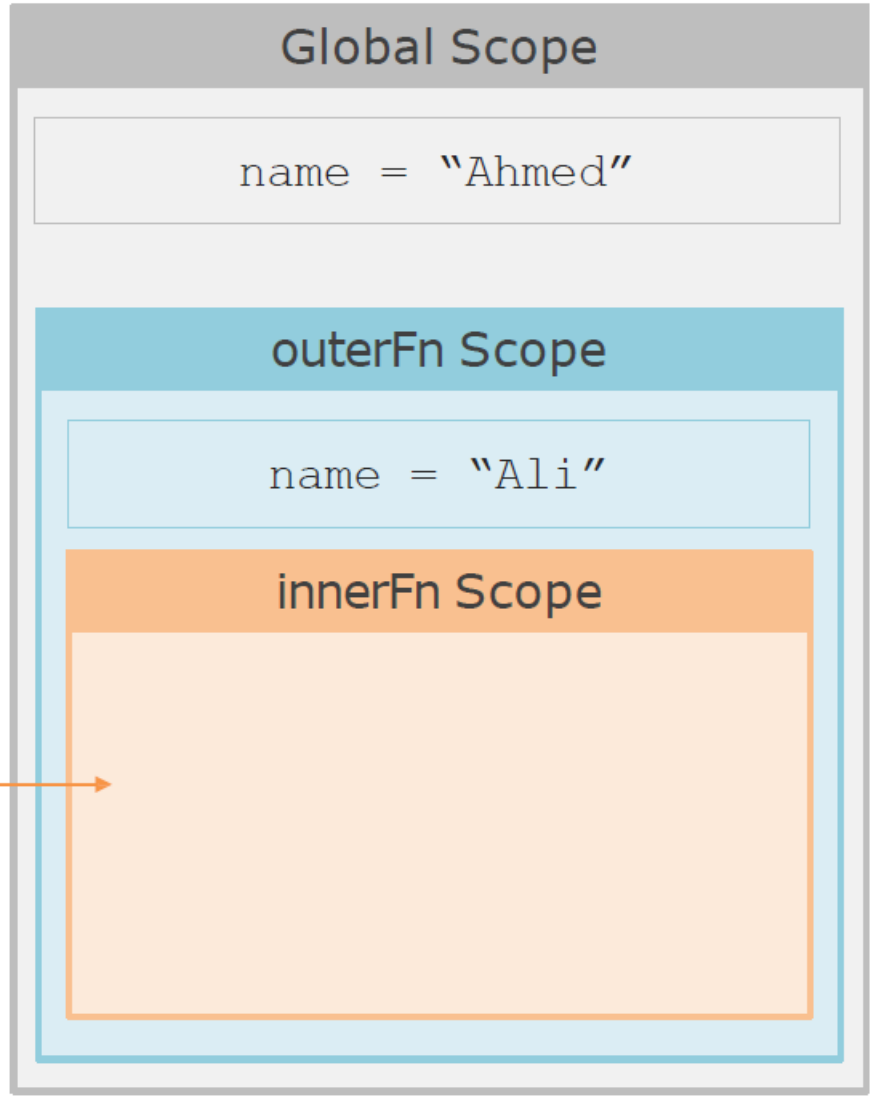


Lexical Scope

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
            print(name)
        innerFn()
outerFn()
```

Output:

name
???

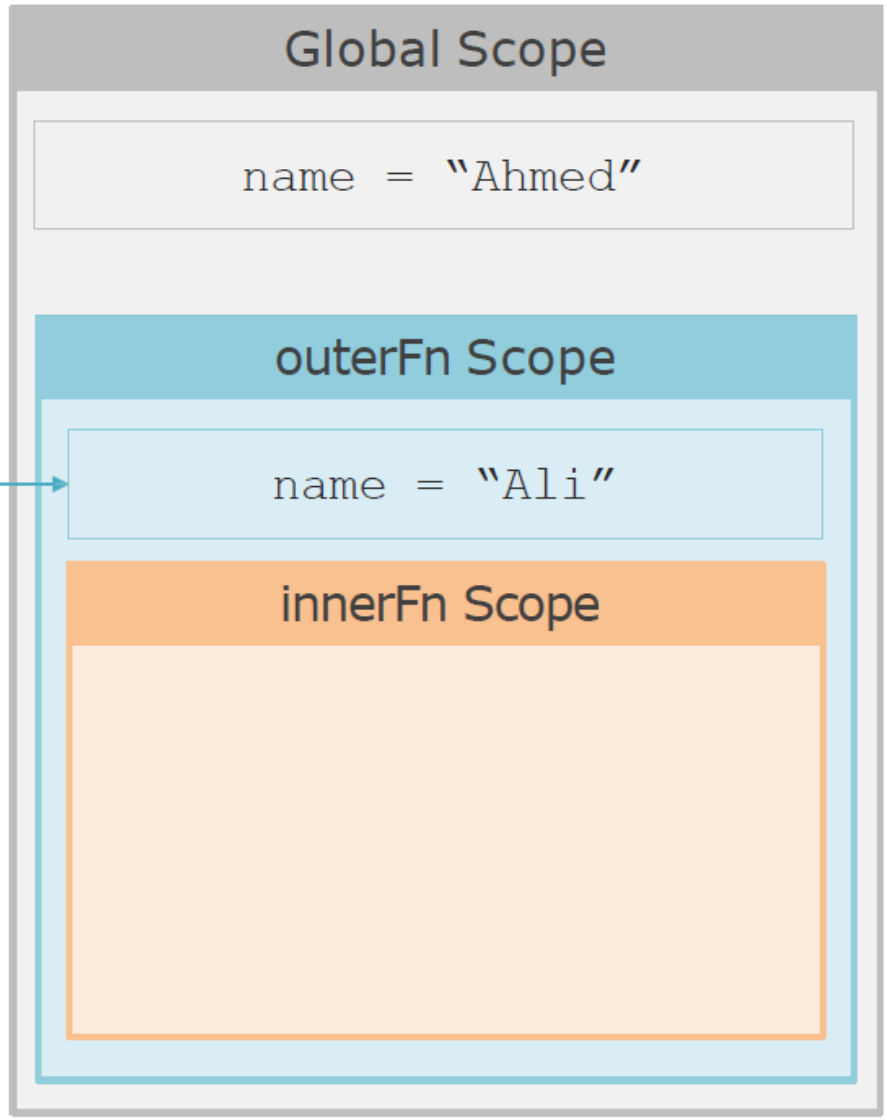


Lexical Scope

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
            print(name)
        innerFn()
outerFn()
```

Output:

name
???

