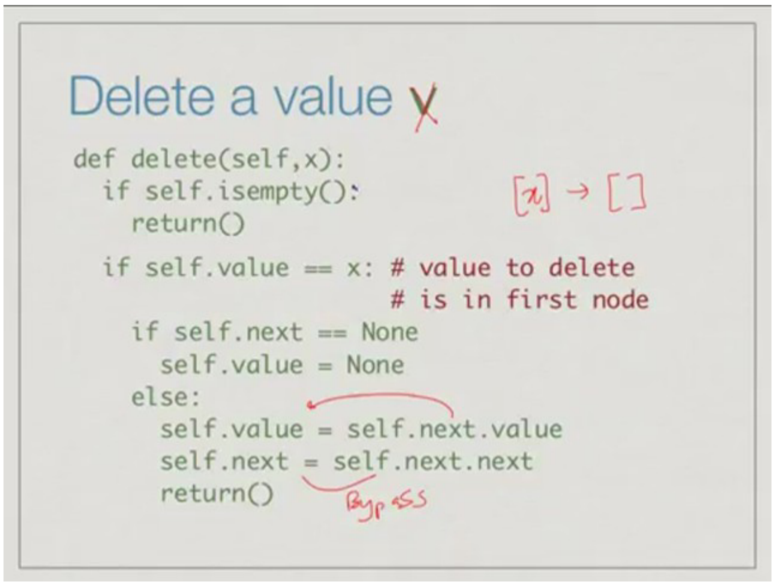
(Refer Slide Time: 14:07)



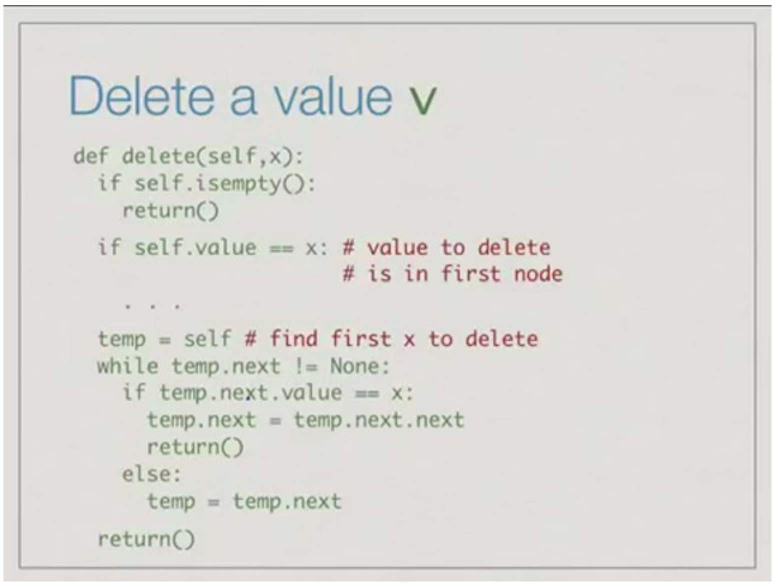
Delete a value x

```
def delete(self,x):
  if self.isempty():
    return()
  if self.value == x: # value to delete
                      # is in first node
    if self.next == None
      self.value = None
    else:
      self.value = self.next.value
      self.next = self.next.next
      return()
```

Here is a part of the delete function. First of all, if we were looking for v and then we do not find it. So, sorry in this code, it is called x. So, this is deleting value x if you want. If we say that the list is empty, then obviously, we cannot delete it because delete says if there is a value of this... node with value x then delete it. If it is empty we do nothing; otherwise if this self dot value is x the first node is to be deleted. Then if there is only 1 node, then we are going from x to empty, this is easy. If there is no next node right, if we have only a singleton then we just set the value to be none and we are done.

This is the easy case, but if it is not the first node, I mean, it is the first node and this is not also the only node in the list then what we do is we do what we said before. We copy the next value. We pretend that we are deleting the second node. So, we copy the second value into the first value and we delete the next node by bypassing. This is that bypass. This is part of the function; this is the tricky part which is how do you delete the first value. If it is only 1 value, make it none; if not, bypass the second node by copying the second node to the first node.

