

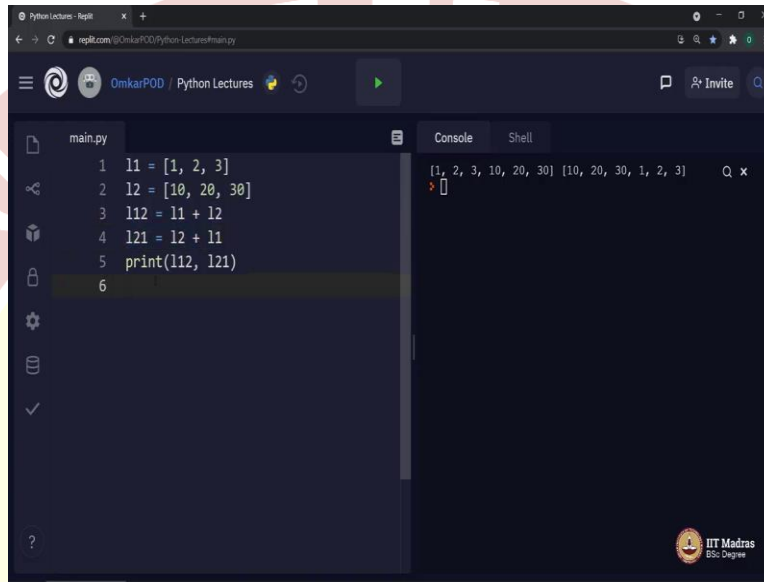


IIT Madras

ONLINE DEGREE

Programming in Python
Professor. Sudarshan Iyengar
Department of Computer Science & Engineering
Indian Institute of Technology, Ropar
More on Lists

(Refer Slide Time: 00:16)

A screenshot of a Python REPL window titled 'Python Lectures - Repl.it'. The code editor on the left shows a file named 'main.py' with the following code:

```
1 l1 = [1, 2, 3]
2 l2 = [10, 20, 30]
3 l12 = l1 + l2
4 l21 = l2 + l1
5 print(l12, l21)
6
```

The console on the right shows the output of the code:

```
[1, 2, 3, 10, 20, 30] [10, 20, 30, 1, 2, 3]
```

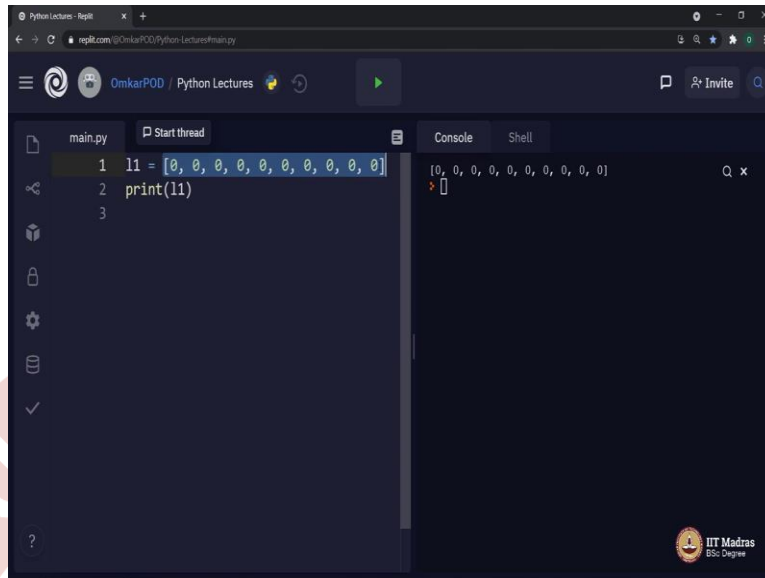
The background of the slide features a large, faint watermark of the Indian Institute of Technology Madras logo.

Hello, Python students. It has been two weeks we have using lists. And I am sure by now you all must have got comfortable with lists. So, this is the good time to introduce some advanced concepts related to lists. So, let us start with operations on lists.

Let us look at this particular code, l1 is a list with three elements 1, 2, 3, l2 is another list with three elements 10, 20, 30, and then we have done some new operation. It looks interesting, because we have used plus operator with two lists. We know we can use plus operator to add two numbers or we can use the plus operator to concatenate two strings. But here, we are using the same plus operator to concatenate two lists. In line number three, we have written l1 plus l2, whereas in line number four, we have written l2 plus l1. The sequence of two operands for this plus operator is different. Let us execute.

In first case, it prints 1, 2, 3, followed by 10, 20, 30, because 1, 2, 3 is l1 appended with l2, which is 10, 20, 30, whereas in second case, we started with l2 and then concatenated with l1, as we have done over here in these two lines.

(Refer Slide Time: 2:00)

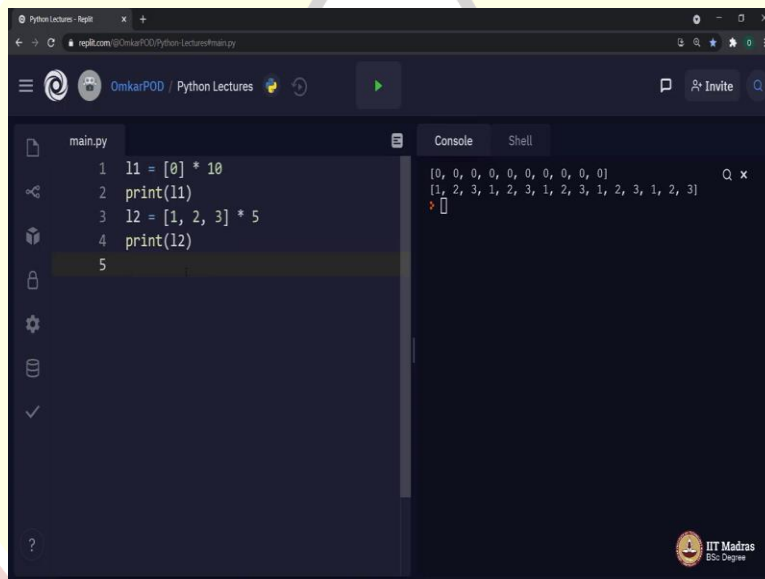


The screenshot shows a web-based Python REPL interface. The code editor on the left contains the following code:

```
1 l1 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
2 print(l1)
3
```