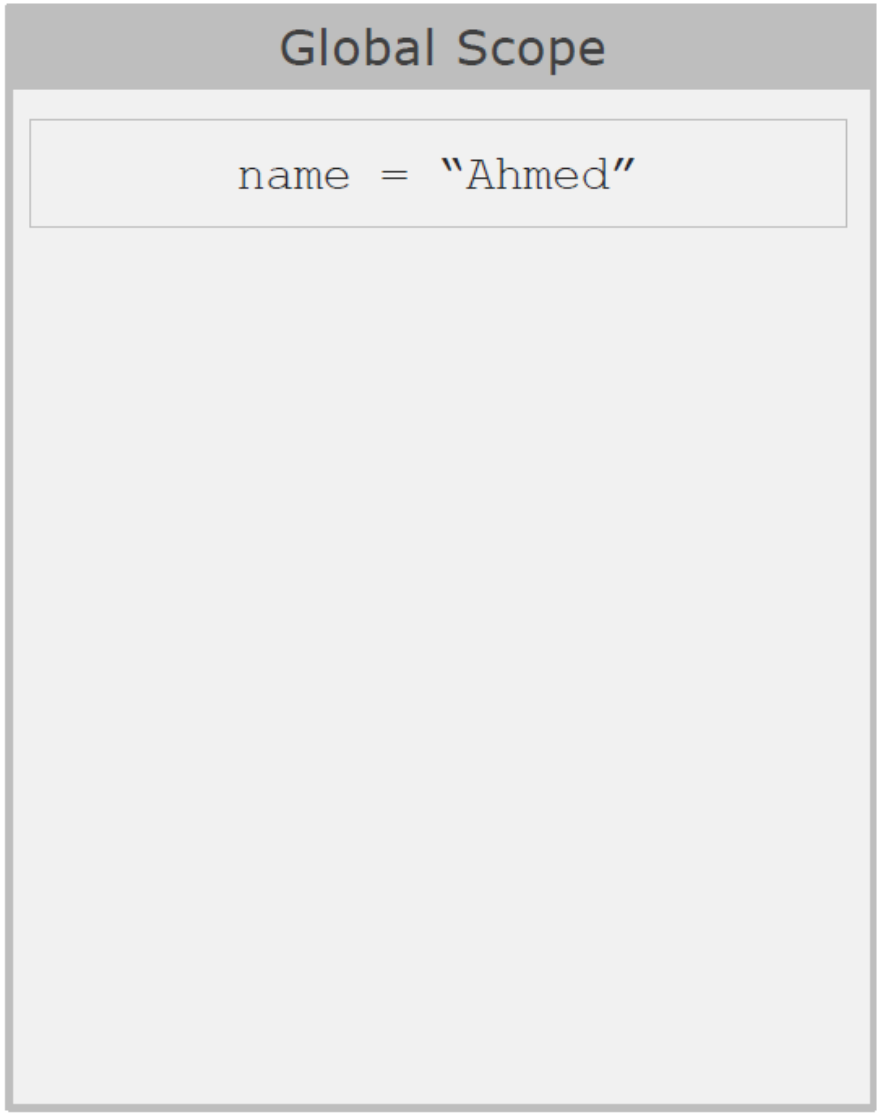


Lexical Scope

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
                print(name)
        innerFn()
outerFn()
print(name)
```

Output:

```
Ali
```

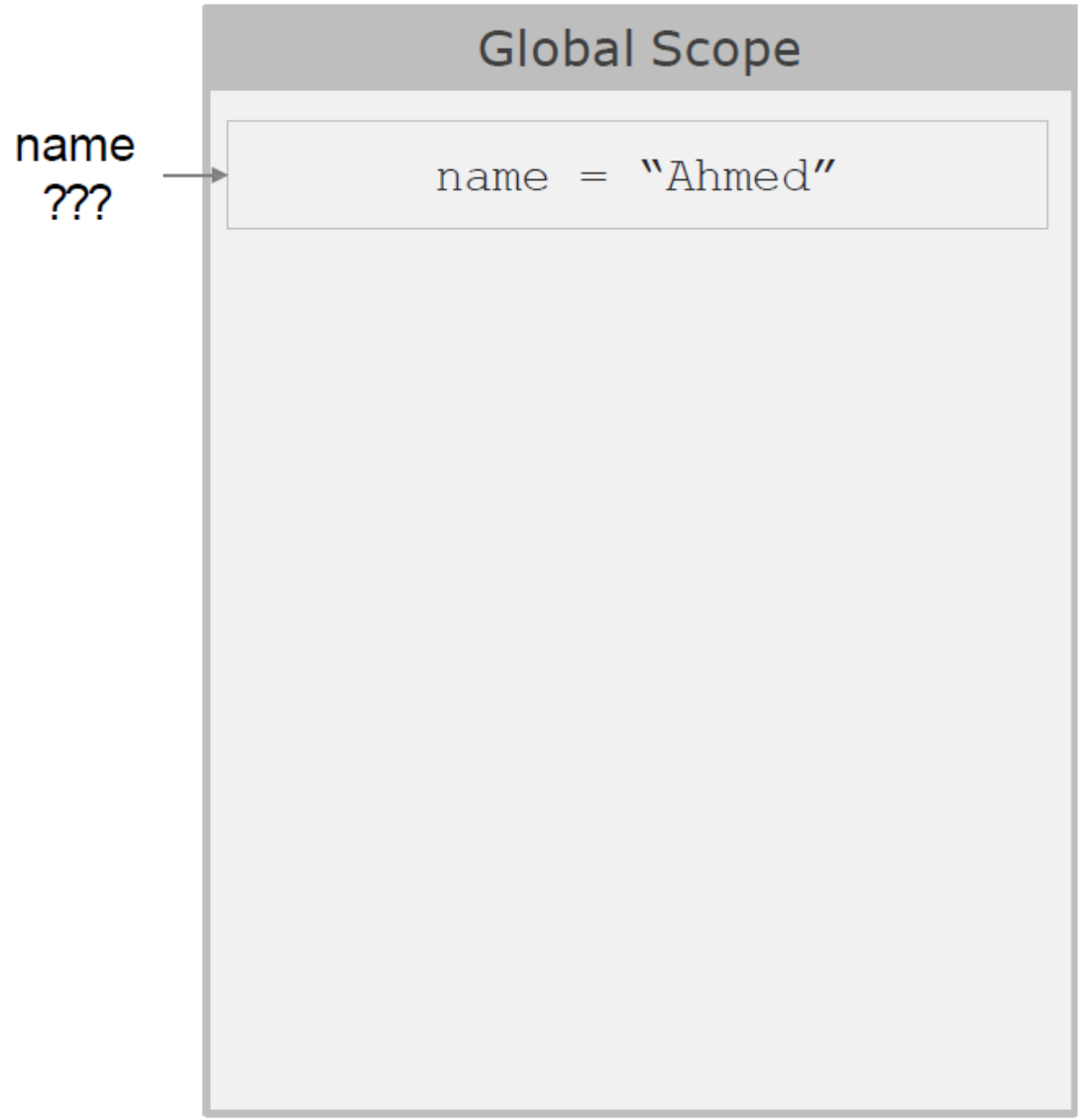


Lexical Scope

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
                print(name)
        innerFn()
outerFn()
print(name)
```

