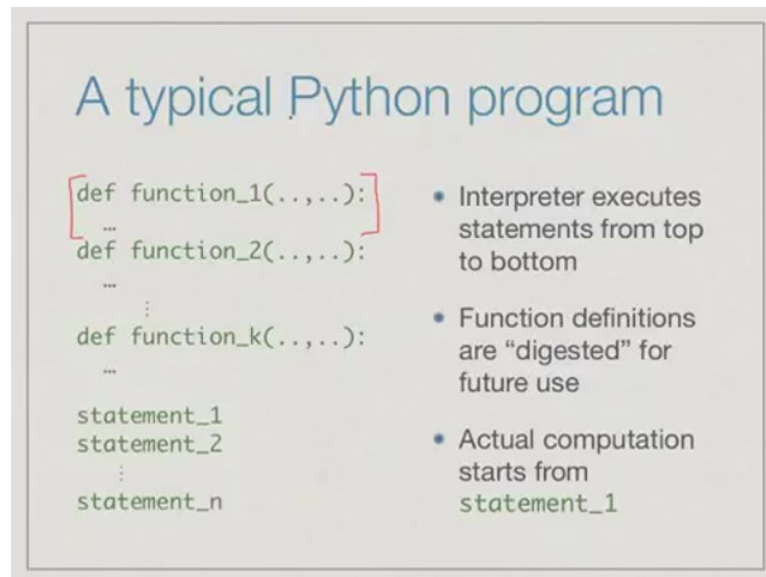


Programming, Data Structure and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Chennai Mathematical Institute, Madras

Week - 02
Lecture - 05
Functions

We have seen how to alter the flow of a program by using if, for and while. We can have conditional execution, we can have repeated execution.

(Refer Slide Time: 00:01)



A typical Python program

```
def function_1(...):  
    ...  
def function_2(...):  
    ...  
    ...  
def function_k(...):  
    ...  
statement_1  
statement_2  
    ...  
statement_n
```

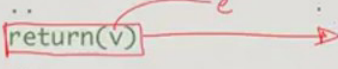
- Interpreter executes statements from top to bottom
- Function definitions are “digested” for future use
- Actual computation starts from statement_1

The last ingredient in our typical Python program is a function. What is a function? A function is a group of statements which performs a given task. So of course, we could write the function code as part of the main program, but by isolating it we can logically separate out units of work and very often these functions are called repeatedly with different arguments. So, they constitute a unit of computation which can be used repeatedly from time to time.

(Refer Slide Time: 00:43)

Function definition

```
def f(a,b,c):  
    statement_1  
    statement_2  
    ..  
    return(v)  
    ..
```



- Function name, arguments/parameters
- Body is indented
- `return()` statement exits and returns a value

We define functions using the `def` statement as we have seen informally. So the definition defines the name of function, in this case we have just called it `f` usually we would give it more meaningful names. Then it says that this function takes three values as inputs, so these are called parameters or arguments. So, the first one is called `a`, the second is called `b`, and third is called `c`, and within the body of the program of the function `a`, `b` and `c` will refer to the values which are **passed** to this function for a given call. Within a function we might have a statement like this called `return`.

The body of the function is indented like we had for `if`, `while` and `for`, and the `return` statement if it **is** encountered, **it says that** at this point the execution of the function will end and you will get back to where you called function from returning the value in the name `v`. This could be any expression; we could just have `return` of a constant or `return` of `v` plus one or whatever.

(Refer Slide Time: 01:50)

Passing values to functions

- Argument value is substituted for name

```
def power(x,n):  
    ans = 1  
    for i in range(0,n):  
        ans = ans*x  
    return(ans)
```

```
power(3,5)  
x = 3  
n = 5  
ans = 1  
for i in range..
```

- Like an implicit assignment statement

When we call a function we have to pass values for the arguments, and this is actually done exactly the same way as assigning that value to a name. Suppose, we have function like this which takes x and raises **it** to the power n . Let us just look at the function just to understand what the code is doing. We assume that the value of the answer is 1. And now for as many i as there are in the range 0 to n minus 1 we multiply x into answer so we get effectively x times, x times, x n times. Each time we go through this loop we multiply one more x and finally we return the answer that we have got.

Now the way we would use this function in our code is to write an expression of the form, say `power 3, 5`, so obviously, what this means is that 3 should be used for x and 5 should be used for n and we would then run this code with the values x equal to 3 and n equal to 5.

Actually, you can imagine that when we run this code, it is as though we have this code inserted into our program at this point preceded by this **assignment**. So, this assignment basically says set the value of the name x to the value **passed** by this namely 3, set n to 5. This assignment is what takes place effectively when you call a function. And since it is an assignment, **this** behaves very much like assignment in the regular case.

