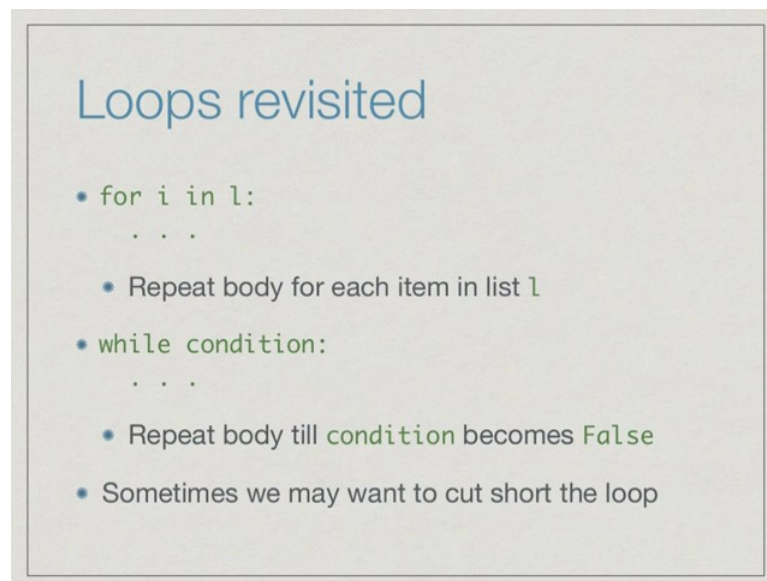


Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 03
Lecture – 03
Breaking out of a loop

(Refer Slide Time: 00:01)



Loops revisited

- `for i in l:`
 . . .
 - Repeat body for each item in list `l`
- `while condition:`
 . . .
 - Repeat body till `condition` becomes `False`
 - Sometimes we may want to cut short the loop

Let us revisit Loops. So, we **saw** that we have two types of loops in Python. The first is a 'for loop' where the value ranges over a list in general or say the output of the range sequence; so `for i in l` will repeat the body for each item in the list `l`.

Sometimes we do not know how many times we need to repeat in advance, so then we use **the** while forms. While takes a condition and it keeps repeating the body until the condition becomes false. But the main thing to notice in both these cases is that how long will the loop takes is determined in advance either we have a fixed number of iterations depending on number of values in `l` for a for loop or we keep going until the condition becomes false we cannot stop before that. Now it does turn out the sometimes we have natural situations where we want to cut short the loop and not proceed till the very end.

(Refer Slide Time: 01:00)

Search for value in a list

```
def findpos(l,v):
    # Return first position of v in l
    # Return -1 if v not in l
    (found,i) = (False,0)
    while i < len(l):
        if not found and l[i] == v:
            (found,pos) = (True,i)
    if not found:
        pos = -1
    return(pos)
```

Suppose, we want to write a function findpos which finds the first position of a value v in the list l. We had seen that there is a built in function called index which pretty much does this, so l dot index v will report the first position where v occurs in the list l.

However, the index function will report an error if there is no occurrence of v in l. So, we want to change that slightly and what we want to say is that if there is no v in l **report** minus 1. So either we get a value between 0 to n minus 1, where n is the length of the list or we get the value minus 1 saying that v is not in the list. Here is a while loop implementation. We use a name found as the name suggest **to indicate** whether the value have been found or not as we **have been seeing** so far. Initially we have not seen any values in l, so v is, the found - is false. And we use i as a name to go through the positions, so i is initially 0 and we will increment i until we hit the length of l.

So, so long as, while i is less than the length of the l, if we see the value we are looking for then we update found to be true and we mark the position that we have found it to be pos. At the end of this, if we have not found the value, so the value found is not been set to true that means there was no v in the list then we will set pos to minus 1 which is the default value we **indicate at** beginning. Then we return the current value of pos which is either the value of pos we found or the value will set to minus 1 because we did not find it.

There are two points to observe about this; the first point which is the main issue at hand is that we are going to necessarily go through every position i from 0 to the length of l minus 1 regardless of where we find it. Supposing, we had several hundreds of thousands of elements in our list and we found the value at the very second position we are still going to scan the remaining hundreds of thousands of positions before we return the position two. Now this seems very unnecessary.

The other point, which is an issue of correctness, this is an issue of efficiency that we are running this loop too many times unnecessarily. In case we are actually able to report the first value quite early. The other problem which is correctness is that we want to report the first position, but the way we have written it every time we continue pass the first position and we find another copy of v we are updating the position to be the new thing.

So, we are actually finding not the first position but the last position so this is not a very good way to do this. So, we first change that. So we say that we want the first position. So, we want the first position we want this update to happen only if we have not found the value so far. We had this extra condition which says that if $l[i]$ is v and we have not found so far then we update found to true and we said pos to i .