Output:

```
Ali
Ahmed
```



global Keyword

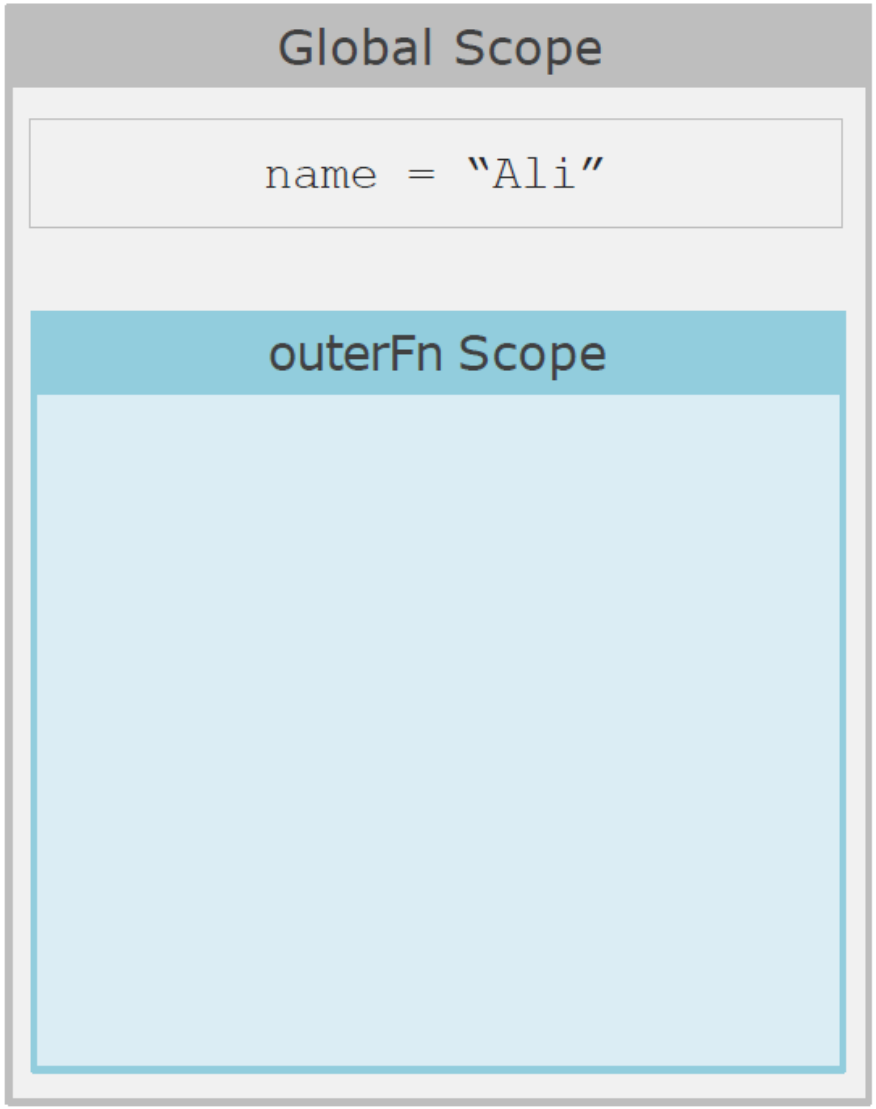
```
name = "Ahmed"
def outerFn():
        global name
        name = "Ali"
        def innerFn():
                print(name)
        innerFn()
outerFn()
```

Output:

global Keyword

```
name = "Ahmed"
def outerFn():
        global name
        name = "Ali"
        def innerFn():
                print(name)
        innerFn()
outerFn()
```

Output:

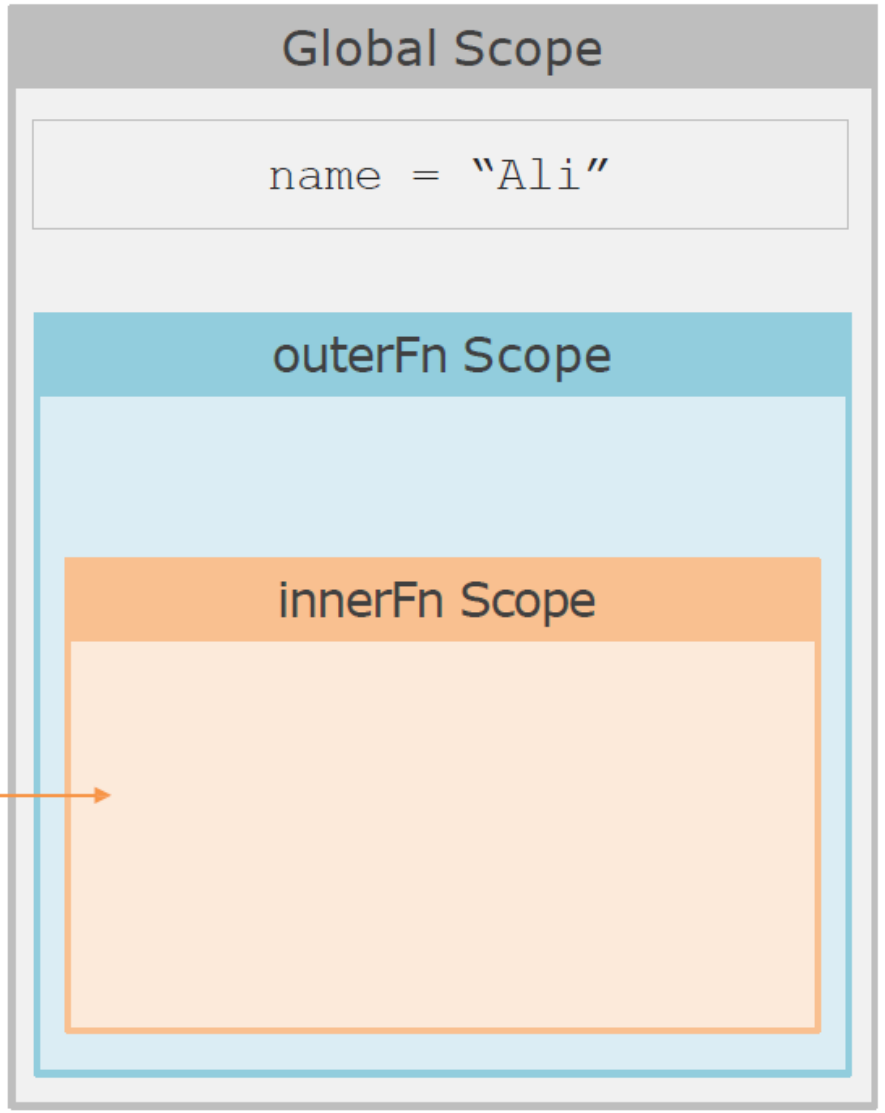


global Keyword

```
name = "Ahmed"
def outerFn():
        global name
        name = "Ali"
        def innerFn():
            print(name)
        innerFn()
outerFn()
```

Output:

name
???