(Refer Slide Time: 03:32)

Passing values ...

$x = y$ { copy - immutable
share - mutable

- Same rules apply for mutable, immutable values
- Immutable value will not be affected at calling point
- Mutable values will be affected

In particular the same rules **apply** for mutable and immutable values. Remember we said that when we write something like $x = y$, if it is immutable that is the value in y cannot be change in place then we copy the value and we get a fresh copy in x , so the value in x and the value in y are **disjoint**. So this is if it is immutable. And if it is mutable, we said we do not copy, we share the value; that is, both names will point to the same copy of the value, so change in one will also make a change in the other; that happens with mutable things like lists.

Immutable values will not **be affected** at the calling point in our case and mutable values will be affected. **It is as** though we are making an assignment of the expression or the name in the calling function, calling point to the name in the function. So, if the function modifies that name, the value of that name; if it is immutable value, nothing will happen here, if it is a mutable value something will happen.

(Refer Slide Time: 04:36)

Example

```
def update(l,i,v):  
    if i >= 0 and i < len(l):  
        l[i] = v  
        return(True)  
    else:  
        v = v+1  
        return(False)
```

ns = [3,11,12]
z = 8
✓ res = update(ns,2,z)
✓ res = update(ns,4,z)

- ns is [3,11,8]
- z remains 8

- Return value may be ignored ✓
- If there is no return(), function ends when last statement is reached

So, here is a simple function just to illustrate this point. The aim of this function is to update a list. So, I give you a list which is called in this function l and I give you a position which is i and what I want to do is I want to replace whatever is there by a new value v. So I get three arguments, l is a list, and then i is the index of the position, and finally v is the value to be replaced.

So, what do we do? First **check** that the index is a valid index. We check that it lies between 0 and l minus 1. So it is greater than equal to 0 and it is strictly lesser the length of l. If so, what we do is just replace l of i by the value v which we have got and we return true to indicate that the update succeeded. Now if i is not in this range then we cannot do an update. So, what we will do is effectively return false. This is just say that the update did not work and then the person, the part of code which is calling this can understand that something went wrong and presumably what went wrong is the index was not in the valid range.

But just to illustrate what happens with immutable values, in this case we are also updating for no good reason the value v to be v plus 1. So, remember that v is being passes a value to be put **in** here and we are assuming normally that v would be a immutable value. Let us assume we call it now, so what we use do is we set up a list of

numbers 3, 11, 12 and then we want to replace this 12 say by 8. So, just for the sake of argument we first set up a new name z called 8 and we say update the list n s at position 2, so remember the positions are 0, 1, 2. So, update **the list in** position 2 by the value of z.

And then we say update the same list at position 4 by the value of z. Now as we saw if the values 4 right then this if will fail, so it will instead go here, this won't work, so it will go here. And what will happen inside the code is that v will be incremented, now v **has** been copied from z. The question is what happens to z? So, as you would expect after executing these four statements, because of this update succeeding the value of z is copied into the list that position 2 and so we get the value 8 instead of the value 12 that we started with. On the other hand, if we execute this statement, then because this is an immutable value the change in v inside the function does not affect z at all. Although v has been incremented from 8 to 9, z remains 8.

This is just to illustrate that if we pass a parameter, through a parameter a value that is mutable it can get updated in function and this is sometimes called a side effect. So the function affects the value in the other program, so this is called a side effect. A side effect can happen if the value is mutable, but if the value is immutable then the value does not change no matter what we do inside the program.

Now, there are couple of other points to note about this function just to illustrate; one is that we have here two return statements: return true or return false. The idea is that they indicate to the calling function whether or not the update succeeded. So ideally you should have said something like result is equal to update, and then check after the update where the result is true or false.

Remember update will update the list or not update the list depending on whether the index is valid and it will return true or false depending on whether they update succeeded. So, by examining the value of whatever is **returned** we can check whether the update we intended worked or not. This is something which we would expect **but we have** not done **it**, so this is just to illustrate that there may be a return value but may be the idea is a function will actually update some mutable value so we do not care what it

returns all the work is done inside the function.

Even though there is a return value you are not obliged to use it, you can just call a function as a separate statement as we have done here it does not have to be part of an assignment. The other thing is that because of this there may be functions which do not return anything useful at all. A typical example would be a function which just displays a message like there was an error or it displays some other indicative things for you to understand what your code is doing. Now such a function just as to display something, **it does** not have to compute or return anything. So, there may be no return function. So, by default what happens is that a function executes like everything else from top to bottom when it is involved.

And now if you encounter a return statement at that point the function stops **executing** and you go back. On the other hand if you run out **of** statements **to execute**, if you reach the last statement then there is nothing more then also the function will end. There is no obligation for a function to actually have a return statement. So, a return statement is useful if the function computes the value and gives you back some result which **you** will use later on, but you may have functions which do not have return value, in which case you can either return some empty thing or you can return nothing and everything will work fine.

