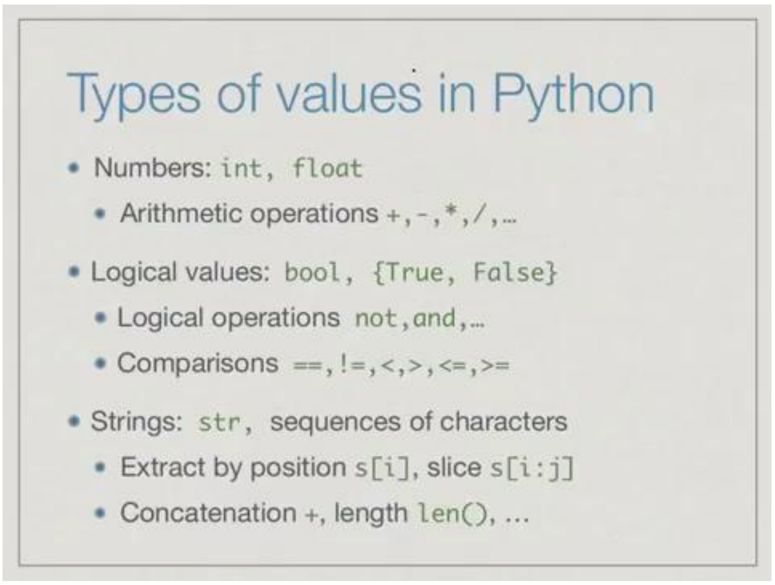
Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 02
Lecture - 03
Lists

So far we have seen some basic Types of values in Python.

(Refer Slide Time: 00:02)



Types of values in Python

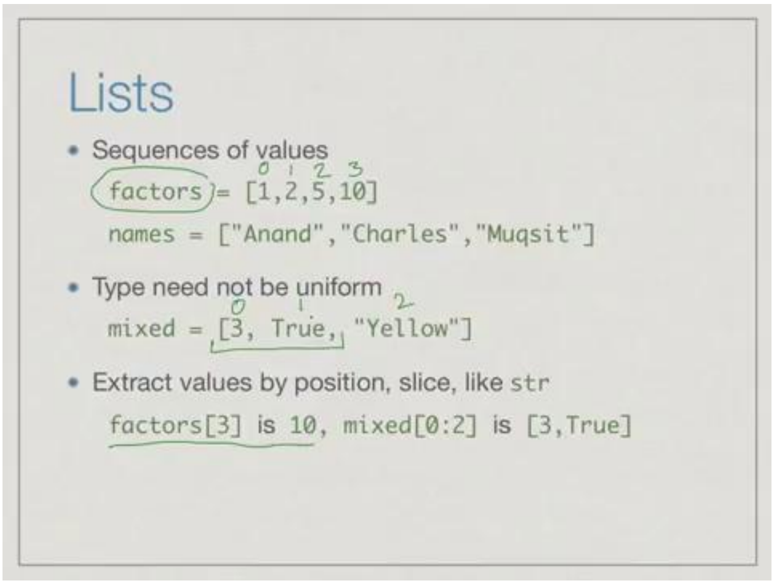
- Numbers: int, float
  - Arithmetic operations +, -, *, /, ...
- Logical values: bool, {True, False}
  - Logical operations not, and, ...
  - Comparisons ==, !=, <, >, <=, >=
- Strings: str, sequences of characters
  - Extract by position s[i], slice s[i:j]
  - Concatenation +, length len(), ...

You began with the numeric types, which divided into two categories int and float. So, int represented whole numbers or integers, and float represented values which have a decimal point. And for these, we had arithmetic operations such as plus, minus, times, divide and also other functions which we can import using the math library, which is built into python. Then we introduce a new type of value, which may not be so familiar for logical values true and false which are of type bool.

We can operate on these values using functions such as not, which negates the value makes at the opposite 'and' and 'or'. And when we do comparisons between numeric values for instances the outcome of such a comparison is typically a bool value and we can combine these comparisons using 'not' and 'and' to make complex conditions.

In the previous lecture, we look at strings. So, strings are used to represent text a string is of type str. It is a sequence of characters. And since it is a sequence we can talk about positions in the sequence. The position start numbering at 0 and go up to n minus one where n is the length of the string. If we say s square bracket i for a string value s then we get the ith position using this numbering convention. And a slice gives us a sub sequence a string from position i to position j minus one written s square bracket i colon j. The basic operation we can do with strings is to glue them together using the plus operation. Plus means concatenation for strings and not addition in the arithmetic sense, we can extract the length of a string using the len function and we said that we will look at more complex string functions later on.