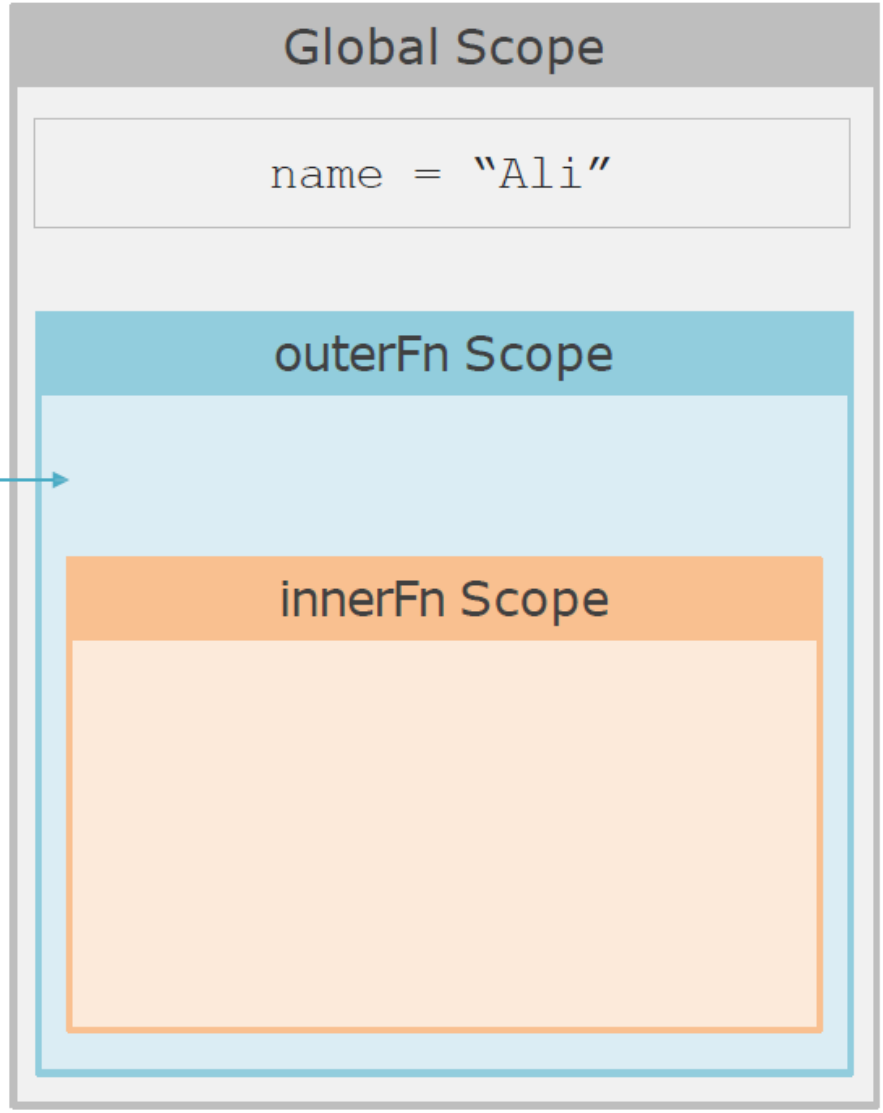


global Keyword

```
name = "Ahmed"
def outerFn():
        global name
        name = "Ali"
        def innerFn():
            print(name)
        innerFn()
outerFn()
```

Output:

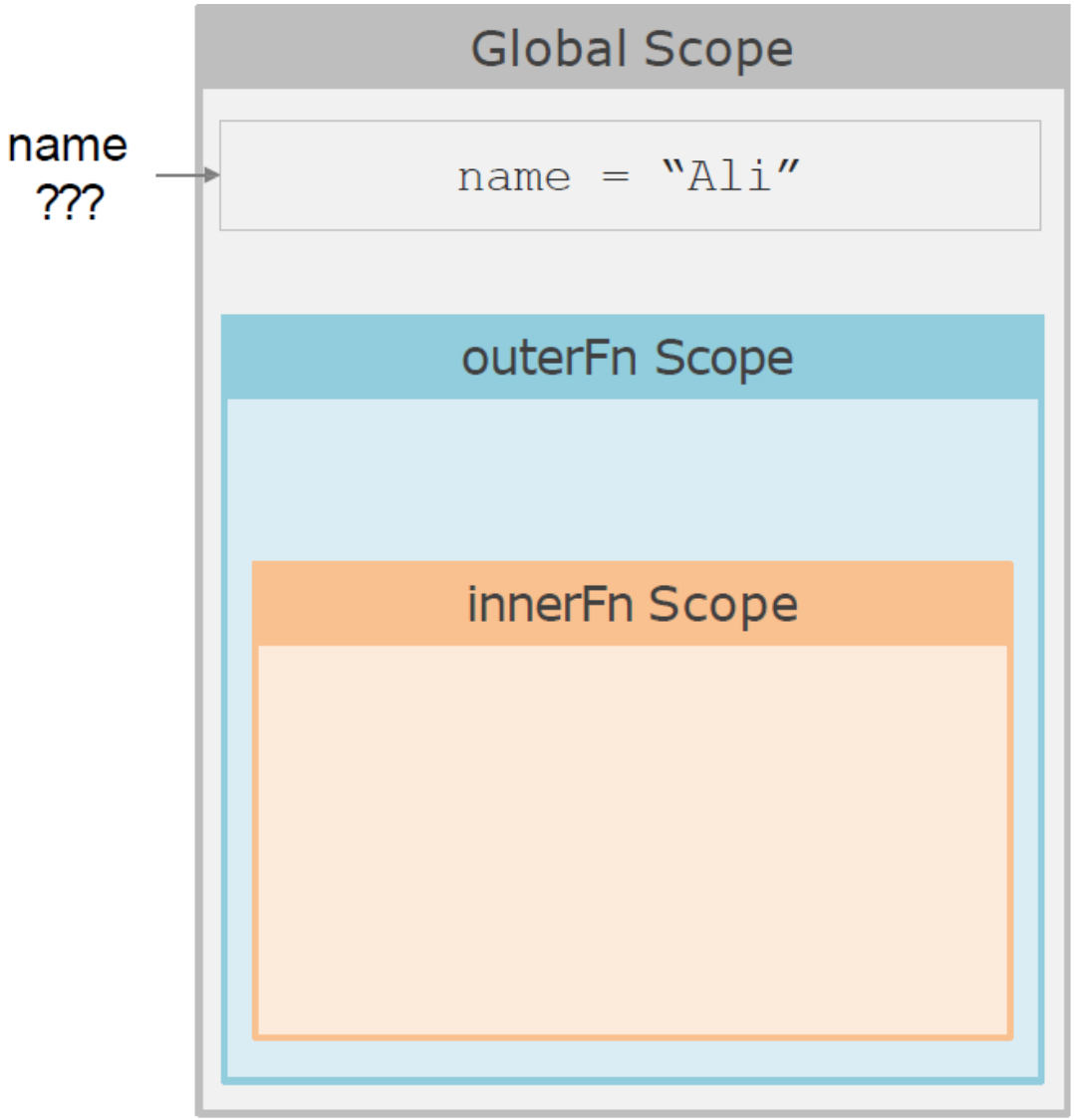
name
???



global Keyword

```
name = "Ahmed"
def outerFn():
        global name
        name = "Ali"
        def innerFn():
            print(name)
        innerFn()
outerFn()
```