(Refer Slide Time: 10:07)

Scope of names

- Names within a function have local **scope**

```
def stupid(x):  
    n = 17  
    return(x)  
  
n = 7  
v = stupid(28)  
# What is n now?
```

- n is still 7
- Name n inside function is separate from n outside

Another point to note about functions in Python is that names within a function are **disjoint** from names outside a function. So let us look at again **at a** kind of toy example which does not have anything useful to do. We have a function **which we call** stupid **which** takes essentially takes an argument and return it, so it does nothing. But in between what it does is it just for no good reason sets name n to have the 17. Now suppose we had in our program outside, a statement which assigned **the** value 7 to the name **n** and then we call this function. Now obviously, if we say stupid of 8 then v will also be the input, so v will become 28.

The question is that while **executing** the fact that v **is** 28; the function internally set n equal to **17**. The question is, we have asked n to be 7 then we call this function n became 17 inside the function is n 17 now or not. So the answer is that n is still 7 and that is because the n inside and the n outside are two different copies of n. So, any name which is used in side of function is to be thought of as **disjoint** from the name outside. Names outside are not visible inside, the names inside **are** not visible outside.

Now this is not something that you **would** normally do because is just confusing if you use the same name inside and outside, but sometimes it is useful to have this separation because very often we do use common things like i j k run through list you know like

ranges and things like that. And it will be a nuisance if we have to use a, remember and use i outside and j inside and make sure that they do not interact. Since they do not interact anyway we can freely use i j wherever we want and not worry about the fact that we are already how i or j outside in the calling function.

(Refer Slide Time: 12:07)

Defining functions

- A function must be defined before it is invoked
- This is OK
- This is not

```
def f(x):  
    return(g(x+1))  
  
def g(y):  
    return(y+3)  
  
z = f(77)
```

Diagram illustrating function definitions and invocations. The left side shows a valid sequence: defining `f(x)` which calls `g(x+1)`, then defining `g(y)`, and finally invoking `z = f(77)`. Green arrows show the flow of execution from the invocation to the definition of `f`, and then to the definition of `g`. The right side shows an invalid sequence: defining `f(x)` which calls `g(x+1)` (marked with red question marks), then invoking `z = f(77)`, and finally defining `g(y)`. A red arrow points from the invocation to the definition of `f`, but the definition of `g` is not reached until after the invocation, which is incorrect.

One of the things that we mentioned up front was that a function must be defined before it is invoked. Now this is a slightly subtle point, so let us just look at it little more. Remember that a Python program is read from top to bottom by the interpreter. So, when the Python program is read it reads the definition of `f`, but does not execute it, and notice that this definition of `f` has an invocation to `g` which is actually later.

But the point is when reading definition of `f` `g` is not used it is only remembered that this statement which should be in a bracket, just to be consistent. So, this statement should be computed if I call `f` so it is not calling `f` it is just defining `f`. So, I define `f`, then I define `g`, finally when I come to this statement it says what is `f` of 77. So, `f` of 77 will come here and we will say `f` of 77 is nothing but `g` of 78 right so that will come here. And it will say `g` of 78 is nothing but 81 so it will come here. So 81. And then finally I will get 81.

So, it is only when I execute the statement `f` is executed at that time `g` has already been

seen. Though we say a function must be defined before it is invoked it does not rule out the fact that one function can call a function which is defined after it, provided that you use this function only after **that** definition. So this sequence is fine. Suppose, we rewrote this sequence in a different way, so supposing we had the definition of f then we had this statement. We **have** basically **exchanged** these two statements.

Now what happens is that when the Python interpreter comes down this line at this point it will try and call f, so f will try and call g and g will say well I do not have a definition for g **yet** because I am not yet gone **past** this **statement**. So if I put this statement, execute f before I define g and f requires g then this statement will create an error **whereas** this statement will not.

It's really useful if we define all functions upfront because any inter dependency between functions will be resolved right way by the interpreter and we do not have to worry about it. Whereas, if we do this inter mixing of functions and statements then we have to be careful that functions do not refer to the **later** things which **have not been** scanned **yet** by **the interpreter**. This is one more reason to put all your function definitions at the beginning and only then have the statements that you want to execute.