(Refer Slide Time: 15:24)

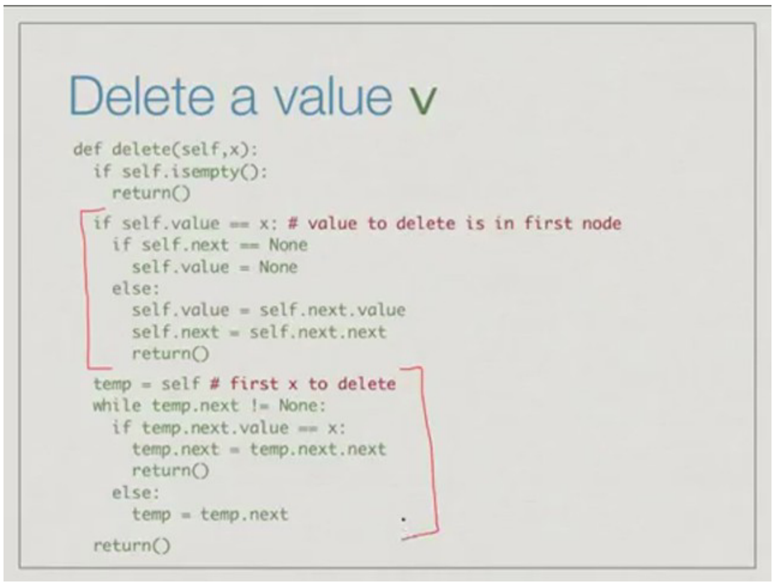


Delete a value v

```
def delete(self,x):
  if self.isempty():
    return()
  if self.value == x: # value to delete
                      # is in first node
    . . .
  temp = self # find first x to delete
  while temp.next != None:
    if temp.next.value == x:
      temp.next = temp.next.next
      return()
    else:
      temp = temp.next
  return()
```

And if this is not the case then we just walk down and find the first x to delete. We start as… this is like our iterative append. We start pointing to self and so long as we have not reached the end of the list if we find the next value is x and then we bypass it and if you reach the end of the list, we have not found it, we do nothing, we just have to return. In this case it is not like append where when we reached the end of the list we have to append here, if we do not find a next by the time we reach the end of the list, then there's nothing to be done.

(Refer Slide Time: 15:54)

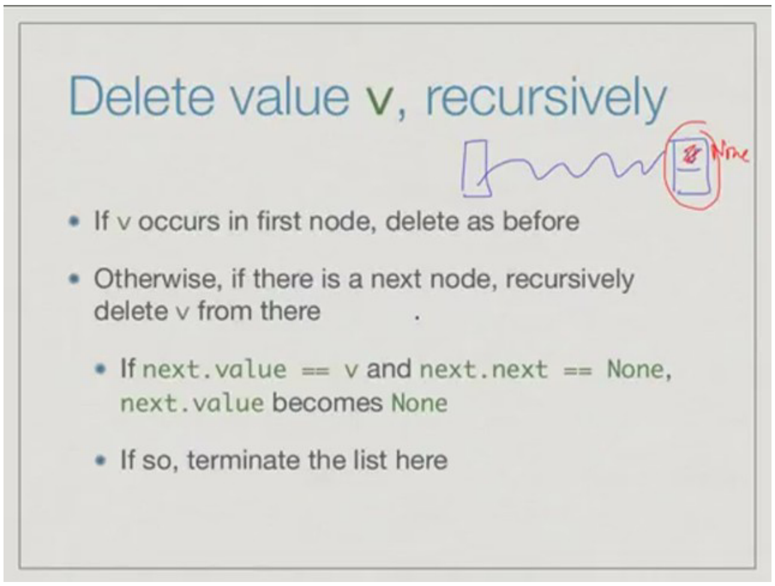


Delete a value v

```
def delete(self,x):
  if self.isempty():
    return()
  if self.value == x: # value to delete is in first node
    if self.next == None
      self.value = None
    else:
      self.value = self.next.value
      self.next = self.next.next
      return()
  temp = self # first x to delete
  while temp.next != None:
    if temp.next.value == x:
      temp.next = temp.next.next
      return()
    else:
      temp = temp.next
  return()
```

So, just for completeness, here is the full function, this was the first slide we saw which is the case when the value to be deleted is in the first node and this is the second case when we walk down the list looking for the first x to delete.