The console on the right displays the output:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```



The screenshot shows the same Python REPL interface with updated code:

```
1 l1 = [0] * 10
2 print(l1)
3 l2 = [1, 2, 3] * 5
4 print(l2)
5
```

The console on the right displays the output for both lines:

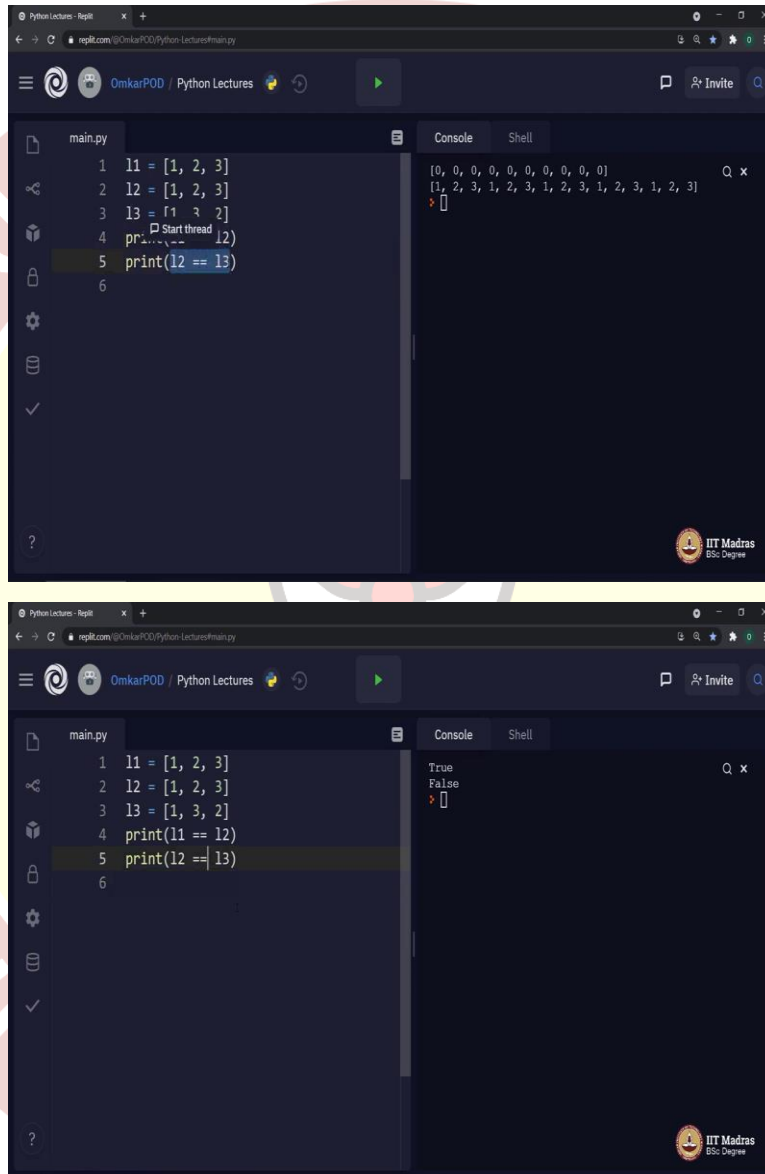
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Let us move to next operator. Look at this particular code block. Here, we have declared one list with 10 zeros in it. This is a very common operation which we come across many times. If you remember, when we use matrices, we always had to create lists with zeros in it. Every time typing these 10 zeros for every row of a matrix is an unnecessary and repetitive task. So, is there any way to avoid this? So, in order to simplify this particular line of code, we can simply use multiplication operator along with this list with only one zero in it. This code will give the same output as it was earlier with 10 zeros in it.

Now, you must be thinking, is it possible to do this only with a list with single 0 or can we do it with a list with some other numbers as well. Let us try, l2 is equal to 1, 2, 3 multiplied by 5, print

12. Let us execute 1, 2, 3 once, twice, thrice, fourth time and fifth time, as we expected. After concatenation and this replication, let us move to next set of operators, which are logical operators.

(Refer Slide Time: 03:37)



The image displays two screenshots of a Python REPL interface, likely from a video lecture. The interface shows a code editor on the left and a console on the right. The code in the editor defines three lists: l1 = [1, 2, 3], l2 = [1, 2, 3], and l3 = [1, 3, 2]. The console shows the output of the code execution, including the lists themselves and the results of equality checks (l1 == l2 and l2 == l3).

