

**Programming, Data Structures and Algorithms in Python**  
**Prof. Madhavan Mukund**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Week – 06**  
**Lecture – 02**  
**Global Scope, Nested Functions**

(Refer Slide Time: 00:02)

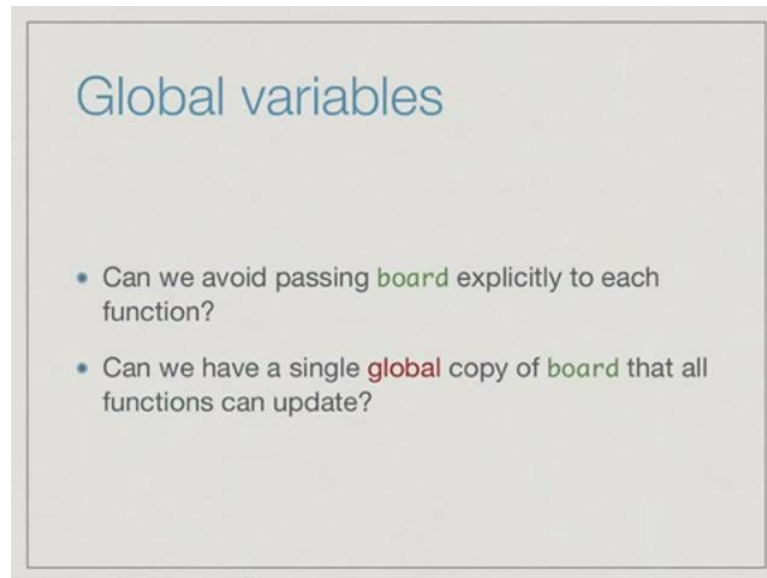
```
Recall 8 queens

def placequeen(i,board): # Trying row i
    for each c such that (i,c) is available:
        place queen at (i,c) and update board
    if i == n-1:
        return(True) # Last queen has been placed
    else:
        extendsoln = placequeen(i+1,board)
        if extendsoln:
            return(True) # This solution extends fully
        else:
            undo this move and update board
    else:
        return(False) # Row i failed
```

We were looking at the 8 queens problem, and our solution involved representing the board, which squares are under attack and placing the queens one by one.

One feature of this solution is that we had to keep passing the board through the functions in order to update them or to resize them and I had to initialize them and so on because the board had to kept updated through each function. Now the question is can we avoid passing the board around all over the place?

(Refer Slide Time: 00:31)

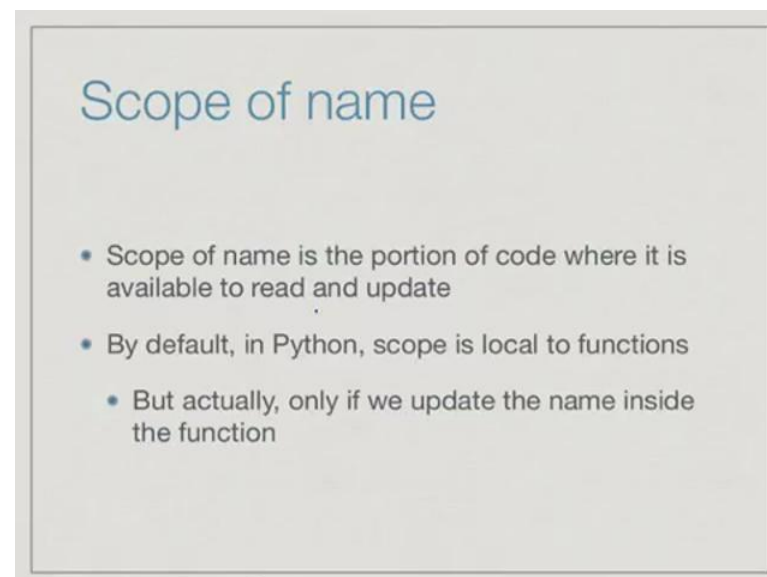


Global variables

- Can we avoid passing `board` explicitly to each function?
- Can we have a single `global` copy of `board` that all functions can update?

So, can we avoid passing this board explicitly or can we have a single global copy of the board that all the functions can update which will save us passing this board back and forth.