(Refer Slide Time: 16:09)



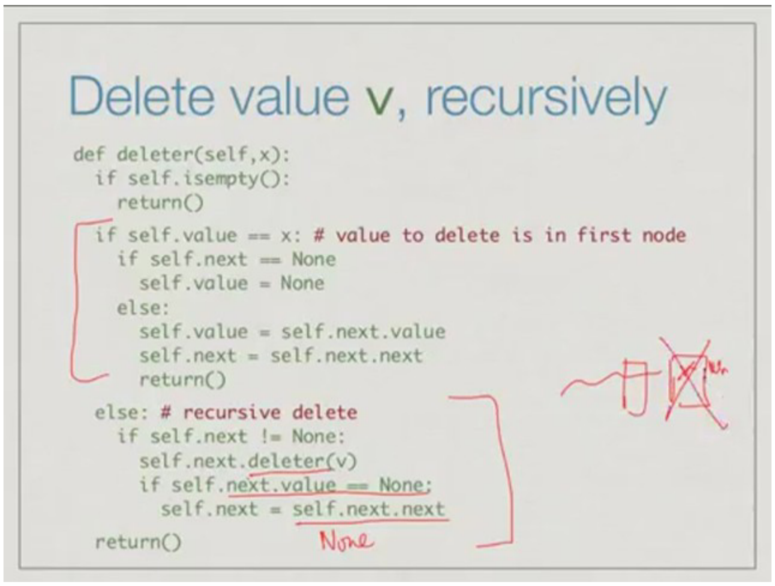
Delete value v, recursively

- If v occurs in first node, delete as before
- Otherwise, if there is a next node, recursively delete v from there
  - If `next.value == v` and `next.next == None`, `next.value` becomes `None`
  - If so, terminate the list here

Just like append can be done both iteratively and recursively, we can also delete recursively which is if it is the first node we handle it in a special way by moving the second value to the first and bypassing it as we did before. Otherwise we just point to the next node and ask the next node, the list starting at the next node, what is normally called the tail of the list, to delete v from itself. The only thing that we have to remember in this is that if we reach the end of the list and we delete the last node. Supposing it turns out, the value v to be deleted is here. So, we come here and then we delete it. What we will end up with is finding a value none, because when we delete it from here, it is as though we take a singleton element v and delete v from a singleton and will create none none. So, this is the base case, if we are recursively deleting as we go whenever we delete from the last node, it is as though we are deleting from a singleton list with value v and we are not allowed to create a value none at the end.

We have to just check when we create the next thing if we delete the next value and it is value becomes none then we should remove that item from the list. So, this is the only tricky thing that when we do a recursive delete you have to be careful after we delete you have to check what is happening.

(Refer Slide Time: 17:32)



Delete value v, recursively

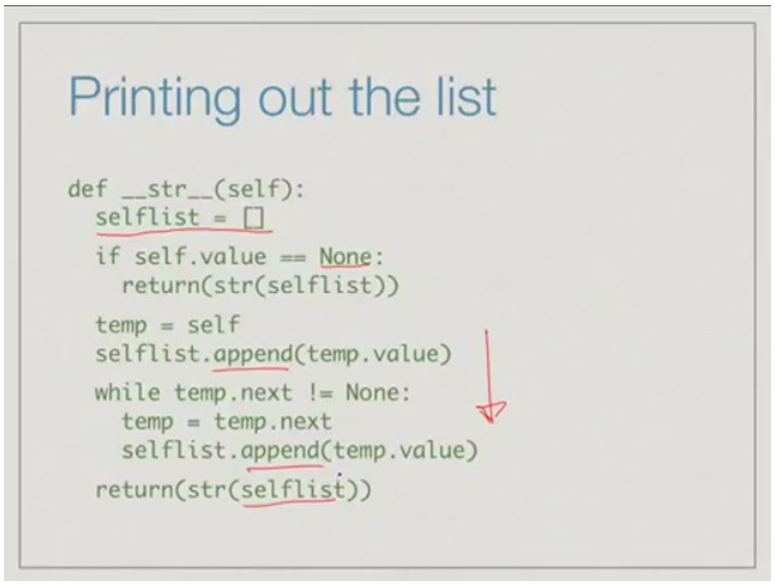
```
def deleter(self,x):
  if self.isempty():
    return()
  if self.value == x: # value to delete is in first node
    if self.next == None
      self.value = None
    else:
      self.value = self.next.value
      self.next = self.next.next
    return()
  else: # recursive delete
    if self.next != None:
      self.next.deleter(v)
      if self.next.value == None:
        self.next = self.next.next
  return()
```

