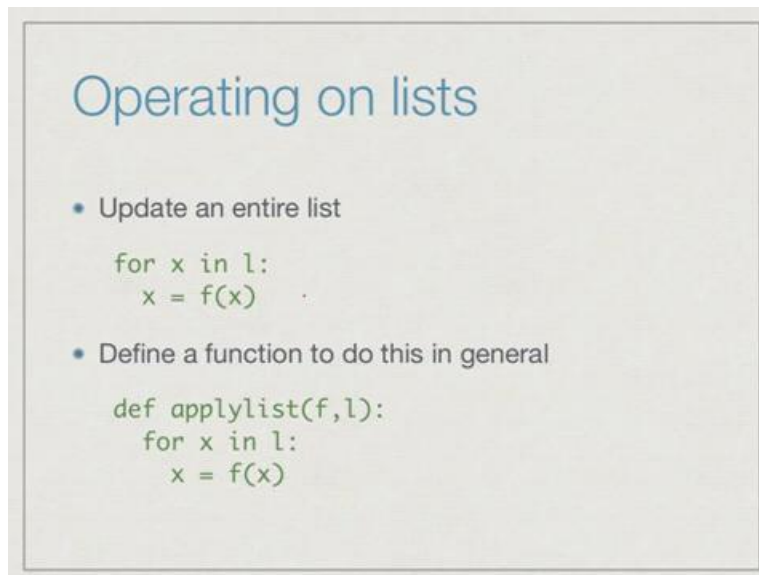**Programming, Data Structures and Algorithms in Python**
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 04
Lecture - 07
List Comprehension

Quite often, we want to do something to an entire list.

(Refer Slide Time: 00:02)
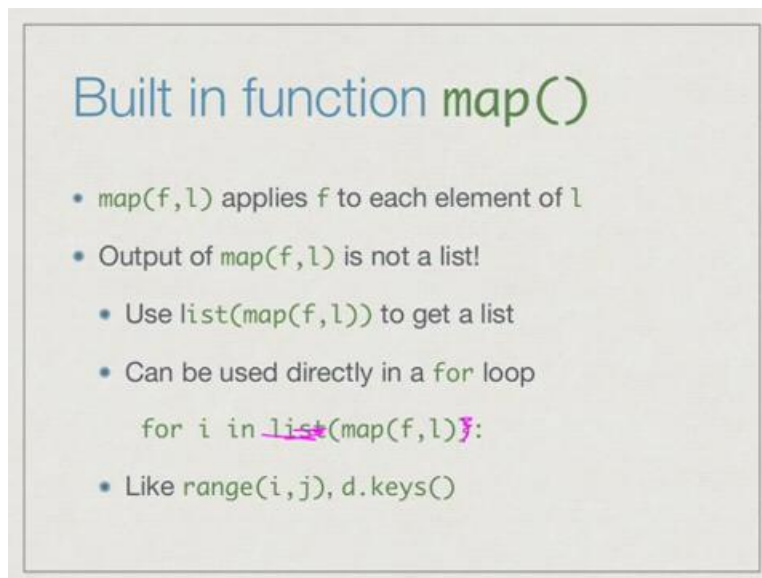
For instance, we might want to replace every item in the list by some derived value f of x. So, we would write a loop as follows, for every x in it, replace x by f of x. Now, we could write a function to do this, which does this for different lists and different values of l. We could say, define apply list, which takes the function f and the list l, and for every x and l, you just replace this x by f of x; and since l, list is a mutable item, this will update list in the calling function as well.

Python has a built-in function map, which does precisely this. So, map f l applies f, in turn to each element of l. Now, although you would think that, if you take a list, say, x 1, x 2, and you apply map, and you get f of x 1, f of x 2, that the output of map should be another list, unfortunately, in python 3, and this is another difference between python 3 and python 2, the output of map is not a list. So, you need to use the list function like we did before. So, you need to say list of map f l to get a list, and you can however, use the output of map directly in a for loop, by saying, for i in list map f l or you can even say for i in map f l, this will work.

So, you do not need to use the list notation, if you just wanted to index menu, but if you want to use it as a list, you must use the list function to convert it. And, this is pretty much what happens, with functions like range and d dot keys and so on. These are all things which give us sequences of values. These sequences are not absolutely lists; they can be used in for functions but if you want to use them as lists, and manipulate them as lists, you must use list to convert them from their sequence to the list form.

Another thing that we typically want to do is to take a list and extract values that satisfy a certain property. So, we might have a list of integers called number list, and from this, we might want to extract the list of primes. We start off by saying that, the list of primes we want is empty, and we run through the number list, and for each number in that list, we apply the test, is it a prime; if it is a prime, then we append the list to our output list. So, we start with a list x 1, x 2 and so on.