(Refer Slide Time: 00:42)



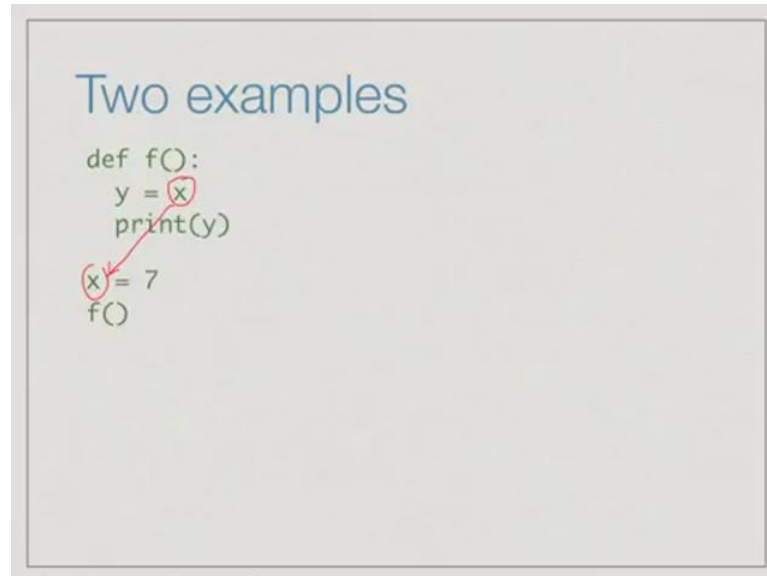
Scope of name

- Scope of name is the portion of code where it is available to read and update
- By default, in Python, scope is local to functions
  - But actually, only if we update the name inside the function

So, this brings us to a concept of Scope. The scope of a name in Python is the portion of the code where it is available to read and update. Now by default in python scope is local to a function, we saw that if we use a name inside a function and that it is different from

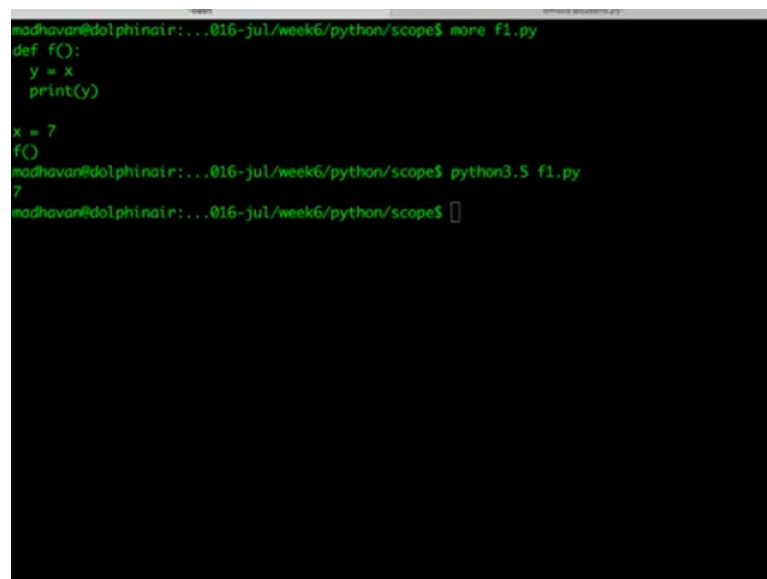
using the same name outside the function. But actually this happens only when we update the name inside the function.

(Refer Slide Time: 01:08)