Top Screenshot:

```
main.py
1 l1 = [1, 2, 3]
2 l2 = [1, 2, 3]
3 l3 = [1, 3, 2]
4 print(l1 == l2)
5 print(l2 == l3)
6
```

Console:

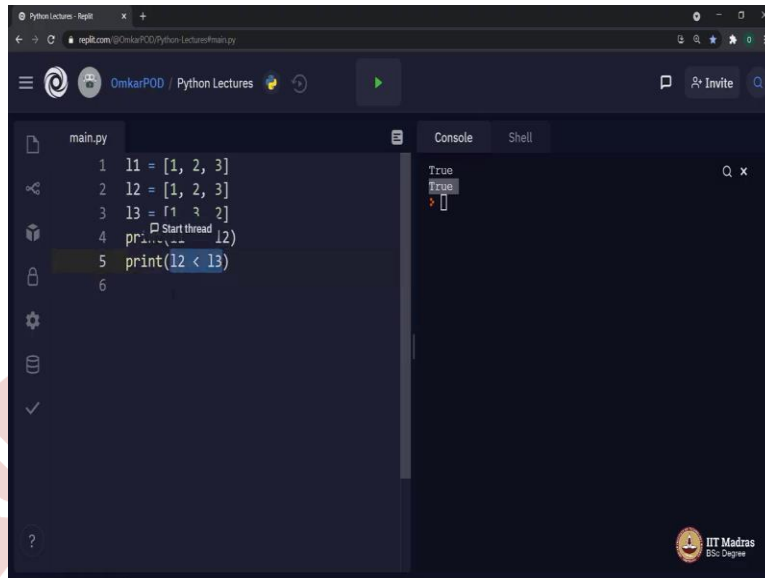
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>
```

Bottom Screenshot:

```
main.py
1 l1 = [1, 2, 3]
2 l2 = [1, 2, 3]
3 l3 = [1, 3, 2]
4 print(l1 == l2)
5 print(l2 == l3)
6
```

Console:

```
True
False
>
```

A screenshot of a web-based Python IDE. The editor shows a file named 'main.py' with the following code:

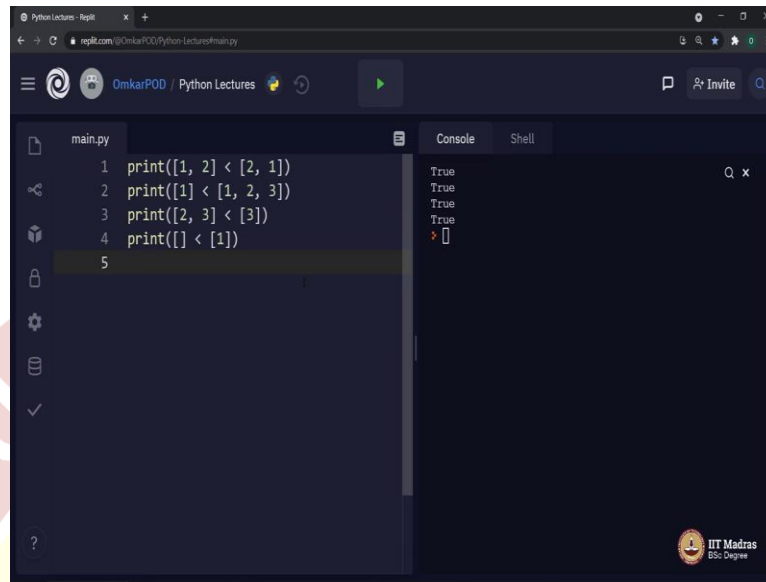
```
1 l1 = [1, 2, 3]
2 l2 = [1, 2, 3]
3 l3 = [1, 3, 2]
4 print(threading.Thread(target=lambda: print(l2 < l3)).start())
5 print(l2 < l3)
```

The console on the right shows the output: 'True'. The IDE interface includes a file explorer on the left, a top bar with 'OmkarPOD / Python Lectures', and a bottom right corner with the IIT Madras logo and 'BSc Degree' text.

Let us look at this particular code, l1 is equal to 1, 2, 3, l2 is equal to again 1, 2, 3, whereas l3 is 1, 3, 2. Now, let us check whether l1 is equal, equal to l2 or not and then l2 is equal, equal to l3 or not. In first case, it says true, and in second case, it says false. Which means the equality operator not only checks the values in the list, it also checks whether the sequence of those values is identical or not. As we have started with comparing two lists using equality operator, let us check is it possible to compare two lists using less than or a greater than symbol. Let us try this.

Now, it says true. Which means the computer says l2 is less than l3. Now, you must be wondering how computer can compare two lists. Let us look at few more examples which will explain how two lists are compared with each other.

(Refer Slide Time: 04:52)



The screenshot shows a web-based Python REPL interface. The left pane contains a file named 'main.py' with the following code:

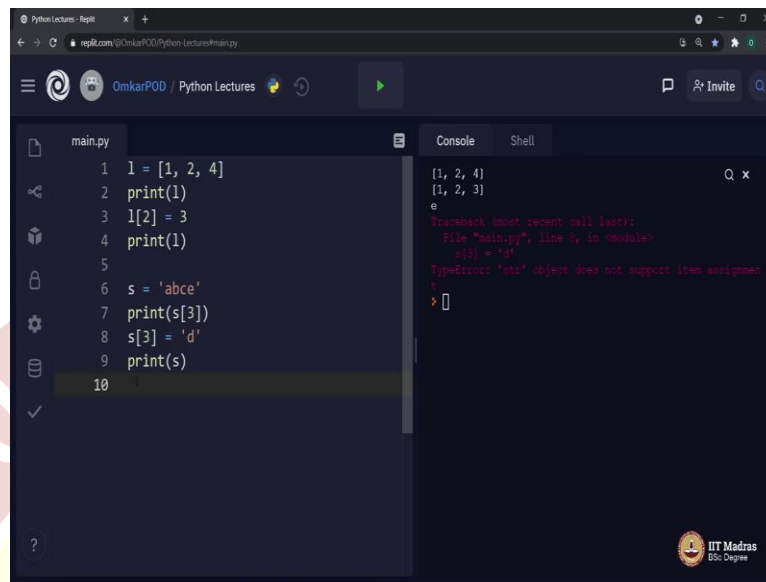
```
1 print([1, 2] < [2, 1])
2 print([1] < [1, 2, 3])
3 print([2, 3] < [3])
4 print([] < [1])
5
```

The right pane shows the 'Console' output, which displays 'True' for each of the four print statements. The interface includes a top bar with 'OmkarPOD / Python Lectures' and a bottom right corner with the 'IIT Madras' logo.

Look at these four print functions. Let us execute and observe the output. It is printing true for every expression over here. This is happening because computer compares two lists using the elements one at a time. The element at 0th index is compared against element at 0th index in second list. Now, as we know 1 is less than 2, it simply says true. In second case, now, 1 is equal to 1, after that there is no element in first list to compare against the second element of second list. Hence, once again it says true.

In third instance, now, 2 is less than 3. Hence, true. And in last instance, empty list has no element to compare against this element of second list. Once again, it says true. Now, after operations the next concept is called mutability. Before explaining this new jargon, let us look at one simple code, which will help us understand this particular concept.

(Refer Slide Time: 06:15)

A screenshot of a Python REPL window titled "Python Lectures - Repl.it". The window shows a file named "main.py" with the following code:

```
1 l = [1, 2, 4]
2 print(l)
3 l[2] = 3
4 print(l)
5
6 s = 'abce'
7 print(s[3])
8 s[3] = 'd'
9 print(s)
10
```

The console output shows:

```
(1, 2, 4)
[1, 2, 3]
```

Below the output, a red error message is displayed:

```
Traceback (most recent call last):
  File "main.py", line 8, in module:
    s[3] = 'd'
TypeError: 'str' object does not support item assignment
```

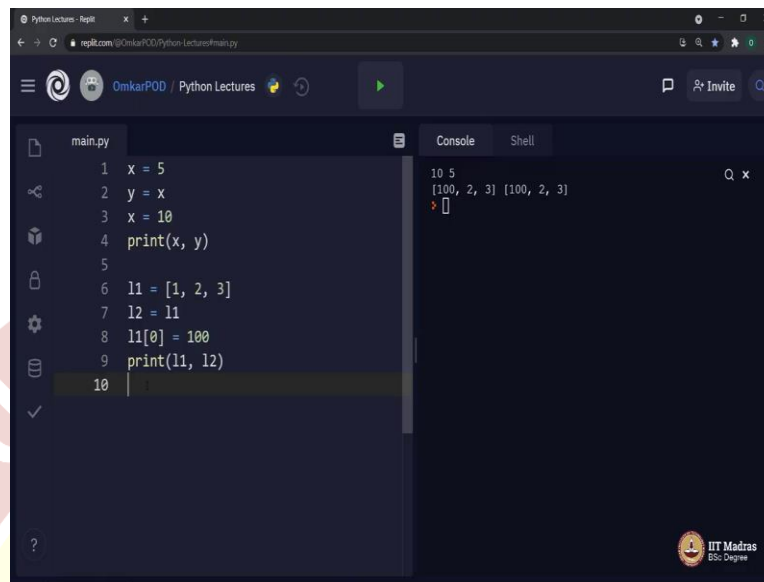
The IIT Madras logo is visible in the bottom right corner of the window.

Let us look at this simple code, `l` is equal to `[1, 2, 4]`, print `l`, `l` of `2` is equal to `3`, print `l`. We all know what will be the output of this code `1, 2, 4, 1, 2, 3`. Now, you must be thinking what is so special about this code? This is very, very basic thing. This line number three is very interesting. We never talked about the logic behind this particular line. Python allows us to update a list in place, in place as in at specific location, which is this index two, we are modifying its value from `4` to `3`. Python allows us to modify the list in such a way and due to that Python list is called as mutable.

Now, you might be thinking what, why there is so much of importance is given to this particular thing. Let us try to execute same operation using a string then we will know why this is so much important with respect to Python programming. Look at this code. Let us execute and then we will see. There is an error. It says, `str` object does not support item assignment. We can access a particular character of the string using the index, but we cannot modify that character. As you can see, `s` of `3` is equal to `d`. This particular line is giving an error.

I am sure many of you must have came across this error in earlier weeks and many of you might still have that doubt why that was happening with strings. And the answer is string is an immutable entity in Python. Immutable as in you cannot change a value of a specific character like this. And this makes this concept of mutability very important. Now, let us execute one more interesting code block, which will force you to think about how lists are defined in Python.

(Refer Slide Time: 08:42)



The screenshot shows a Python REPL window with a dark theme. The code editor on the left contains the following code in `main.py`:

```
1 x = 5
2 y = x
3 x = 10
4 print(x, y)
5
6 l1 = [1, 2, 3]
7 l2 = l1
8 l1[0] = 100
9 print(l1, l2)
10
```

The console on the right shows the output of the code:

```
10 5
[100, 2, 3] [100, 2, 3]
```

The background of the slide features a large, faint watermark of the IIT Madras logo and the text "INSTITUTE OF TECHNOLOGY MADRAS" and "सिद्धिभवति कर्मजा".

Look at this code, as expected, output is 10, 5. Now, you must be thinking why I am teaching you this very basic kg level code. If that is what you are thinking then tell me what will be the output of next code block, l1 is equal to 1, 2, 3, similar to x is equal to 5, l2 is equal to l1, which is again similar to y is equal to x, instead of changing the value of x and changing the 0th index value of l1 to 100 and then printing l1 comma l2. Take a pause and tell me what will be the output of these four lines of code.

In such a case, the l1 should print 100, 2, 3, whereas l2 should print 1, 2, 3. I am sure you all must be thinking that is the output. And once again, what is so special about it. Let us execute and see. No there is some difference. We updated the value of only l1 of 0, even after that l2 of 0 is showing 100. Ideally, it should have been 1 only. But that is not the case over here. It works like this with integers. But when we try to replicate similar thing with list, it is behaving in a different manner. Now, the question is why? And the answer is, it is because of the internal implementation of integers and lists in Python language.

Now, let us see what happens internally when we execute this particular code block. When we say x equal to 5, computer allot a certain memory in the computer system and stores value 5 in that memory and gives a name to that particular memory location as x. When we say y is equal to x, computer creates another memory location, stores the value of x in that new memory location and gives it a name y, which means at this point of time, there are two different memory

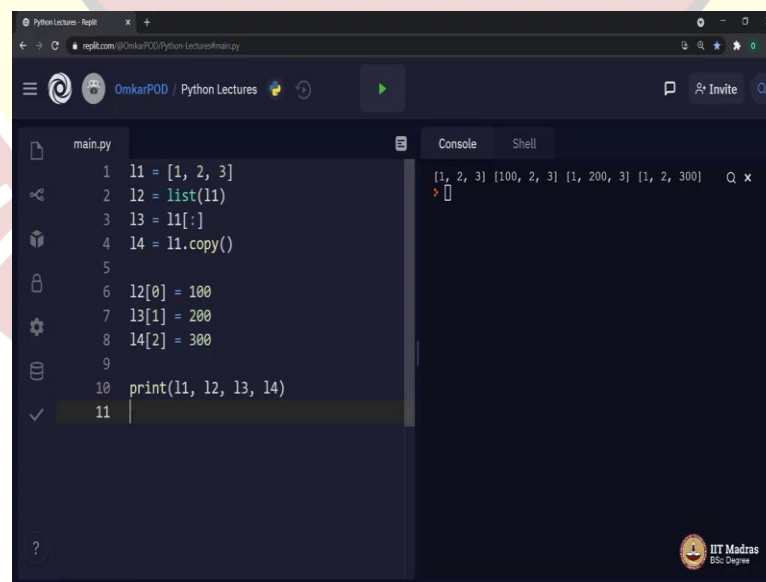
locations in the computer system both has 5 as value in it, but the names given to both these memory locations are different x for first and y for second one.

Whenever we say x equal to 10, computer replace the existing value at memory location x from 5 to 10. But at the second memory location, which is named as y still has value 5. But when we talk about lists, the computer behaves in a different manner. When we say l1 is equal to list of 1 comma 2 comma 3, similar to x equal to 5 computer allots a memory block and in that memory location computer stores this particular list with these three values inside it and gives it a name, l1.

When we say l2 is equal to l1, instead of creating a new memory location, computer simply adds one more name to that same memory location which was created earlier. Which means after executing this particular line, computer still has only one memory location where these values are stored. And because of this, l1 as well as l2 both variables are printing the same value. If you change any value in this list using either of these names, it will reflect for other name as well.

Now, because of this discussion, there arises a new question how to create separate copy of l2. In order to solve this problem, Python provides three different ways using which we can copy a list, which will create a new memory location. Let us look at those three ways.

(Refer Slide Time: 13:20)



The screenshot shows a Python IDE with a file named 'main.py' and a console window. The code in 'main.py' demonstrates three ways to create a copy of a list 'l1'.

```
1 l1 = [1, 2, 3]
2 l2 = list(l1)
3 l3 = l1[:]
4 l4 = l1.copy()
5
6 l2[0] = 100
7 l3[1] = 200
8 l4[2] = 300
9
10 print(l1, l2, l3, l4)
11
```

The console output shows the result of the print statement:

```
[1, 2, 3] [100, 2, 3] [1, 200, 3] [1, 2, 300]
```

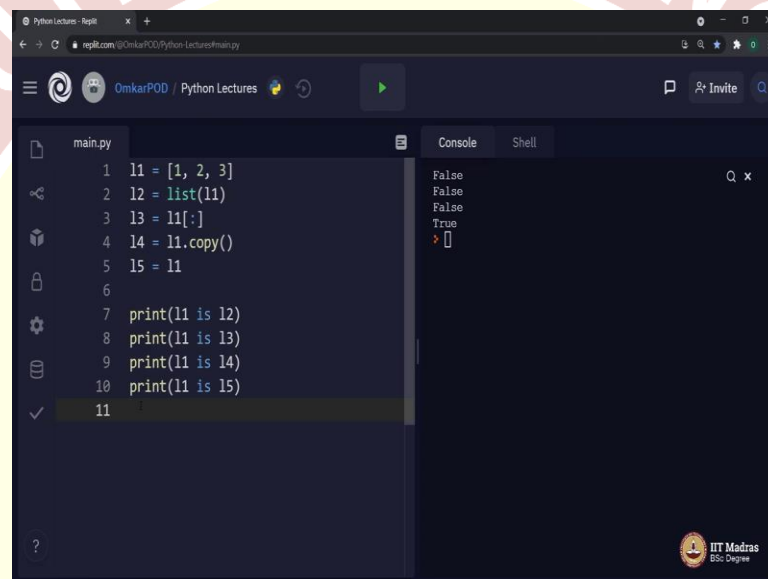
The IDE interface includes a file explorer on the left, a code editor in the center, and a console on the right. The background features a large, faint watermark of the IIT Madras logo.

l1 is equal to 1, 2, 3. The first way is l2 is equal to list of l1. Second option, l3 is equal to l1 square bracket inside colon. This is exactly like slicing which we saw with strings. And the third

option is using a list method called copy. Let us execute this code and observe the output. And as you can observe, these three changes are specific to lists l2, l3 and l4, respectively. It has no effect on the original list or any other lists.

Now, one may ask a question, using a Python program how to figure out whether two different variables are pointing at a same memory location or they are pointing at two different memory locations. Python has an answer to this question as well and the answer is, is operator.

(Refer Slide Time: 14:34)



The screenshot shows a Python IDE window titled 'Python Lectures - Replit'. The code in 'main.py' is as follows:

```
1 l1 = [1, 2, 3]
2 l2 = list(l1)
3 l3 = l1[:]
4 l4 = l1.copy()
5 l5 = l1
6
7 print(l1 is l2)
8 print(l1 is l3)
9 print(l1 is l4)
10 print(l1 is l5)
11
```

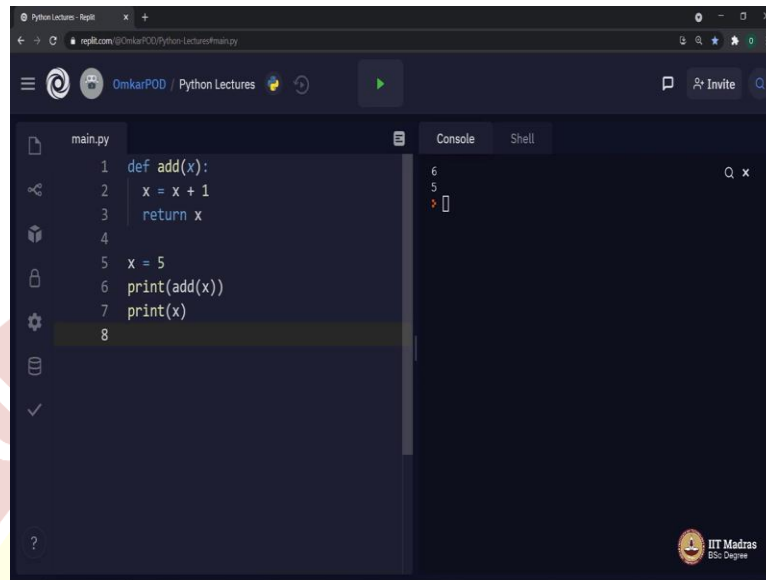
The console output on the right shows the results of the 'is' operator:

```
False
False
False
True
```

The IIT Madras logo is visible in the bottom right corner of the IDE window.

Let us look at this particular code. Let us see how computer executes this particular keyword is. It says false for first three print statements followed by true. If the is operator says true, which means l1 is l5 only. That means there are simply two names given to same memory location, whereas for first three print statement, it says false, l1 is not l2, l1 is not l3, and l1 is not l4, which means these are variable names which are associated with different memory locations. In last week, we have introduced functions. So, the next concept is how lists behave when we pass list variable as an argument to a function.

(Refer Slide Time: 15:31)



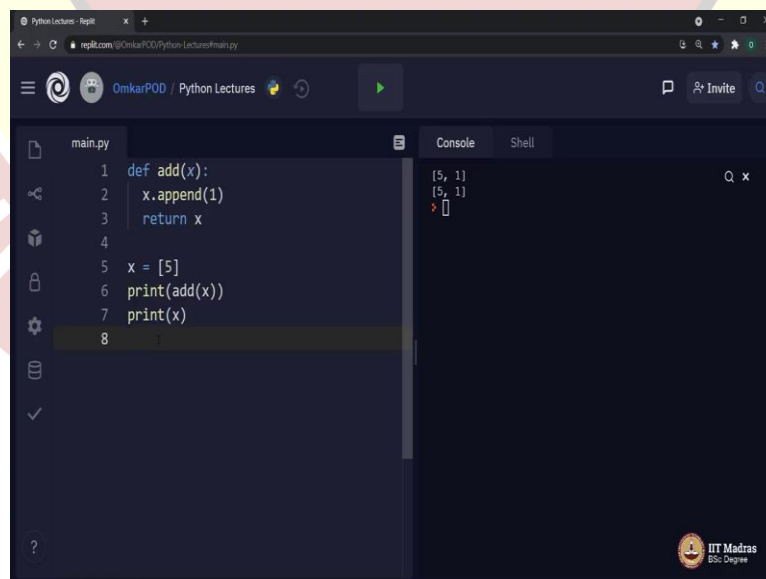
```
main.py
1 def add(x):
2     x = x + 1
3     return x
4
5 x = 5
6 print(add(x))
7 print(x)
8
```

Console

```
6
5
```

Let us execute this code. It says 6 followed by 5. And we have seen this earlier in previous week. This happens because there are two different versions, two different copies of variable x. One is in a global space, whereas second is in a local space. The local space x has value 6, whereas this global x has value 5. Now, let us modify this code and convert this integer into a list and then we will see whether is there any difference or it behaves in a same manner.

(Refer Slide Time: 16:16)



```
main.py
1 def add(x):
2     x.append(1)
3     return x
4
5 x = [5]
6 print(add(x))
7 print(x)
8
```

Console