And then, we apply the test and some of them will pass, and some of them will succeed, some of them will fail, and at the end, wherever the things pass, those items will emerge in the output. So, in general, we could write a select function which takes the property and a list, and it creates a sub list by going through every element in the list, checking if the property holds, and for those elements which the property holds, appending it to the sub list. The difference between select and our earlier map function is that, property is not an arbitrary function; it does not manipulate l at all, all it does is, it checks whether the property is true or not. The property will be a function which takes an element in the list, and tells us true or false; if it is true, it gets copied to the output; if it is false, it gets discarded.

Built in function `filter()`

- `filter(p,l)` checks p for each element of l
- Output is sublist of values that satisfy p

There is a built-in function for this as well. It is called filter. So, filter takes a function p, which returns true or false for every element, and it pulls out precisely that sublist of l, for which every item in l, which falls into the sublist satisfies p. Let us look at a concrete example.

Combining map and filter

- Sum of squares of even numbers from 0 to 99

```
list(map(square,filter(iseven,range(100)))

def square(x):
    return(x*x)

def iseven(x):
    return(x%2 == 0)
```

Supposing, we have the list of numbers from 0 to 99. We want to first pull out only the even numbers in the list. That is a filter operation; and then, for each of these even numbers, we want

to square them. So, here, we take the even numbers, right, by using the filter, and then, we map square. Then, we get a list.

And then, of course, having got this list, then we can add it up. The sum is not the part of this function. If we want to first extract the squares of the even numbers, and that can be done using a combination of filter, and then, map. Filter, first gives us the even numbers and then map gives us the squares and the square is defined here and this even is defined here.

(Refer Slide Time: 04:46)

There is a very neat way of combining map and filter, without using that notation. Let us get to it, through a simpler mathematical example. So, you might have studied in school, from right hand, right angled triangles that, by Pythagoras' theorem, you know that, if x, y and z are the lengths of the two sides and the hypotenuse respectively, then, x square plus y square will be z square. So, Pythagorean triple is a set of integers, say 3, 4 and 5, for example, such that, x square plus y square is z square; 3 square is 9; 4 square is 16; 5 square is 25. Let us say, we want to know all the integer values of x, y and z, whose values are below n, such that, x, y and z form a Pythagorean triple instance.

In conventional mathematical notation, you might see this kind of expression. It says, give me all triples x, y and z, such that this bar stands for such that; such that, x, y and z, all lie between 1

and n. And, in addition, x square plus y square is equal to z square. This is, in some sense, where we get the values from; this is an existing set. We have x ranging from 1 to n, y ranging from 1 to n, z ranging from 1 to n, and we put together all possible combinations, then we take out those combinations to satisfy a given property, x square plus y square is equal to z square, and those are the ones that we extract out.

In set theory, this is called set comprehension. This is the way of building a new set by applying some conditional things to an old set. This is also implicitly applying a kind of a tripling operator; it takes 3 separate sets, x from 1 to n, y from 1 to n, z from 1 to n, combines them into triples. There is a filtering process by which you only pull out those triples, where x square plus y square is z square; and then, there is a manipulating step, where you combine them into a single triple, x comma y comma z.

But, in general, the main point is that you are building a new set from existing sets. So, what python does and many other languages also, from which python is inspired to, is allow us to extend this notation to lists. This actually comes from a style of programming called functional programming, which, from where this kind of a notation is there, and python has borrowed it and it works quite well.

Here is how you will write our earlier thing, which we had said, the squares of the even numbers below 100. Earlier, we had given a map filter thing. So, we had said, we will take a range, and we will filter it progressively, and then, we would do a map of square. In python, there is an implicit perpendicular line below, before the 'for' from the set notation. It just takes a square of x, for i in range 100, such that, iseven of x - we have here 3 parts. So, we have a generator, which tells us where to get the values from. Remember that, list comprehension or set comprehension, pulls out values from an existing set of lists, so we first generate a list. In this case, the list range 100, but we could use our other lists; we could use for i in any one, just like a 'for'.

Then, we will apply a filter to it, which are the values in this list, which you are going to retain. And then, for each of those values we can do something to it. In this case, we squared and that will be our output. This is how we generate a list using map and filter without using the words map and filter in between, you just use the 'for' for the generator, 'if' for the filter, and the map is implicit by just applying a function to the output of the generator in the filter.

(Refer Slide Time: 08:52)

Let us go back to the Pythagorean triple example. We want all Pythagorean triples with x, y, z below 100. This, as we said, requires us to cycle through all values of x, y, and z in that range. it is a little bit more complicated than the one we did before, where we only had a single generator, all the values in range 0 to 100. It is simple enough to write it with multiple forms. So, we say, I want x comma y comma z, for x in range 100, for y in range 100, for z in range 100, provided, x squared plus y squared is equal to z squared. That is written with the 'if'. Now, just to fit on the slide, I have split it up into multiple lines, but actually, this will be a single line of python code.