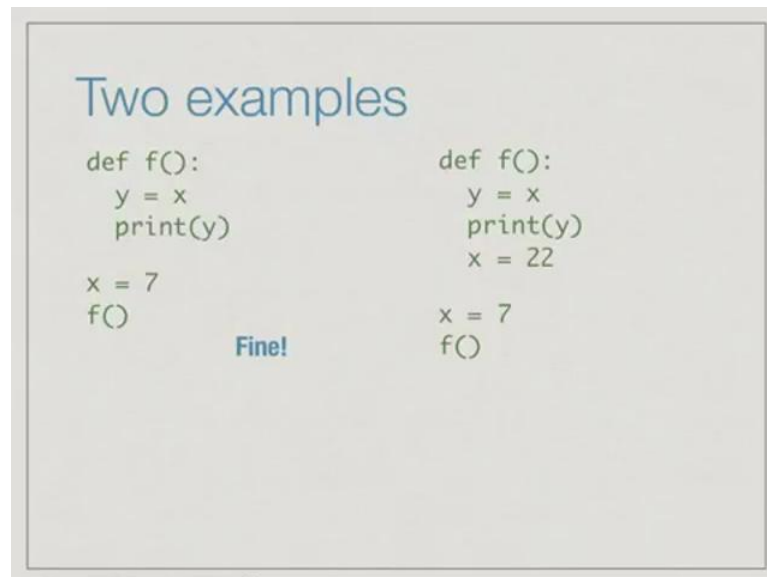
Let us look at this particular code. Here we have a function f which reads the values x and prints it by storing it in the name y, Now the question is: what is this x? Well there is an x here. So, will this x inside the function correctly reflect the x outside the function or not.

(Refer Slide Time: 01:33)



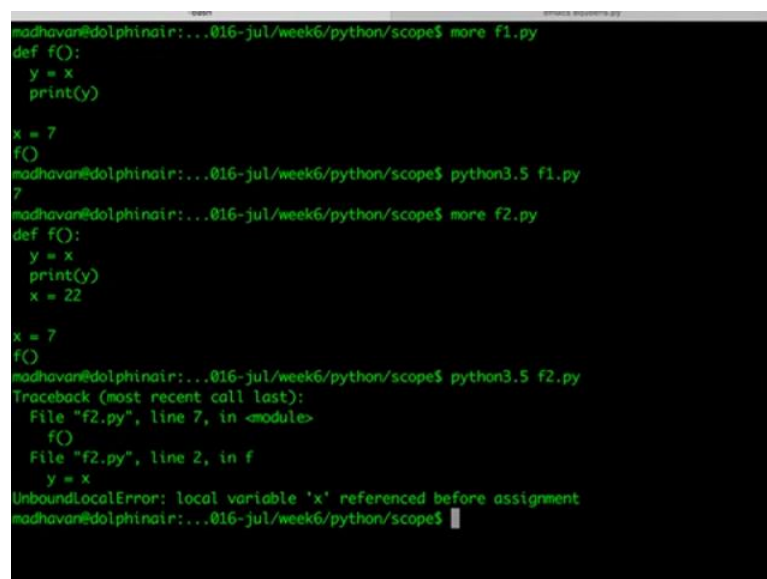
So here we see that function, we have written a file f1 dot py which contains exactly that code. So, we have function f which reads an x from outside and tries to print it. If you run this, then indeed it prints the value 7 as we expect, so y gets the value 7 because the x has the value 7 outside and that x is inherited itself a function from inside f .

(Refer Slide Time: 02:03)



So this works. Now what if you do this, and this is exactly the same function except that after printing the values of y it sets x equal to 22 inside f. Now what happens?

(Refer Slide Time: 02:19)



So here is f2 dot py the code in **the middle** of the screen, so only difference with **respect** to f 1 dot py is extra assignment x equal to 22 inside f. Now if you try to run f 2 dot py, then it gives us an error saying that the original assignment y equal to x gives us an unbound local name there is no x which is available at this point inside f. So, somehow assigning x equal to 22 inside f **changed** the status of x, it is no longer willing to look up the outside x it will insist that there is an inside x. This gives as an error.

(Refer Slide Time: 02:59)

Two examples

```
def f():  
    y = x  
    print(y)  
x = 7  
f()
```

Fine!

```
def f():  
    y = x  
    print(y)  
    x = 22  
x = 7  
f()
```

Error!

- If x is not found in f(), Python looks at enclosing function for **global** x
- If x is updated in f(), it becomes a **local** name!

So **if** x is not found in f, Python is willing to look at the enclosing function for a global x. However, if x is updated in f then it becomes a local name and then it gives an error.

(Refer Slide Time: 03:15)

## Global variables

- Actually, this applies only to immutable values

```
def f():  
    y = x[0]  
    print(y)  
    x[0] = 22  
  
x = [7]  
f()
```

So strictly speaking this applies only to immutable values. If we change this function as follows we made `x` not an integer, but a list for example and we asked `y` to pick up the 0th element in the list and then later in `f` we change the 0th element of `x` to 22.