This part is the earlier part and now this is recursive part. So, recursive part is fairly straight forward. So the first part is when we delete the first element from a list, but the recursive part we check if self dot next is equal to none then we delete recursively that is fine. So, this is the delete call.

Now, after the delete is completed we check whether the next value has actually become none. Have we actually ended up at the last node and deleted the last node? If so, then we remove it, this we can either write self dot next is equal to self dot next dot next or we could even just write self dot next is equal to none which is probably a cleaner way of saying it because it can only happen at the last node. So, you make this node the last node. Remember if the next node is none, it's next must also be none.

This has the same effect: self dot next dot next must be none. So, we can also directly assign self dot next is equal none and it would basically make this node the last node. The only thing to remember about recursive delete is when we reach the end of the list and we have deleted this list this becomes none then we should terminate the list here and remove this node.

(Refer Slide Time: 18:34)



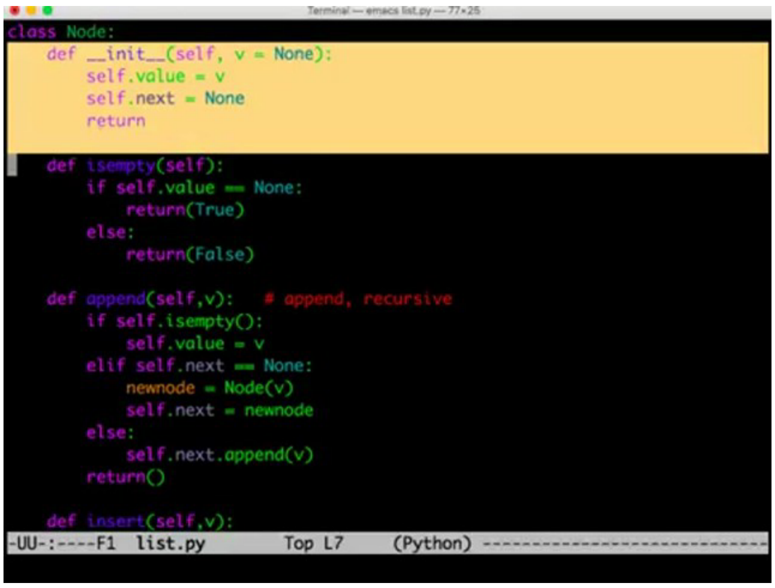
Printing out the list

```
def __str__(self):
  selflist = []
  if self.value == None:
    return(str(selflist))
  temp = self
  selflist.append(temp.value)
  while temp.next != None:
    temp = temp.next
    selflist.append(temp.value)
  return(str(selflist))
```

Finally let us write a function to print out a list. So, that we can keep track of what is going on. We will print out a list by just constructing a python list out of it and then using str on the python list. So, we want to create a python list from the values in our list. So, we first initialize our list that we are going to produce for the empty list.

If our list, the node itself has nothing then we return the string value of the empty list, otherwise we walk down the list and we keep adding each value using the append function. So, we keep appending each value that we have stored in each node building up a python list in this process and finally, we return whatever is the string value of that list. Let us look at some python code and see how this actually works.