In what order will these be generated? Well, it will behave exactly like a nested loop. Imagine, we had written a loop in which we had said, for x in range 100, for y in range 100, for z in range 100, and so on. So, what happens here is that, first - a value of 0 will be fixed for x, and then a value of 0 will be fixed for y, then 0 for z. In the first pair, triple that comes out is 0, 0, 0. Then, the value of z will change, the innermost loop changes next. The next one will be 0, 0, 1. This is x, this is y, this is z.

So, in this way, we will keep going until it will be 0, 0, 99. So, when this hits 99, then this for loop will exit and we go to 1. So, I will get 0, 1, 0, and to 0, 1, 90, 0, 1, 99, and so on. The innermost for, so z will cycle first, then y, and then x will cycle slowest. So, just remember that.

Let us see how this works in python. Let us first begin by defining square; a square of x return x times x; then, we can define iseven x, to check that the remainder of x divided by 2 is 0. So, we have square 8, 64; iseven 67 should be false; iseven 68 should be true and so on. Now, we have list comprehension. Let us look at the set of square x, for x in range 100, such that x is even. So, we see now that, 0 is there. So 0 square, 2 square, 4 square, 6 square, and so on. This is our list comprehension.

Now, let us do the Pythagorean triple one. We said, we want x, y, z, for x in range 100, y in range 100, for z in range 100. This is our 3 generators, with the condition that x times x, plus y times y, is equal to z times z. Now, you see a lot of things which have come. In particular, you should see in the early stages somewhere, things which we are familiar with, like 3, 4, 5, and so on. But, you also see some nonsensical figure, that 4, 0, 4.

So, we should probably have done this better, but you will not worry about that but, what I want to emphasize is that, you see things like, say, you see 0, 77, 77, which is a stupid one; but, let us see, for instance, you say, you see 3, 4, 5. So, we saw 3, 4, 5, somewhere - so 3, 4, 5. But, you will also see, later on 4, 3, 5. Now, one might argue that, 3, 4, 5, and 4, 3, 5, are the same triplets. So, how do we eliminate this duplicate?

So, we can have a situation, just like we have in a 'for loop', where the later loop can depend on an earlier loop; if the outer loop says, i to some, i goes from something to something, the later loop can say that, j starts from i, and goes forward. For instance, we can now rewrite our Pythagorean triples to say that, x is in range 100, but y does not start at 0; it starts from x onwards. So, y is never smaller than x, and z is never smaller than y. So, z is also never smaller than x, because y itself is never smaller than x, and this version will actually eliminate duplicates.

(Refer Slide Time: 13:21)

Here is our earlier definition of Pythagoras, where we had x, y, and z unconstrained. So, what I do is, I go back, and I say that, y is not in range 100, but y is in range x to 100, and z is in range y to 100. And now, you will see a much smaller list and in particular you will see that, in every sequence that is generated, x is less than or equal to y is less than equal to z; you only get one copy of things like 3, 4, 5. So, you see 3, 4, 5, but you do not see 4, 3, 5; 3, 4, 5 is here. Next one is 5, 12, 13; 4, 3, 5 is eliminated. The key thing is that, generators can be dependent on outer generators - inner generators can be dependent on outer generators.

(Refer Slide Time: 14:06)

This list comprehension notation is particularly useful for initializing lists, for example, for initializing matrices, when we are doing matrix like computations. Supposing, I want to initialize a 4 by 3 matrix to all zeros. So, 4 by 3 matrix has 4 rows and 3 columns, and I am using the convention that, I store it row-wise. So, I have to store the first row. So, it will be 3 entries for the first row; then, 3 entries for the second row, and so on.

Here is an initialization, which says, l consists of something for the outer things says for, this is for each row. It is something for each row. For 4 rows 0, 1, 2, 3, I do something; and what is that something? I create a list of zeros of that size 3. Each row j, from 0 to 3, consists of columns 0, 1, 2, which are zeros. This will actually generate the correct sequence that we saw at, that would we need to initialize the generators.

(Refer Slide Time: 15:18)


```
>>> l = [ [0 for i in range(3) ] for j in range(4) ]
>>> l
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>>
```

Here is that list comprehension notation for initializing the matrix. So, it says, for every j in range 4, right, we create a list, and that list itself has zero for i in range 3, and if you do this, and look at it, then, correctly it has 3 zeros, and 3 zeros, and 3 zeros, 4 times. These are the four rows.