(Refer Slide Time: 03:38)

```
madhavan@dolphinair:~$ cat f3.py  
def f():  
    y = x[0]  
    print(y)  
    x[0] = 22  
  
x = [7]  
f()  
madhavan@dolphinair:~$ python3.5 f3.py  
7  
madhavan@dolphinair:~$
```

Here we have this function in which we now changed `x` from an integer to a list and then we try to assign it in `y`. But we update that list inside the function and then if you run it then it does print the value 7 as we expect.

(Refer Slide Time: 03:55)

## Global variables

- Actually, this applies only to immutable values
- Global names that point to mutable values can be updated within a function

```
def f():  
    y = x[0]  
    print(y)  
    x[0] = 22  
  
x = [7]  
f()
```

**Fine!**

So this works. **If** we have an immutable value, I mean mutable value sorry, then we can actually change it inside f and nothing will happen. So, global names that point to mutable values can be updated within a function.

In fact, this means therefore that the problem that we started out to solve namely; how to avoid passing the board around **with** its inside 8 queens **actually** requires no further explanation. Since board is a dictionary, it is a mutable value and in fact we can write 8 queens in such a way that we just ignore passing the board around, we change all the definitions so that board does not occur and works fine.

(Refer Slide Time: 04:33)

```
madhavan@dolphinair:~/.6-jul/week6/python/8queens$ python3.5 8queens-global.py
How many queens? 8
(0, 0)
(1, 4)
(2, 7)
(3, 5)
(4, 2)
(5, 6)
(6, 1)
(7, 3)
madhavan@dolphinair:~/.6-jul/week6/python/8queens$
```

Here we have rewritten the previous code just removing board from all the functions. So initialize earlier took board and n now it just takes n print board does not taken argument at all and all of them are just referring to this global value board which you can see everywhere. So, we have this global value board here which is being referred to inside the function and it does not matter that is not being passed because this is the mutable value, so it is going to look for the value which is defined outside namely this empty dictionary. And then all these functions like place queen or undo queen or add queen just take their relevant parameters and implicitly refer to the global value of board.

So, if you run this now this global version, we get exactly the same output. So, as we said for our purpose which is to fix that 8 queens problem without having to pass the board around, the fact that python implicitly treats mutable global names **as** updatable within a function is all that we need.



(Refer Slide Time: 05:36)

## Global immutable values

- What if we want a global integer
- Count the number of times a function is called
- Declare a name to be global

```
def f():  
    global x  
    y = x  
    print(y)  
    x = 22  
x = 7  
f()  
print(x)
```

But, what if we actually want a global integer? Why would you want a global integer? Well, suppose for instance we want to count the number of times a function is called. So every time a function is called we would like to update that integer inside this function. But that integer cannot be a local name to the function because that local value will be destroyed in the function, we want it to persist, so it must be a value which exists outside the function. But being an integer it is an immutable value and therefore we try to update it inside the function it will treat it as a local value. So, how do we get around this?

Python has a declaration called Global, which says a name is to be referred to globally and not taken as a local value. If we change our earlier definition of f, so that we add this particular tag global x then it says that this x and this x both refer to the same x outside. This is the way of telling python, do not confuse this x equal to 22 with creation of a new local name x. All x's referred to in f are actually the same as the x outside and to be treated as global values. So, this is one way in which we can make an immutable value accessible globally within a Python program.

(Refer Slide Time: 06:51)

```
madhavan@dolphinair:~$ cd /home/madhavan/016-jul/week6/python/scope && more f4.py
def f():
    global x
    y = x
    print(y)
    x = 22

x = 7
f()
print(x)
madhavan@dolphinair:~$ cd /home/madhavan/016-jul/week6/python/scope && python3.5 f4.py
7
22
madhavan@dolphinair:~$ cd /home/madhavan/016-jul/week6/python/scope &&
```

So here is that global code. We have global x, and just make sure that the x is equal to 22 inside is actually affecting that x outside. We have a print x now after the call to f. So the bottom of the main program we have print x, now x was 7 before f was called but x got set to 22 inside f. So, we would expect the second print statement to give us 22. This statement should first print 7. It should print a 7 from the print y and then print 22 from this print x. So, if you run this indeed this is what we see right we have two lines, the first 7 comes from print y and the second level 22 comes from print x outside.