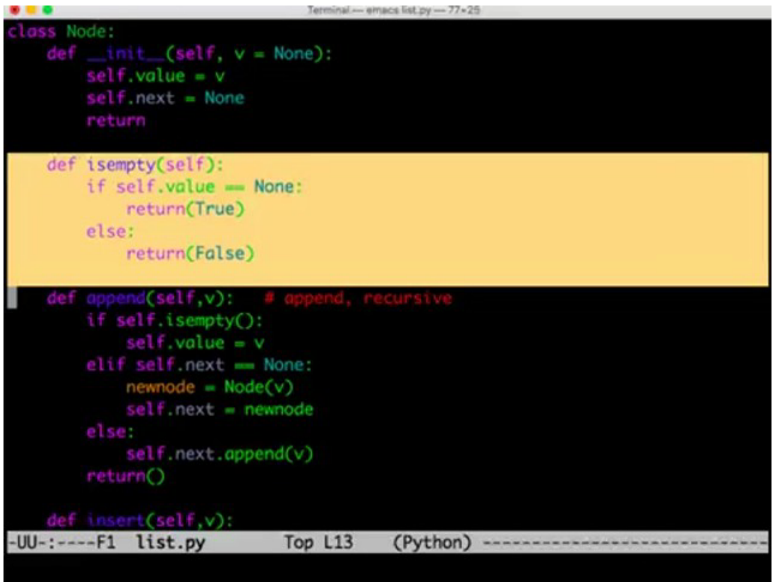
(Refer Slide Time: 19:24)



```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)

    def append(self,v):   # append, recursive
        if self.isempty():
            self.value = v
        elif self.next == None:
            newnode = Node(v)
            self.next = newnode
        else:
            self.next.append(v)
        return()

    def insert(self,v):
-UU-:----F1  list.py        Top L7     (Python) ----------------------------
```

Here we have code which exactly reflects what we did in the slides. We have chosen to use the recursive versions for both append and delete. So, we start with this initial initialization which sets the initial value to be none by default or otherwise v as an argument provided.

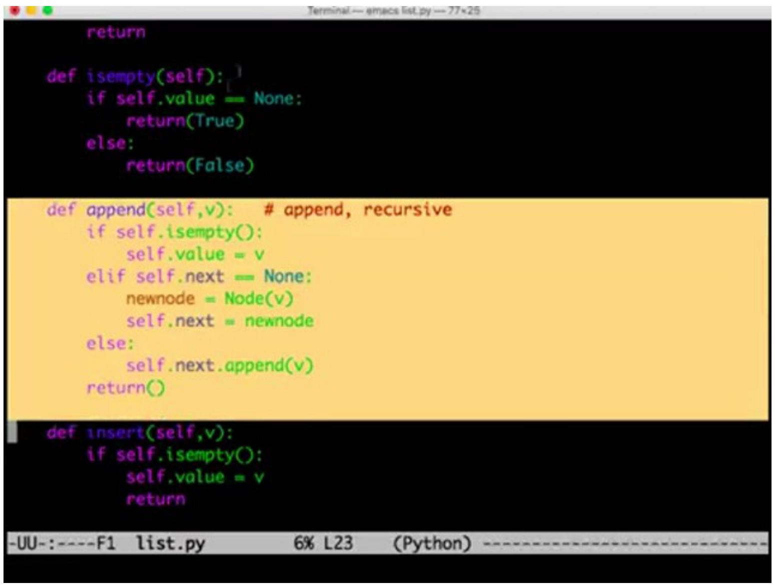
(Refer Slide Time: 19:44)



```
class Node:
    def __init__(self, v = None):
        self.value = v
        self.next = None
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)

    def append(self,v):   # append, recursive
        if self.isempty():
            self.value = v
        elif self.next == None:
            newnode = Node(v)
            self.next = newnode
        else:
            self.next.append(v)
        return()

    def insert(self,v):
-UU-:----F1  list.py        Top L13    (Python) ----------------------------
```

Then isempty just checks whether self dot value is none, we had written a more compact form in the slide by saying just return self dot value equal to equal to none, but we have expanded it out as an if statement here.