(Refer Slide Time: 15:37)

Warning

- What's happening here?

```
>>> zerolist = [ 0 for i in range(3) ]
>>> l = [ zerolist for j in range(4) ]

>>> l[1][1] = 7
```
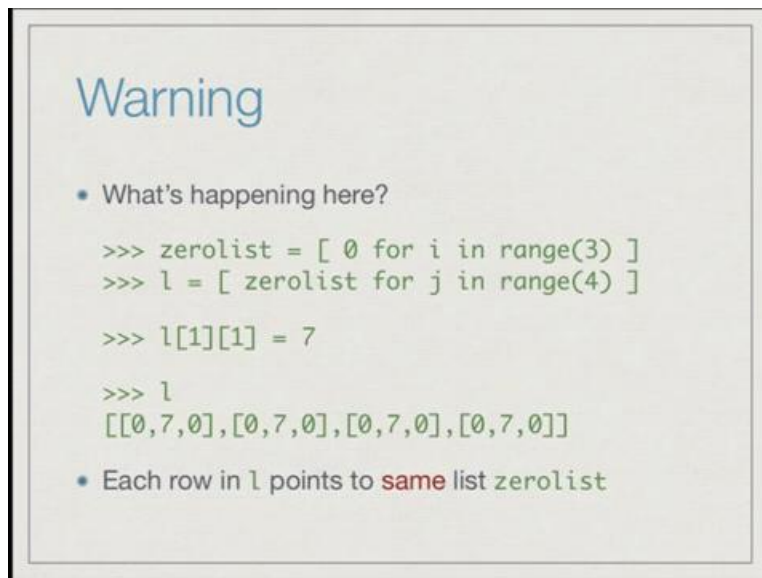
Suppose, instead, we split this initialization into 2 steps; we first create a list of 3 zeroes called zerolist, which says zero for i in range 3. This creates a list of 3 zeros; and then, we copy this list 4 times, in the four rows. We say that the actual matrix l has 4 copies of zerolist. Now, we go and change one entry; say, we change entry 1 in row 1. From the top, it is actually second row. It is the second row, second column, if you want to think in normal terms. So, we take up list 1, which is the second list. Now, what you expect is the output of this.

(Refer Slide Time: 16:22)


```
>>> zerolist = [ 0 for i in range(3) ]
>>> l = [ zerolist for j in range(4) ]
>>> l
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> l[1][1] = 7
>>> l
[[0, 7, 0], [0, 7, 0], [0, 7, 0], [0, 7, 0]]
>>>
```

There we have the zero lists, and then, we say, l is 4 copies of zerolists, for j in range 4. So, superficially, l looks exactly the same. Now, we say l 1 1 is equal to 7, and if you look at l now, we will find that we have not one 7, but 4 copies of 7. This is apparently something that we did not expect.

The output after l 1 1 is equal to 7 is – [0, 7, 0], [0, 7, 0], [0, 7, 0], [0, 7, 0] and why is this happening; well, that is because by making a single zerolist, and then making 4 copies of it, we have effectively created 4 names with the same list. So, whether we access it through l 0 or l 1 or l 2 or l 3, all 4 of them are pointing to the same zerolist. So, any one of those updates would actually update all 4 lists.

If you want to create a 2 dimensional matrix and initialize it, make sure you initialize it in one shot using a nested range, and not in 2 copies like this, because these 2 copies will un-intentionally combine 2 rows into copies of the same thing, and updates to one row will also update another row.
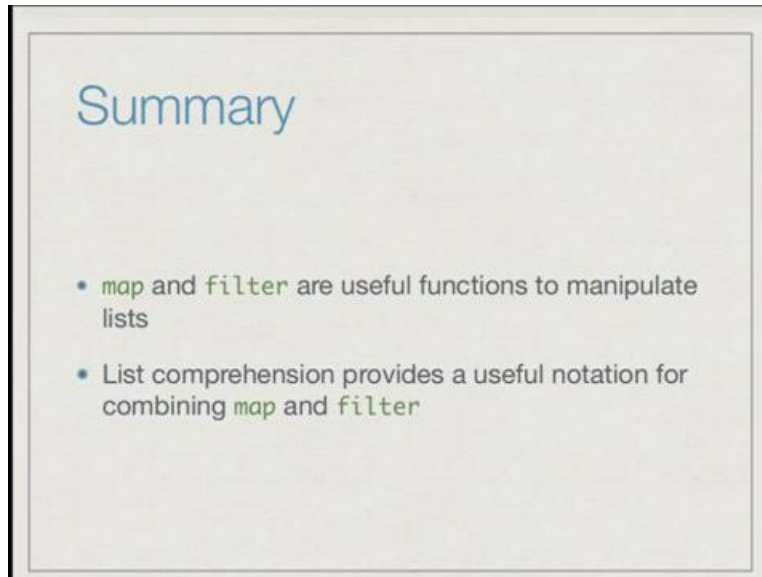
To summarize, map and filter are very useful functions to manipulating lists, and python provides, like many other programming languages, based on the function programming, the notation called list comprehension, to combine map and filter. And, one of the uses that we saw for list comprehension is to correctly initialize 2 dimensional or multi dimensional lists to some default values.