(Refer Slide Time: 01:59)



Lists

- Sequences of values

```
factors = [1,2,5,10]
names = ["Anand","Charles","Muqsit"]
```

- Type need not be uniform

```
mixed = [3, True, "Yellow"]
```

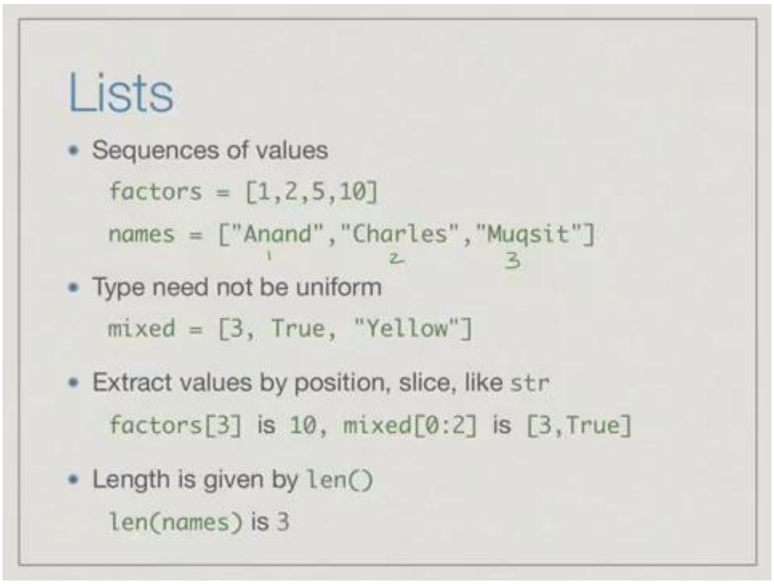
- Extract values by position, slice, like str

```
factors[3] is 10, mixed[0:2] is [3,True]
```

Today we move on to lists. A list is also a sequence of values, but a list need not have a uniform type. So, we could have a list called factors, which has numbers 1, 2, 5, 10. We could have a list called names, which are Anand, Charles and Muqsit. But we could also have a list, which we have called mixed which contains a number or Boolean and a string now it is not usual to have list which have different types of values at different positions, but python certainly allows it. While we will normally have list which are all integers or all strings or all Boolean values it could be the case that different parts of a list have different types.

A list is a sequence in the same way as a string is and it has positions 0 to n minus 1 where n is the length of the list. So, we can now extract values at a given position or we can extract slices. In this example if we take the list factors and we look at the third position remember that the positions are labelled 0, 1, 2, 3 then factors of 3 is 10. Similarly, if you take the list mixed and we take the slice from 0 to 2 minus 1 then we get the sub list consisting of 3 and 2.

(Refer Slide Time: 03:23)



Lists

- Sequences of values

```
factors = [1,2,5,10]
names = ["Anand","Charles","Muqsit"]
```

- Type need not be uniform

```
mixed = [3, True, "Yellow"]
```

- Extract values by position, slice, like `str`

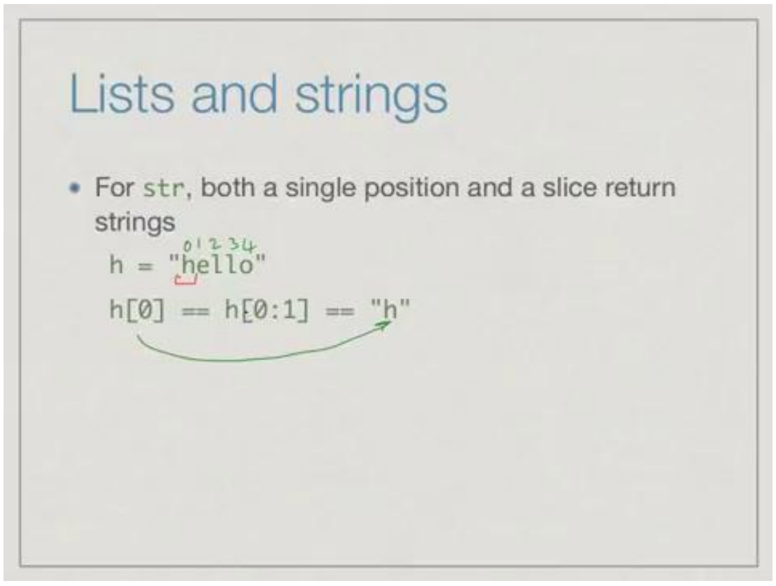
  `factors[3]` is `10`, `mixed[0:2]` is `[3,True]`

- Length is given by `len()`

  `len(names)` is 3

As with a string, the length of the list is given by the function len. So, len of names is 3 because there are 1, 2, 3 values in names. Remember that length is just a normal length, whereas the positions are numbered from 0 to n minus 1.