This ensures that pos is updated to i only the very first time we see v after that the value found prevents us from doing this update again. So, at least we are correctly finding the first value of the position. And finally as before if we never find it then the value found is never set to true and so we report minus 1. Now the issue is why we have to wastefully go through the entire list even though after we find the very first position of v in l we can safely return and report that position.

(Refer Slide Time: 04:32)

Search for value in a list ...

- A more natural strategy
 - Scan list for value
 - Stop scan as soon as we find the value
 - If the scan completes without success, report -1

So, a more natural strategy suggests the following; we scan the list and we stop the scan as soon as we find the value. So, whenever we find the value for the first time we report that position and we get out. Of course, **the** scan may not find the value, so if the scan completes without succeeding then we will report minus 1.

(Refer Slide Time: 04:58)

Search for value in a list ...

- A more natural strategy

```
def findpos(l,v):  
    for x in l:  
        if x == v:  
            # Exit and report position of x  
  
    # Loop over, report -1 if we did not see x
```

We could think of doing this using a 'for loop'. So, we go through every value **in** **l** using **a** variable **x**. So, **x** is the name which goes through the first element, second element, these are not positions now these are the actual values. We check whether the value **x** that we currently pick up in the list is indeed the value **v** we are looking for, if so we exit and we report the position of this **x**. If we come to the end of the loop and we have not seen **x** so far like before we want to report minus 1.

(Refer Slide Time: 05:30)

Search for value in a list ...

- A more natural strategy

```
def findpos(l,v)  
    (pos,i) = (-1,0)  
    for x in l:  
        if x == v: # Exit, report position of x  
            pos = i  
            break  
            i = i+1  
  
    # If pos not reset in loop, pos is -1  
    return(pos)
```

Here is a first attempt **at doing it**, so let us go through it systematically. First of all we have this loop, so for each x in the list if x is v then we report the position. Now we have to find the position because we are going through the values so this **has** forced us to use a name i to record the position and we have to manually do this. So, like in the while loop we start with the position 0 and then every iteration we increment the position. This is only the first version of this we will see how to fix this. So, we have to separately keep track while we are doing this for, kept, separately keep track of the positions and report it.

But what is new and this is a main issue to be **highlighted** here is this statement called break. So what break does is it exits the loop. So this is precisely what we wanted to do if x is v we have found the first position we do not want to continue we just want to break, if not we will go increment the position and go back. Now, how do we record at the end we do not have this found variable anymore. How do we know at the end whether or not we actually saw it? So, the question is was pos set to i or was it not set to i . Well, the default value is minus 1. Supposing, pos is not reset we want to report minus 1. So, this is why in our function we have actually initially set pos to be minus 1. So, the default value is that we do not expect to find. It is like saying found is false. So, by default the position is minus 1.

At the end of this loop if you have not found it pos **has** not been reset so it remains minus 1. On the other hand if we have found it without looking at all the remaining elements we have set it to the first position we found it and we have taken a break statement to get out of the loop so we do not unnecessarily scan. When we come here either way we have either report it to first position we have found it or in the worst case we have scanned through the entire list, and we have not found it in which case the initial value minus 1 is there. So, in any case we can return pos and we have no problem

This is just to illustrate the use of the word break, which allows us in certain situations to get out. Now remember in the worst case we do not find x in it. So, the worst case behavior of this loop is no different from the situation without the break we have to go through the entire list and we have to come out only when we have scanned everything, but if we do find it we can avoid some unnecessary computations.

(Refer Slide Time: 08:10)

Search for value in a list ...

- A more natural strategy

```
def findpos(l,v)
    pos = -1
    for i in range(len(l)):
        if l[i] == v: # Exit, report position
            pos = i
            break
    # If pos not reset in loop, pos is -1
    return(pos)
```

Handwritten notes on the slide:

- A pink '0' is written above the initial `pos = -1`.
- A red bracket on the left side of the loop indicates the range of indices from 0 to `len(l)-1`.
- A red 'x' is written next to the line `pos = i`.
- Two red question marks '??' are written next to the `break` statement.
- Pink handwritten text on the right says: `break → v is found` and `terminate loop normally → v is not found`.

We can simplify this a little bit by first removing this `i` instead of scanning `x` actually it is better to scan the positions. So, it is better to say `pos` is minus 1, but instead of going through `x` in `l` it is better to go through `i` in the range 0. Remember now this is implicitly 0 to the length of `l` minus 1. So, we go for `i` in the range 0 to length of `l`, so if we do not give a 0 it will give only a one argument this is taken as the upper bound. This will go through all the positions. And instead of checking `x` we check `l` at position `i`, so if `l` at position `i` is `v` then we set the position to this current value and we break. So, by changing the variable that we put in the 'for', we have got a slightly more natural function.