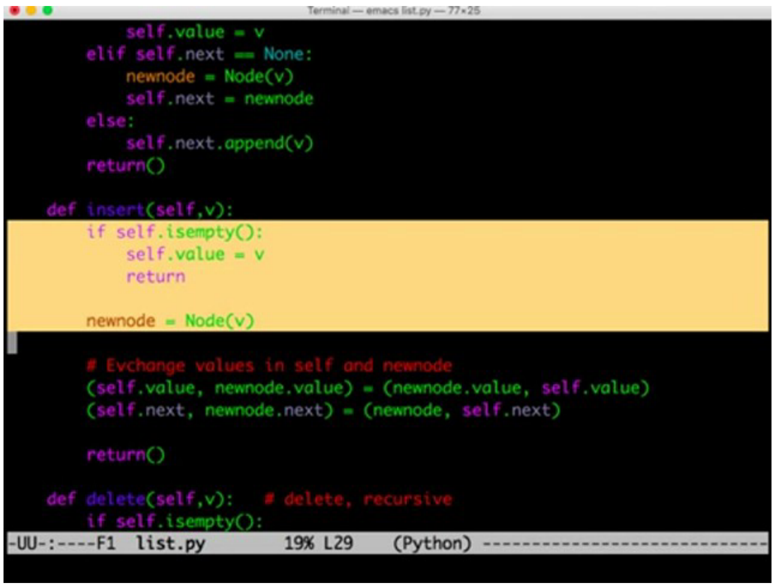
(Refer Slide Time: 19:56)



```
        return

    def isempty(self):
        if self.value == None:
            return(True)
        else:
            return(False)

    def append(self,v):   # append, recursive
        if self.isempty():
            self.value = v
        elif self.next == None:
            newnode = Node(v)
            self.next = newnode
        else:
            self.next.append(v)
        return()

    def insert(self,v):
        if self.isempty():
            self.value = v
            return
-UU-:----F1  list.py        6% L23    (Python) ----------------------------
```

Now, this is the append function. So, append just checks if the current node is empty then it puts it here otherwise it creates a new node... if we have reached the last node it creates a new node and makes the last node point to the new node, otherwise it recursively appends. Then we have this insert function here.

(Refer Slide Time: 20:29)

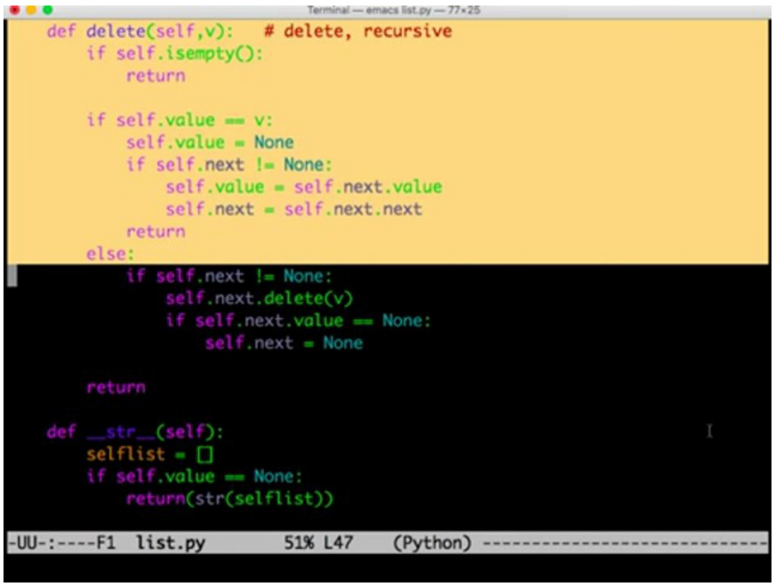


Terminal — emacs list.py — 77×25

```
            self.value = v
        elif self.next == None:
            newnode = Node(v)
            self.next = newnode
        else:
            self.next.append(v)
        return()

    def insert(self,v):
        if self.isempty():
            self.value = v
            return

        newnode = Node(v)

        # Evchange values in self and newnode
        (self.value, newnode.value) = (newnode.value, self.value)
        (self.next, newnode.next) = (newnode, self.next)

        return()

    def delete(self,v):    # delete, recursive
        if self.isempty():
-UU-:----F1  list.py        19% L29     (Python) ----------------------------
```

This insert function: again if it is empty then it just creates a singleton list otherwise it creates a new node and exchanges the first node and the new node. So, this particular thing here is the place where we create this, swap the pointers so that what self points to does not change, but rather we create a reordering of the new node and the first node. So, the new node becomes the second node and the first node now has the value that we just added.