Output:

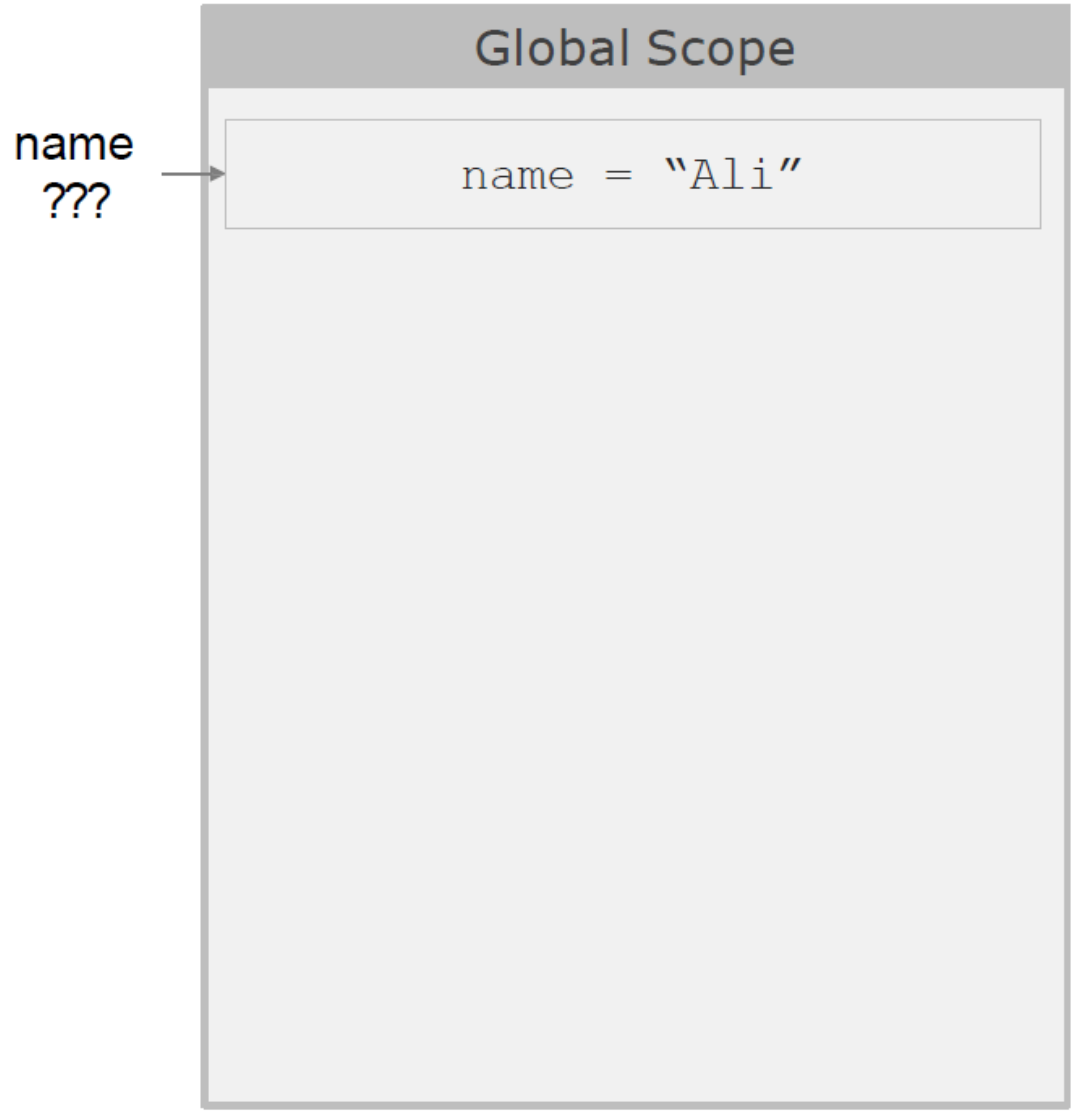


global Keyword

```
name = "Ahmed"
def outerFn():
        global name
        name = "Ali"
        def innerFn():
                print(name)
        innerFn()
outerFn()
print(name)
```

Output:

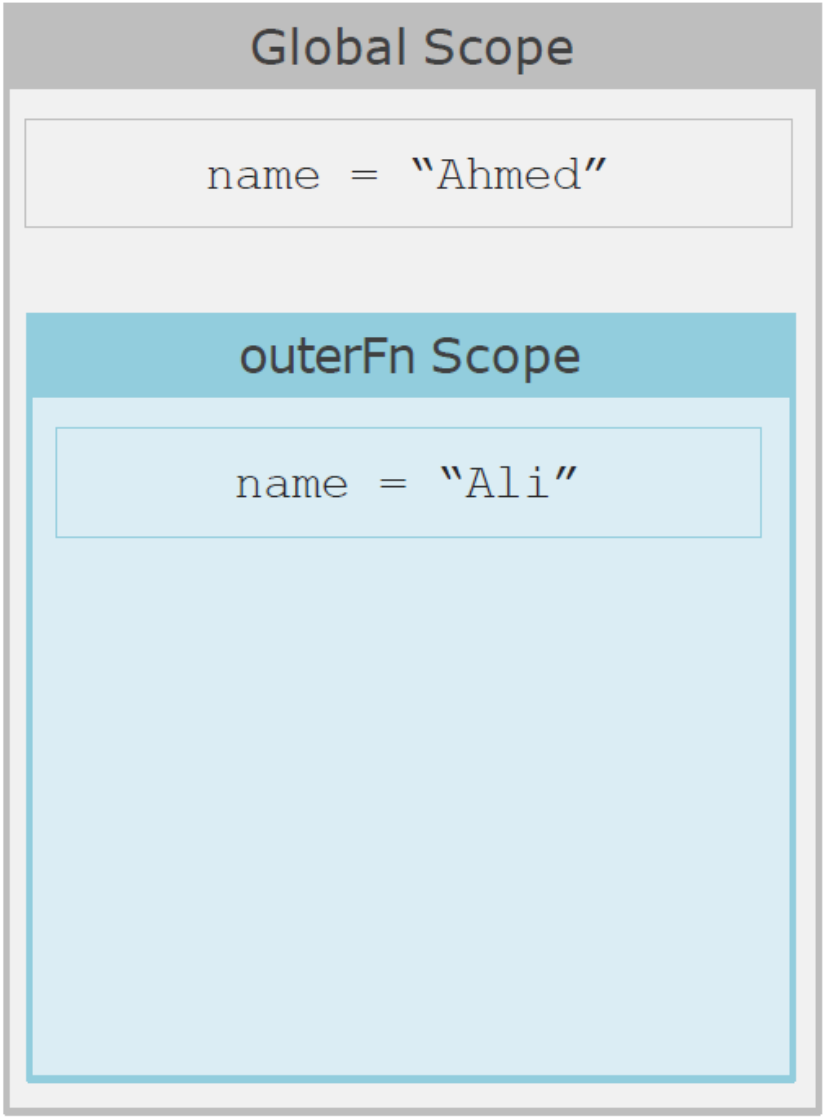
```
Ali
Ali
```



nonlocal Keyword

```
name = "Ahmed"
def outerFn():
    name = "Ali"
    def innerFn():
        nonlocal name
        print(name)
        name = "Sara"
    innerFn()
    print(name)
outerFn()
```