(Refer Slide Time: 07:33)

### Nest function definitions

- If we look up x, y inside g() or h() it will first look in f(), then outside
- Can also declare names global inside g(), h()
- Intermediate scope declaration: nonlocal
- See Python documentation

```
def f():
    def g(a):
        return(a+1)

    def h(b):
        return(2*b)

    global x
    y = g(x) + h(x)
    print(y)
    x = 22

x = 7
f()
```

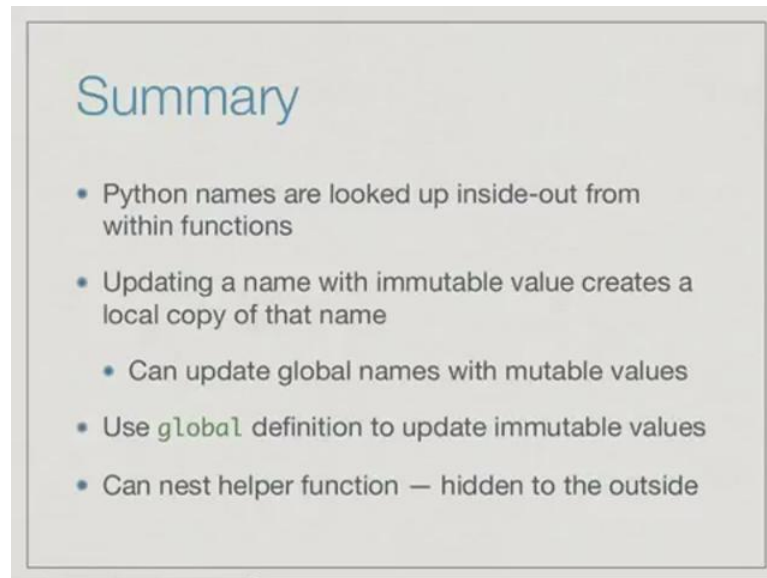
While we are on the topic of local scope, Python allows us to define functions within functions. So, here for instance the function `f` has defined functions `g` and `h`, `g` of `a` returns `a plus 1`, `h` of `b` returns two times `b`. Now we can update `y` for instance by calling `g` of `x` plus `h` of `x` rather than just setting it the value `x`.

Now, the point to note in this is that these functions `g` and `h` are only visible to `f`. They are defined within this scope of `f`. So, they are inside `f`, and hence they are not visible outside. So, from outside if I go if I ask `g` of `x` at this point this will be an error, because it will say there is no such `g` defined. This is useful because now you can define local functions which we may want to perform one specialized task which are relevant to `f`, but it should not be a function which is exposed to everybody else and this is a possibility.

Of course, the same rules apply so if we look up `x` inside `g` or `h`. So, if we look up an `x` here it will first try to look up `f`, if it is not there in `f` it will go outside and so on. So, either we will declare it global in which case we can update it within `g` or `h` or it will use the same rule as before if we do not update an immutable value it will look outside and if it is a mutable value it will allow us to update it from inside.

Now there are some further refinements. Python has an intermediate scope called non local which says within `g` and `h` refer to the value inside `f`, but not to the value outside `f`. This is a technicality and it will not be very relevant if we need it we will come back it, but for the moment if you want to find out more about non local declarations please see the Python documentation. But global is the important one, global allows us to transfer an immutable value from outside in to a function and make it updatable within a function.

(Refer Slide Time: 09:33)



## Summary

- Python names are looked up inside-out from within functions
- Updating a name with immutable value creates a local copy of that name
  - Can update global names with mutable values
- Use `global` definition to update immutable values
- Can nest helper function — hidden to the outside

To summarize, what we **have** seen is **that** Python names **are** looked up inside out from within functions, if we update an immutable value it creates a local **copy** so we need to have a global definition to update immutable values.

On the other hand, if you have mutable values like **lists** and dictionaries there is no problem. Within a function we can implicitly refer to the global one and **update** it. And this we saw in our 8 queens solution, we can make the board into a global value and just keep updating it within each function rather than passing it around explicitly as an argument.

Finally, what we **have** seen is that we can nest functions. We can create so called helper functions within functions that are hidden to the outside that can be **used** inside **the** function to logically break up its activities in to smaller units.