```
[5, 1]
[5, 1]
```

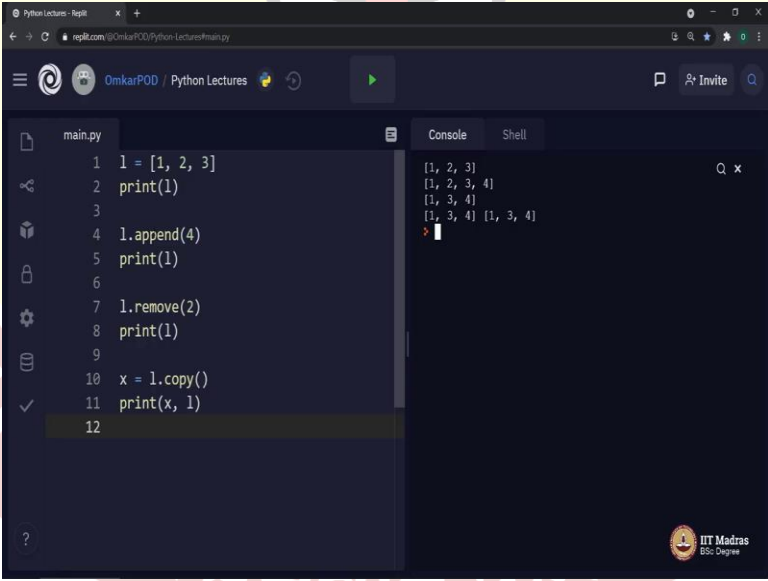
Function add x dot append 1, instead of doing x plus 1 we will do x dot append 1, because it is a list, and x is a list with one element 5 in it, rest of the code is still same. Let us execute and then

we will see. Now, both print functions are giving the same output which was not the case earlier with integers, which means lists behave in a different manner as compared to integers. Now, let us see what is happening over here.

When we were executing a function with an integer argument, we were passing the value of that integer variable. But when we talk about list, we are not passing the value of x, we are passing the x itself. Because of that, when we say x dot append, it appends the value 1 to the original variable x. And because of that, it says 5 comma 1 in both these instances and this kind of argument passing is referred as call by reference.

And because of that, there is a single copy, a single version of this variable x which is associated with that memory location. So, listen to this carefully. If the function argument is of mutable type then it is called by reference. Otherwise, it is called by value. That is why in case of lists, it is called by reference, whereas in case of integer, it was called by value. Let us move to the last point, list methods.

(Refer Slide Time: 18:18)



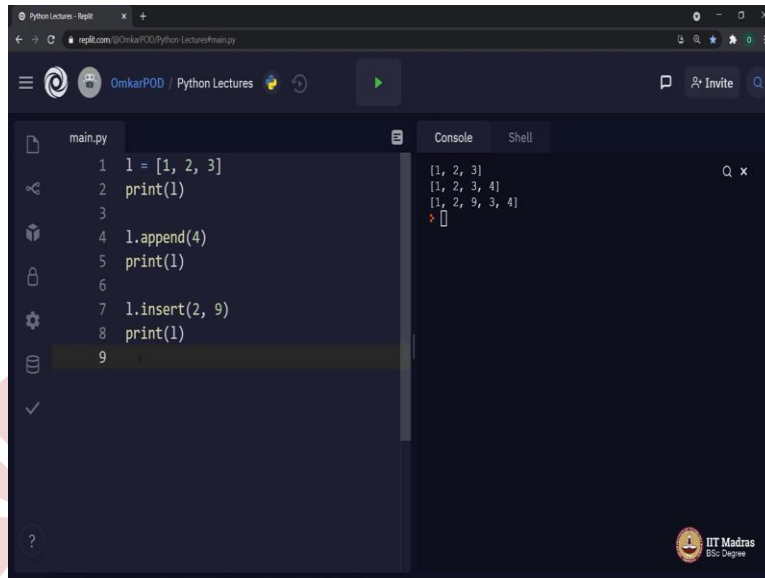
The screenshot shows a Python IDE with a file named `main.py` and a console window. The code in `main.py` is as follows:

```
1 l = [1, 2, 3]
2 print(l)
3
4 l.append(4)
5 print(l)
6
7 l.remove(2)
8 print(l)
9
10 x = l.copy()
11 print(x, l)
12
```

The console output shows the following sequence of list states:

```
[1, 2, 3]
[1, 2, 3, 4]
[1, 3, 4]
[1, 3, 4] [1, 3, 4]
```

The IDE interface includes a file explorer on the left, a code editor in the center, and a console on the right. The background features a large, faint watermark of the Indian Institute of Technology Madras logo.

A screenshot of a Python REPL interface. The left pane shows a file named 'main.py' with the following code:

```
1 l = [1, 2, 3]
2 print(l)
3
4 l.append(4)
5 print(l)
6
7 l.insert(2, 9)
8 print(l)
9
```

The right pane is split into 'Console' and 'Shell' tabs. The 'Console' tab shows the output of the code execution:

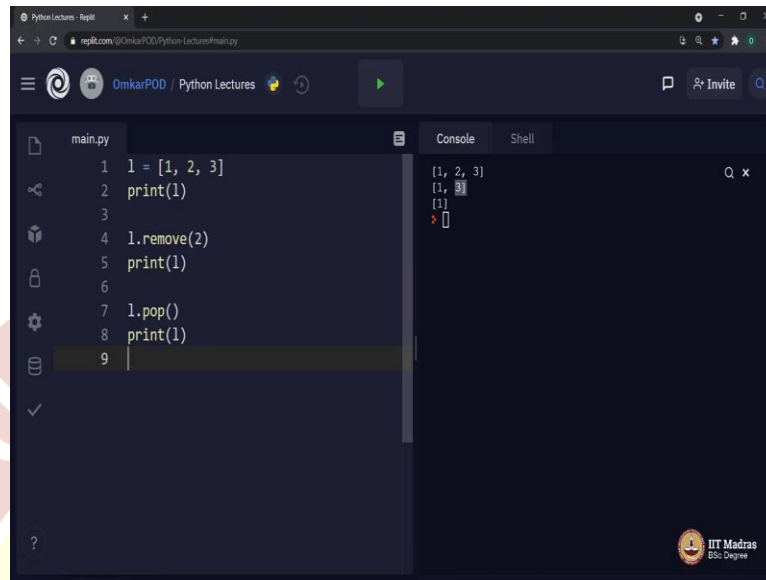
```
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 9, 3, 4]
```

The 'Shell' tab is empty. The interface includes a search bar, a 'Run' button, and a 'Python Lectures' header. A large, faint watermark of the IIT Madras logo is visible in the background.

Let us look at this code block. These append and remove are two methods of lists. In addition to these two methods, we saw one more method few minutes back in the same lecture, and the method is copy. Now, let us look at another list method which is very similar to this append. L dot insert 2 comma 9, append was accepting only one argument, whereas insert is taking two arguments.

Let us execute and then we will see what is the difference between append and insert. Append adds the mentioned element at the end of the list, whereas when you say insert, it will insert the value 9 at second index of that particular list. Now, let us look at another method which is similar to remove.

(Refer Slide Time: 19:25)