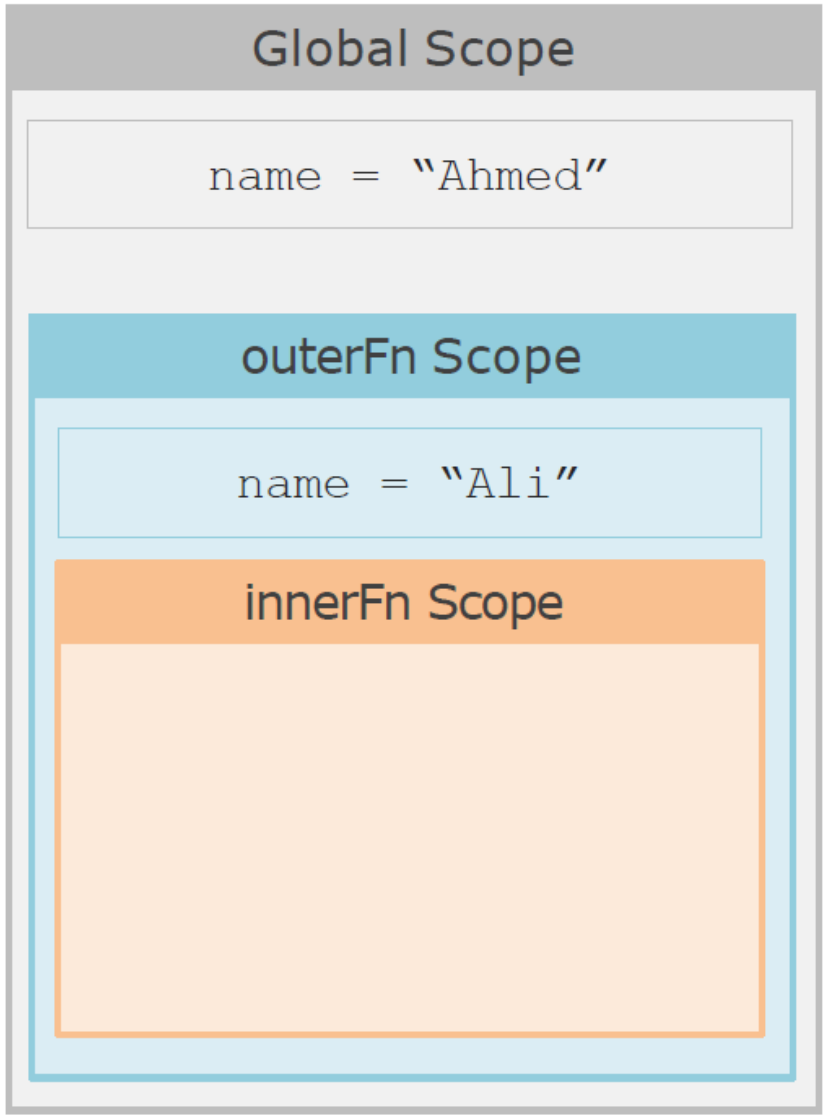
Output:



nonlocal Keyword

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
            nonlocal name
            print(name)
            name = "Sara"
    →   innerFn()
        print(name)
outerFn()
```

Output:

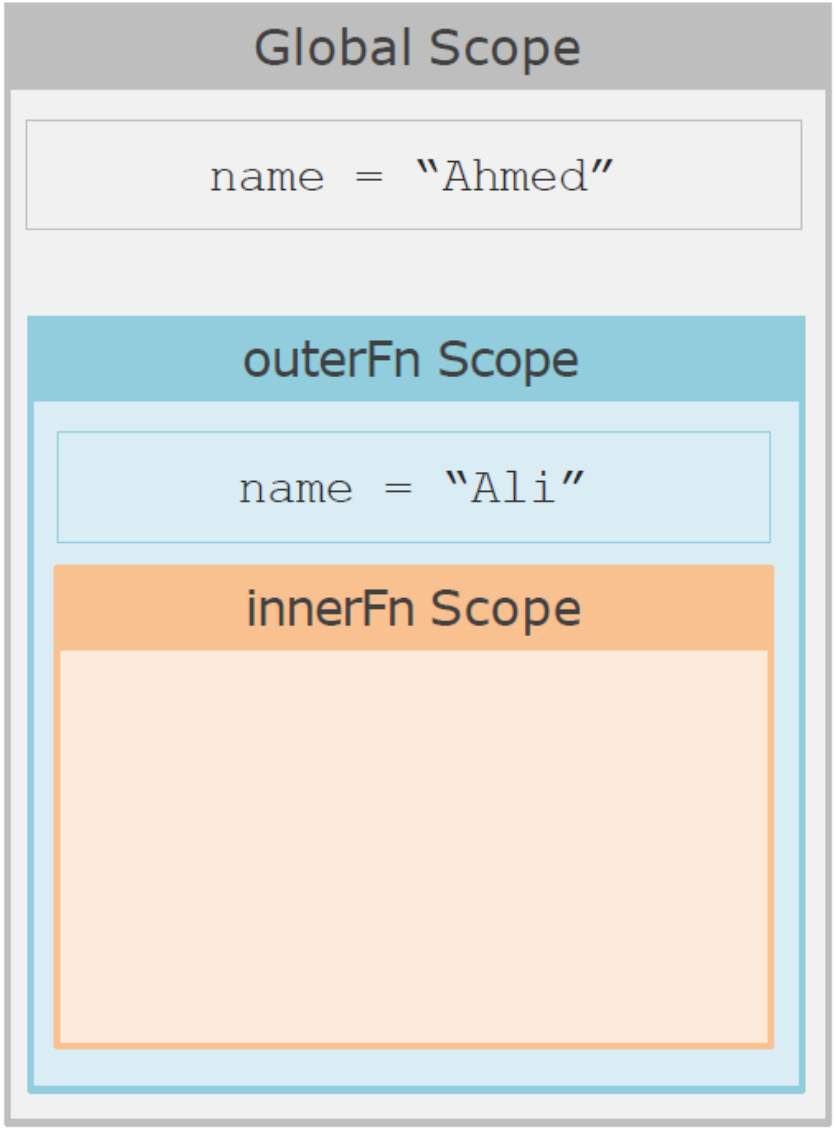


nonlocal Keyword

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
            nonlocal name
            print(name)
            name = "Sara"
        innerFn()
        print(name)
outerFn()
```