(Refer Slide Time: 14:40)

Recursive functions

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$$
$$0! = 1 \quad (n-1)!$$

- A function can call itself — **recursion**

```
def factorial(n):  
    if n <= 0: Base Case  
        return(1)  
    else:  
        val = n * factorial(n-1)  
        return(val)
```

Execution trace for **factorial(3)**:

- factorial(3)** calls **factorial(2)** (3 * factorial(2))
- factorial(2)** calls **factorial(1)** (2 * factorial(1))
- factorial(1)** calls **factorial(0)** (1 * factorial(0))
- factorial(0)** returns 1

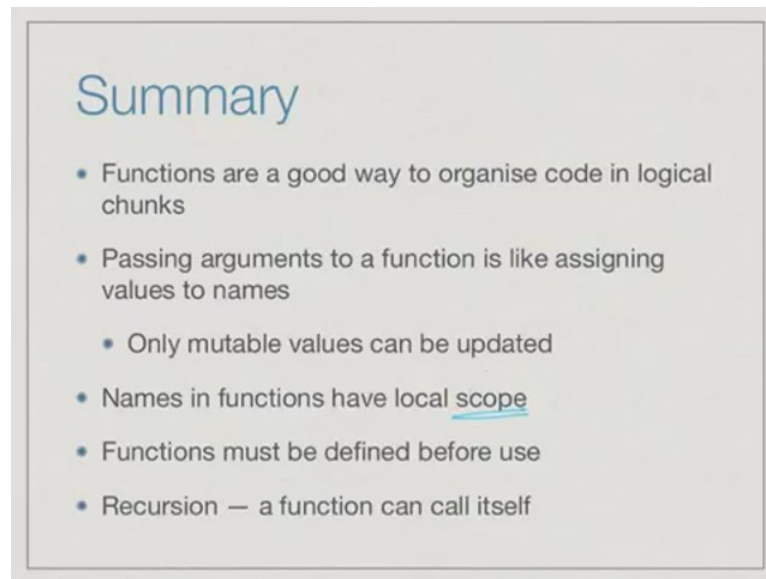
A final point that we will return to later when we go through more interesting examples as we proceed in programming, is that a function can very well call itself. The most **canonical** function of this kind, these are called Recursive functions. Functions which rely on themselves. Is the factorial function. If you remember n factorial is defined to be n into n minus 1 into n minus 2 into n down to 1. So you take n and multiply it by all the numbers **smaller than itself** up to 1 and by definition 0 factorial is defined to be 1.

What we observe in this definition is that, this part from n minus 1 to 1 is actually the same as n minus 1 factorial. In other words n factorial can be defined in terms of a smaller factorial it is n times n minus 1 factorial, so that is what this function is exploiting. There is a base case factorial of 0 is 1 and since the factorial of negative numbers is not defined and we want to be safe we can say that if n is equal to 0 or n is less than equal to 0 we return 1. So this is what we normally call the base case.

In this case the factorial is completely defined without having to do any further work. Now if n is not less than equal to 0 then n is greater than 0. If n is greater than 0 then we take the current number and we **multiply it** by the smaller factorial that is exactly the definition given above. So if I take say factorial of 3, this will result in 3 times factorial of 2 so that will invoke this function again and this will give me 2 times factorial of 1 and so on.

Factorial of 1 will give me 1 times factorial of 0 and the point is **that** factorial 0 will now terminate and it will give me 1, because it says that argument is **less than or** equal to 0 **return** 1. This 1 will return now come back and get multiplied here, so you get 1 times 1, so 1 times 1 will come here and will come here, so then this will bring back 2 and then 3 times 2 this will become 6. This is how the function will execute we will talk about this more later, but just to illustrate that functions can very well call themselves.

(Refer Slide Time: 17:03)



The slide is titled "Summary" in a large, blue, sans-serif font. Below the title is a list of six bullet points, each preceded by a blue dot. The text is in a smaller, dark grey, sans-serif font. The slide has a light grey background with a thin black border.

- Functions are a good way to organise code in logical chunks
- Passing arguments to a function is like assigning values to names
 - Only mutable values can be updated
- Names in functions have local scope
- Functions must be defined before use
- Recursion — a function can call itself

To summarize, functions are a good way to organize your code into logical chunks. So if you have a unit of computation which is done repeatedly and very often done with different possible starting values then you should push it aside into a function. If you break up your code into smaller functions, it is much easier to understand, to read and to maintain. When we pass arguments to a function it is exactly like assigning values to a name.

So, the values that are passed can get updated in a function only if they are mutable, if they are immutable any change within a function does not affect the argument outside. Also if we use the same name inside a function as is found outside a function the name inside the function does not in any way **affect** the name outside. So, functions have local notion of what we call scope. There is a scope of a name where is a name understood, so the name inside a function does not exist outside and vice **versa**.

Also functions must be defined before they are used and this is a good reason to push all your function definitions to the beginning of your program, so that the Python interpreter will digest them all before there are actually **invoked**. So if there are mutual dependencies we do not have a problem. Finally, we saw that we can write interesting functions which call themselves and we will see many more examples of this in the

weeks to come.