(Refer Slide Time: 21:02)

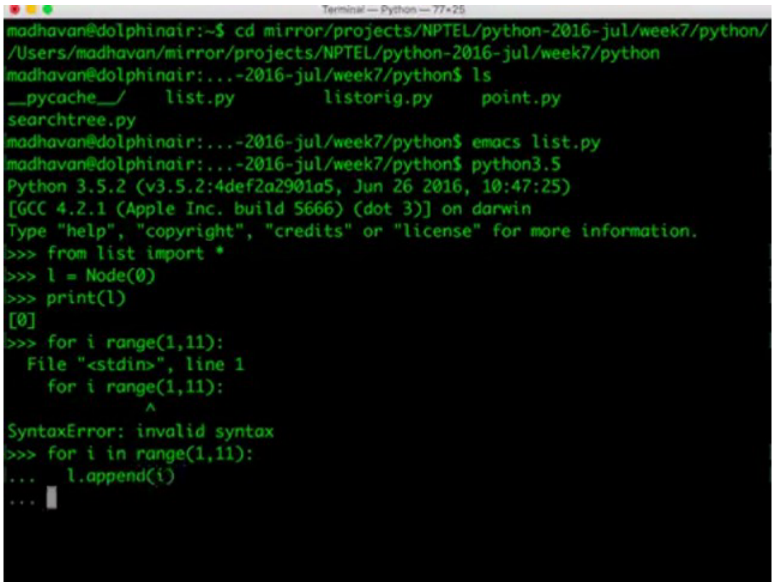


```
    def delete(self,v):   # delete, recursive
        if self.isempty():
            return

        if self.value == v:
            self.value = None
            if self.next != None:
                self.value = self.next.value
                self.next = self.next.next
            return
        else:
            if self.next != None:
                self.next.delete(v)
                if self.next.value == None:
                    self.next = None

        return

    def __str__(self):
        selflist = []
        if self.value == None:
            return(str(selflist))

-UU-:----F1  list.py        51% L47    (Python) ---------------------------
```

Finally, we can come down to the recursive delete. So, the recursive delete again says that if the list is empty then we do nothing, otherwise if the first value is to be deleted then we have to be careful and we have to make sure we delete the second value by actually copying the second node into the first and finally, if that is not the case then we just recursively delete, but then when we finish the delete, we have to delete the spurious empty node at the end of the list in case we have accidentally created it.

So, these 2 lines here just make sure that we do not leave a spurious empty node at the end of the list. And finally, we have this str function which creates a python list from our values and eventually returns a string representation of that list.