Output:

```
Ali
```

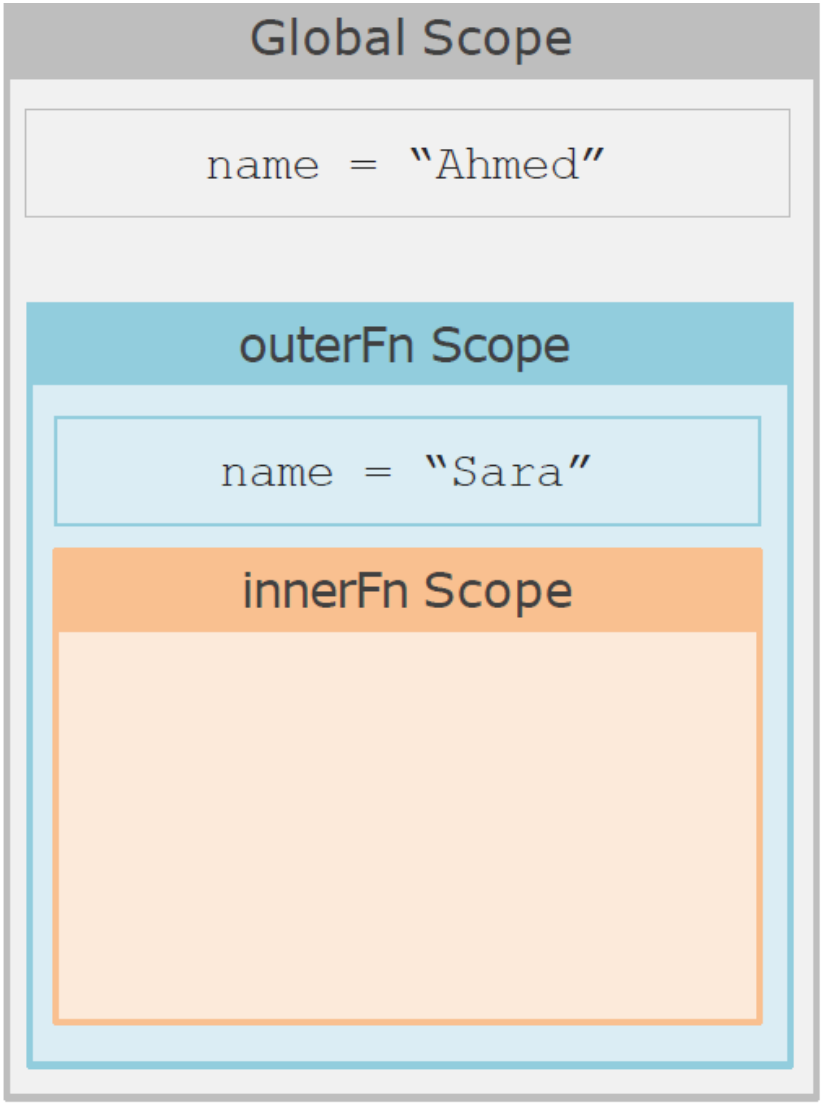


nonlocal Keyword

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
            nonlocal name
            print(name)
            name = "Sara"
        innerFn()
        print(name)
outerFn()
```

Output:

```
Ali
```

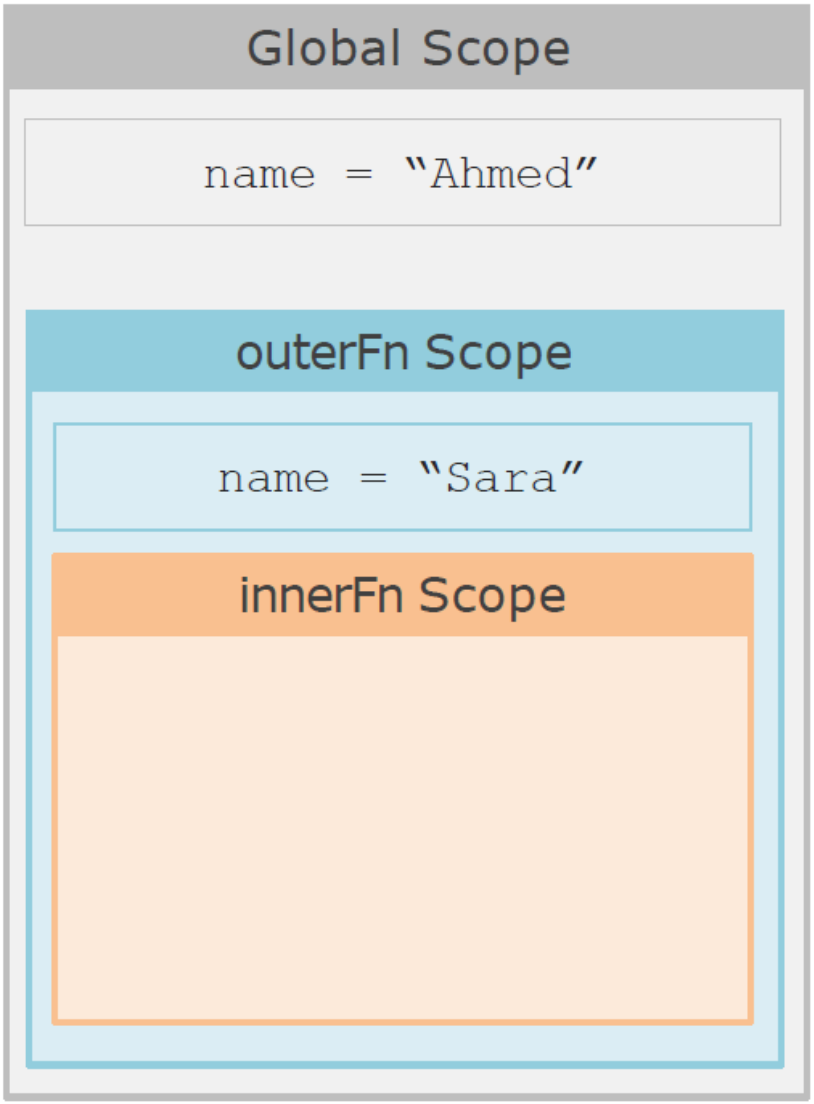


nonlocal Keyword

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
            nonlocal name
            print(name)
            name = "Sara"
        innerFn()
    →   print(name)
outerFn()
```