And again, we have this clever trick which says that since we set `pos` initially to minus 1 if we did not reset it here if no value in the range 0 to `n` minus 1 produced a list value which is equal to the value **we are looking** for we will return minus 1 as before. This requires this clever trick. So the question is what if we do not have a situation where such a clever trick is possible or if we do not think of this clever trick how would we know at this point. So remember now there are two situations either we break, so if we break that means that the value is found, or if we do not break, if we terminate normally, if the loop ends by going through all the things then `v` is not found.

Remember even if *v* is found at very last position we will first break, we will not go back and say that the loop ended. So, if we see *v* at any position from beginning to end we will execute the break statement if not we will not. The question is can we detect whether or not we broke out this loop or whether we terminate it separately.

(Refer Slide Time: 10:06)

Search for value in a list ...

- A loop can also have an `else:` — signals normal termination

```
def findpos(l,v)
    pos = -1
    for i in range(len(l)):
        if l[i] == v: # Exit, report position
            pos = i
            break
    else:
        pos = -1 # No break, v not in l
    return(pos)
```

The slide includes handwritten annotations: a red arrow points to `pos = -1` with a red 'X' above it; a green arrow points down from the `break` statement to the `else:` block; and a red arrow points to the `pos = -1` line in the `else:` block. The `else:` block is also highlighted with a red box.

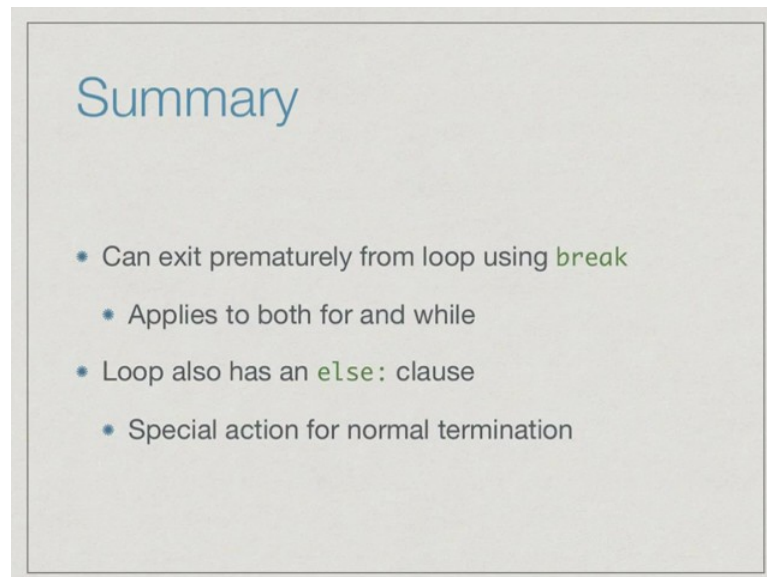
Python is one of the few languages which actually provide way to do this. So it allows something that looks a bit odd because of the name it allows something called `else` which we saw with `if`, it allows an `else` for a loop as well. The idea is that `else` this part will be executed if there is no break if the loop terminated normally, so do not worry about the fact that `else` does not mean this in English so it is just **a way of** economizing or the number of new words you need to use. If you see an `else` attached to a `'for'` it **could** also **be** attach to a **'while'** it means that the `'while'` or the `'for'` terminated in the natural way either for iterated through every value that it was supposed to iterate through or the **while** condition became false. On the other hand aborted by a `break` statement then the `else` will not be executed.

Here for instance, now we do not initialize. We no longer have this clever trick, we do not have this anymore. So what we say is that for *i* this range we check and if it is there we set the position to be the current *i* and then we break.

Now, if we have actually gone through the entire list and not found it `pos` is undefined. If `pos` is undefined we need to define it before we return the value, so we have this `else`

statement. Now we say, we have come through this is whole thing and there will be no break, there is no `v` in `l` because otherwise we would have done a break and otherwise `pos` will be set to a valid value in the range 0 to `n` minus 1. So, there has been no break there is no `v` in `l`. Let us say `pos` to minus 1 and then return it.

(Refer Slide Time: 11:44)



Summary

- Can exit prematurely from loop using `break`
- Applies to both `for` and `while`
- Loop also has an `else:` clause
- Special action for normal termination

To summarize, it is very often useful to be able to break out of a loop prematurely. We can do this for both `for` and `while` we can execute the `break` statement and get out of a loop before **its** natural termination. And sometimes it is important to know why we terminate it, we terminate **it** because the loop ended naturally or because we used the `break` statement. This can be done by supplying the optional `else`. Both, `for` and `while` also allow an `else` at the end, and the statement within `else` is executed only when loop terminates normally.