```
main.py
1 l = [1, 2, 3]
2 print(l)
3
4 l.remove(2)
5 print(l)
6
7 l.pop()
8 print(l)
9
```

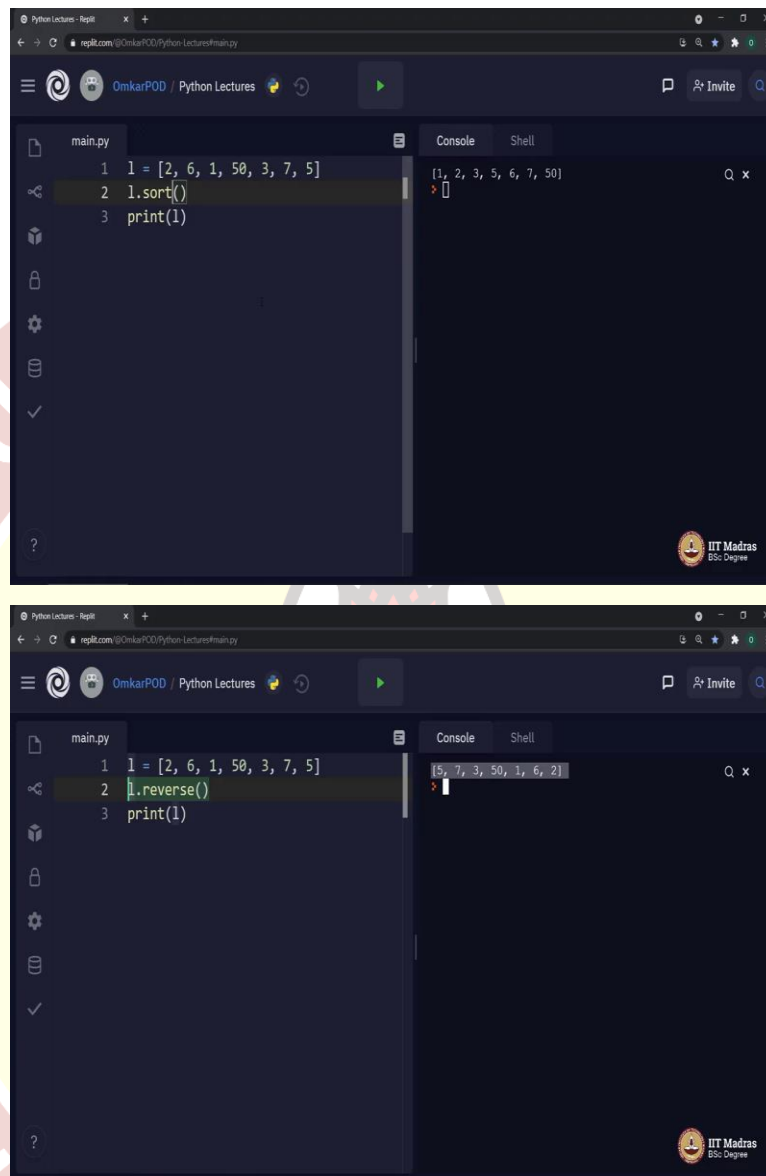
Console

```
{1, 2, 3}
{1, 3}
{3}
```

Now, you must be thinking the element 0 is not there in the list. Be patient and look at this particular output. Original list 1, 2, 3, after remove, we got 1 comma 3 as expected, because this element 2 has been removed from the list. And then when we say 1 dot pop, the output is a list with only one element which is 3. So, what is the difference between remove and pop? When we say remove, computer deletes this particular element from the list. Whereas, whenever we say pop, computer deletes the element which is present at 0th index. That is why the element 1 has been deleted.

Before closing this lecture, let us look at two more list methods, which will come handy while writing Python programs. I am sure you will appreciate both these methods very much.

(Refer Slide Time: 20:34)



The image displays two screenshots of a Python REPL interface, likely from a video lecture. The interface shows a code editor on the left and a console on the right. The code editor contains a file named 'main.py' with the following code:

```
1 l = [2, 6, 1, 50, 3, 7, 5]
2 l.sort()
3 print(l)
```

The console shows the output of the code execution:

```
[1, 2, 3, 5, 6, 7, 50]
```

The second screenshot shows the same code editor with the code:

```
1 l = [2, 6, 1, 50, 3, 7, 5]
2 l.reverse()
3 print(l)
```

The console shows the output of the code execution:

```
[5, 7, 3, 50, 1, 6, 2]
```

Let us look at this particular list. And the question is, write a Python program to sort this given list, `l.sort()` print `l`, let us execute. We got the list in a sorted order in a single line of code. But if you can observe, this particular sort method, it is sorting these numbers in an ascending order. Now, it is your job to find out how to sort the numbers in descending order. Now, the last method of lists, which is `reverse()`, let us execute and the output is the same list in a reverse order. Thank you for watching this lecture. Happy learning.