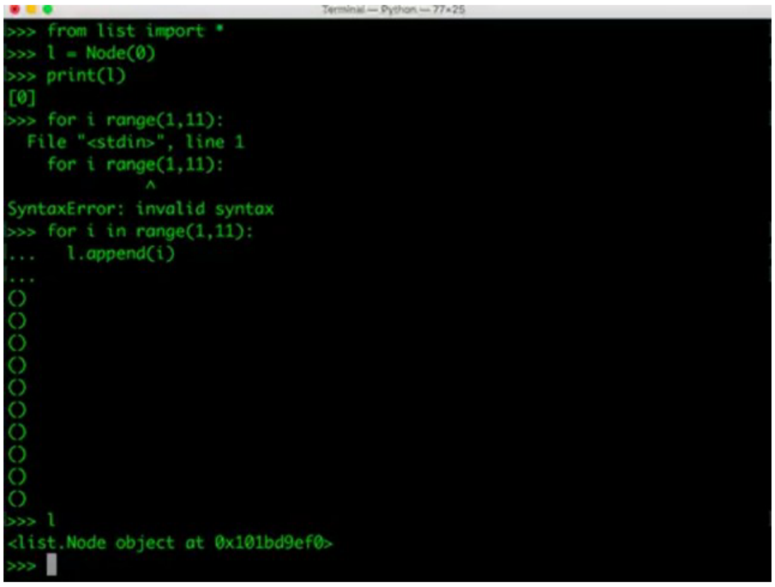
(Refer Slide Time: 21:54)



```
madhavan@dolphinair:~$ cd mirror/projects/NPTEL/python-2016-jul/week7/python/
/Users/madhavan/mirror/projects/NPTEL/python-2016-jul/week7/python
madhavan@dolphinair:...-2016-jul/week7/python$ ls
__pycache__/    list.py         listorig.py     point.py
searchtree.py
madhavan@dolphinair:...-2016-jul/week7/python$ emacs list.py
madhavan@dolphinair:...-2016-jul/week7/python$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from list import *
>>> l = Node(0)
>>> print(l)
[0]
>>> for i range(1,11):
  File "<stdin>", line 1
    for i range(1,11):
              ^
SyntaxError: invalid syntax
>>> for i in range(1,11):
...    l.append(i)
...
```

If we now run this by importing, then we could say, for instance, that l is a list with value 0 and if we say print l then we will get this representation 0, we could for instance put this in a loop and say for i in range 1 say 11, l dot append i.

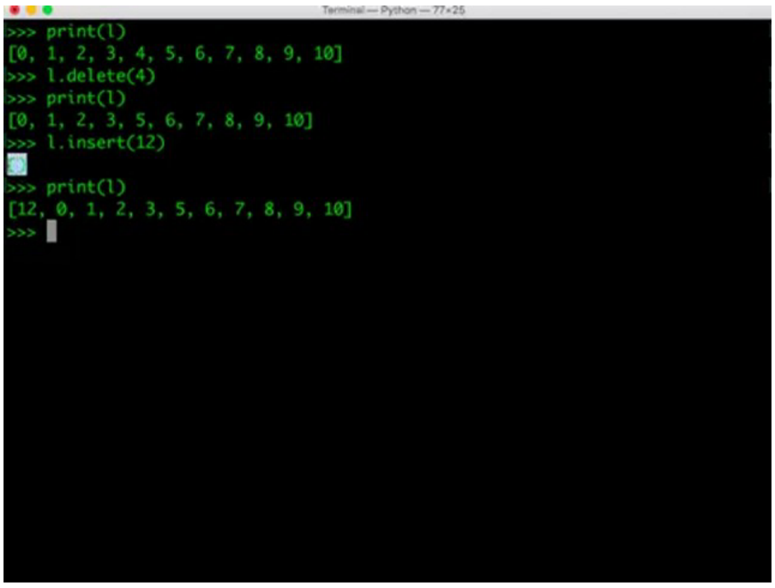
(Refer Slide Time: 22:34)



```
>>> from list import *
>>> l = Node(0)
>>> print(l)
[0]
>>> for i range(1,11):
  File "<stdin>", line 1
    for i range(1,11):
              ^
SyntaxError: invalid syntax
>>> for i in range(1,11):
...    l.append(i)
...
()
()
()
()
()
()
()
()
()
()
>>> l
<list.Node object at 0x101bd9ef0>
>>> 
```

And then if we at this point print l then we get 0 to 10 as before.

(Refer Slide Time: 22:40)



```
>>> print(l)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> l.delete(4)
>>> print(l)
[0, 1, 2, 3, 5, 6, 7, 8, 9, 10]
>>> l.insert(12)
()
>>> print(l)
[12, 0, 1, 2, 3, 5, 6, 7, 8, 9, 10]
>>> 
```

Now we say l dot delete 4 for instance and we print l then 4 is formed and so on. If we say l dot insert 12 and print l, then 12 will begin. So, you can check that this works. Notice that we are getting these empty brackets, this is the returned value. So, when we wrote this return, we wrote with the empty argument. And then we get this empty tuple, we can just write a return with nothing and then it would not display this funnier return value, but what is actually important is that the internal representation of our list is correctly changing with the functions that we have written.