Output:

```
Ali
```

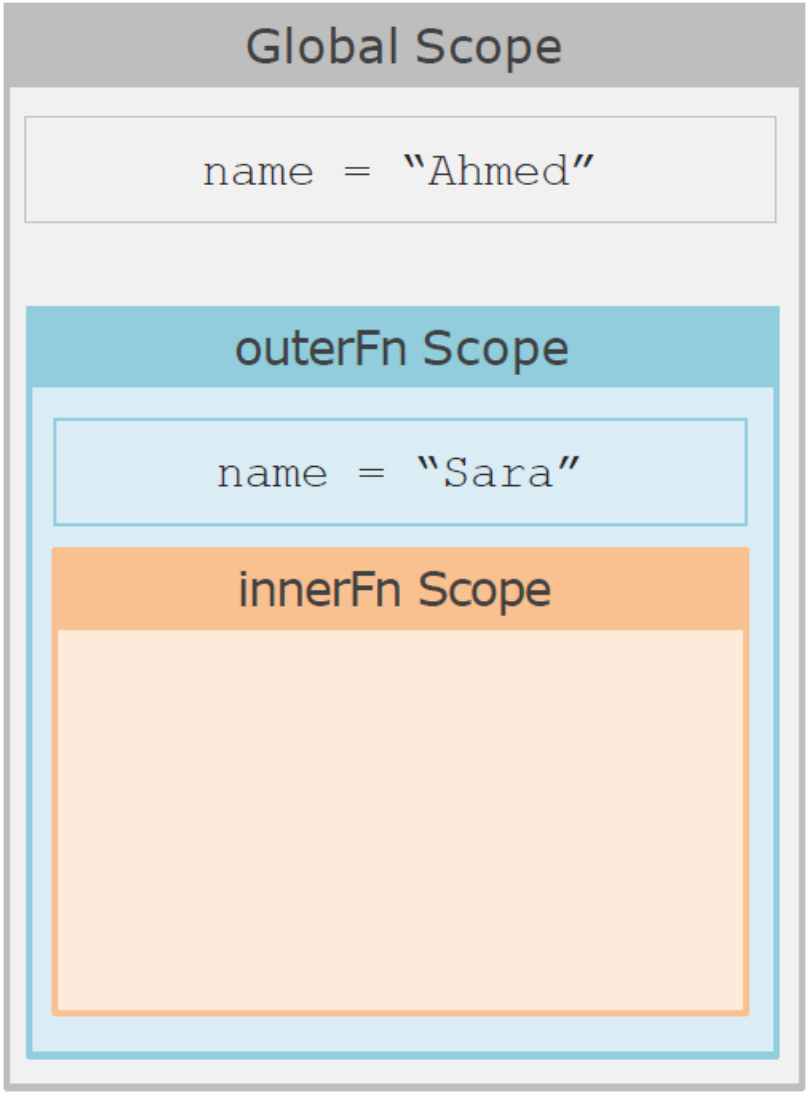


nonlocal Keyword

```
name = "Ahmed"
def outerFn():
        name = "Ali"
        def innerFn():
            nonlocal name
            print(name)
            name = "Sara"
        innerFn()
        print(name)
outerFn()
```

Output:

```
Ali
Sara
```



Lambda

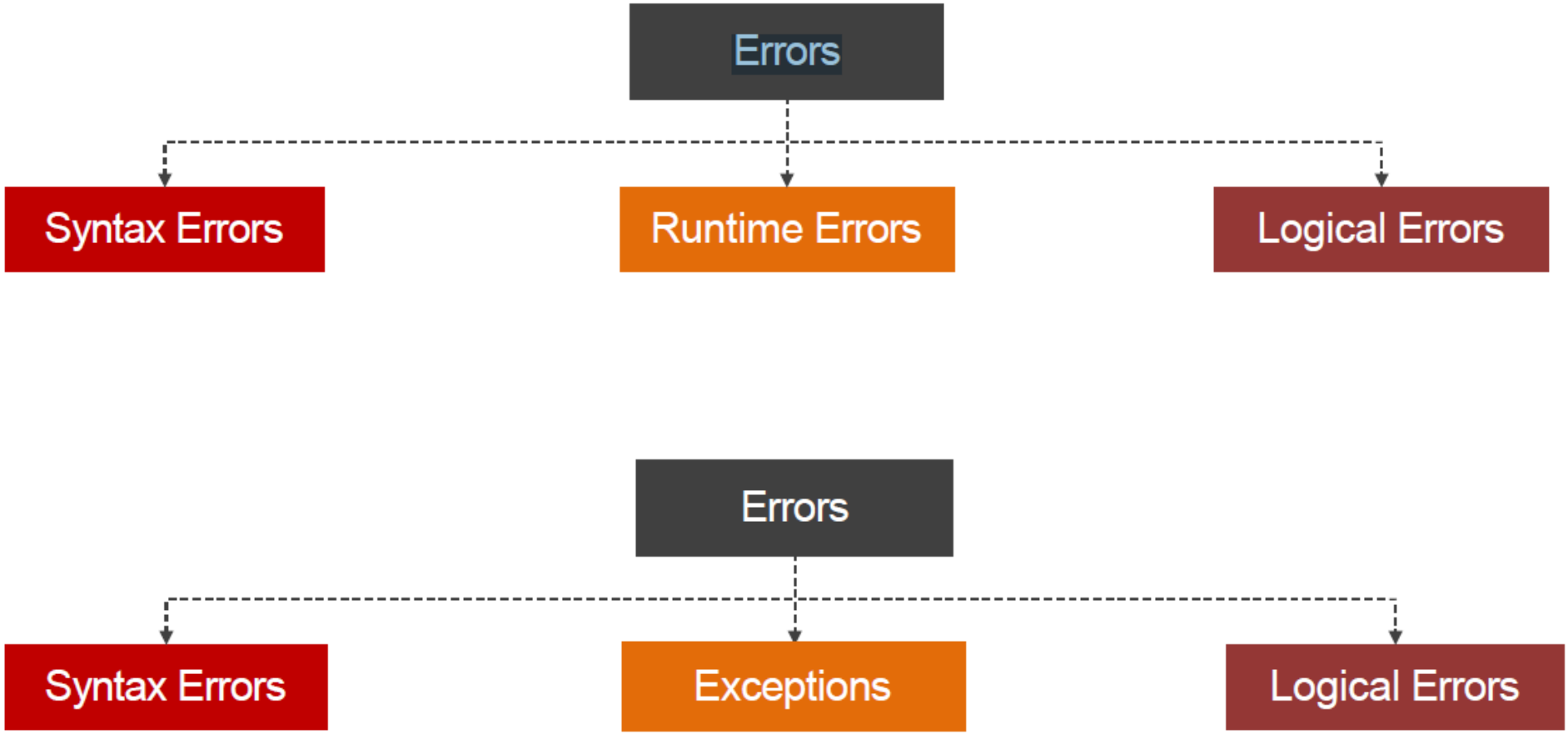
- small **anonymous** function that can have any number of arguments, but can only have one expression.
- used for simple tasks

lambda arguments: expression

```
add = lambda x, y: x + y
print(add( x: 5,  y: 3))   # Output: 8

square = lambda x: x ** 2
print(square(4))   # Output: 16
```

Error handling



Error handling

Syntax Errors

Errors that will show up if you doesn't follow Python Syntax Rules

```
print("You missed the closing round braces "
```

```
 print("You missed the closing round braces "
                                              ^
SyntaxError: invalid syntax
```

Error handling

Exceptions

Errors detected during execution are called **Exceptions**

```
print(firstname);
```

```
NameError: name 'firstname' is not defined
```

Error handling

Handling Exceptions

```
try:
    doTry()
except:
    doExcept()
finally:
    doFinally()
```

Put the code that you want to handle its exceptions

Handle the exception if it raised in the try clause

Put the code that you want to run always if there is an exception or not.

Error handling

Handling Exceptions

```
try:
        for i in range(3):
            print(3/i)
except ValueError:
        print("Value Error")
except ZeroDivisionError:
        print("you divided by 0")
finally:
        print("this will print no matter what")
```

Exercises