(Refer Slide Time: 03:43)



Lists and strings

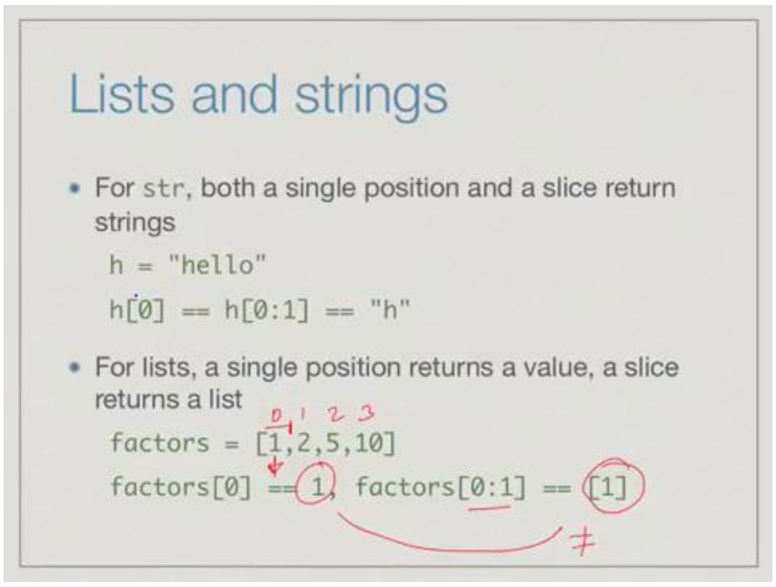
- For `str`, both a single position and a slice return strings

```
h = "hello"
h[0] == h[0:1] == "h"
```

There is one difference between list and strings and what we have seen so far. We said that there was no concept of a single character. In a string if we take the value at single position or we take a string a sub string of length 1, we get the same thing. So, if we have the string h, which has position 1, 2, 3, 4, 5 sorry 1, 2, 3, 4 said as length 5.

And if we ask for the 0th position then this gives us the letter h. But the letter h in python is indistinguishable from the string h similarly if we ask for the sub sequence from 0 to 1, but not including 1 then again we get the string h. So, in one case it's as though we constructed a sub string of length one in one case we got a single character, but python does not distinguish. So, h of 0 is actually equal 2 as a value the sub the slice h 0 colon 1.

(Refer Slide Time: 04:39)



Lists and strings

- For str, both a single position and a slice return strings

```
h = "hello"
h[0] == h[0:1] == "h"
```

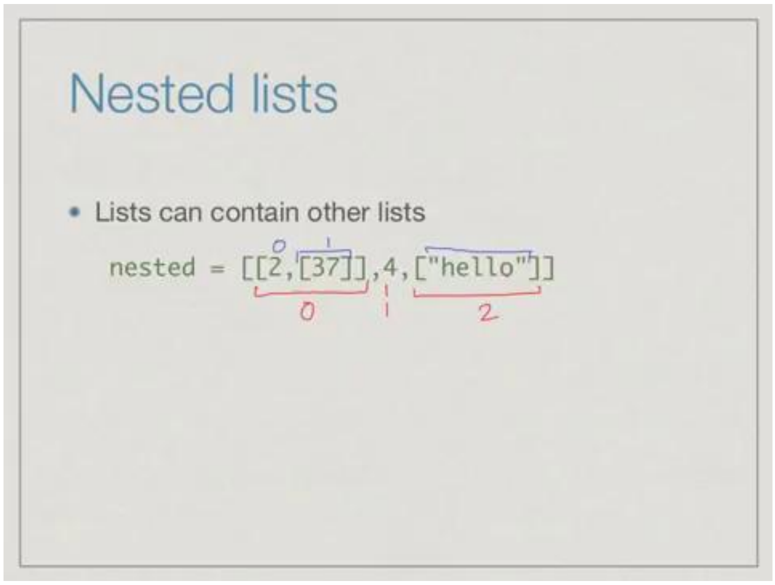
- For lists, a single position returns a value, a slice returns a list

```
factors = [1,2,5,10]
factors[0] == 1, factors[0:1] == [1]
```

Now, this will not happen with the list in general. So, if we have a list right a list consist again positions 0, 1, 2, 3 say. And now we take the 0th position we get a value, we get the value 1 we do not get a list1. On the other hand if we take the slice from 0 up to and not including 1 then we get the sub list of factors of length 1 containing the value 1. So, factors of 0 is 1, factors of 0 colon 1 the slice is also 1, but here we have a single value here we have a list and therefore, these two things are not equal to each other right.

Just remember this that in a string we cannot distinguish between a single value at a position and a slice of length one. They give us exactly the same type of value and the same value itself. Whereas, in a list a slice of length one is a list whereas, a value at a position is a single value at that position.

(Refer Slide Time: 05:39)



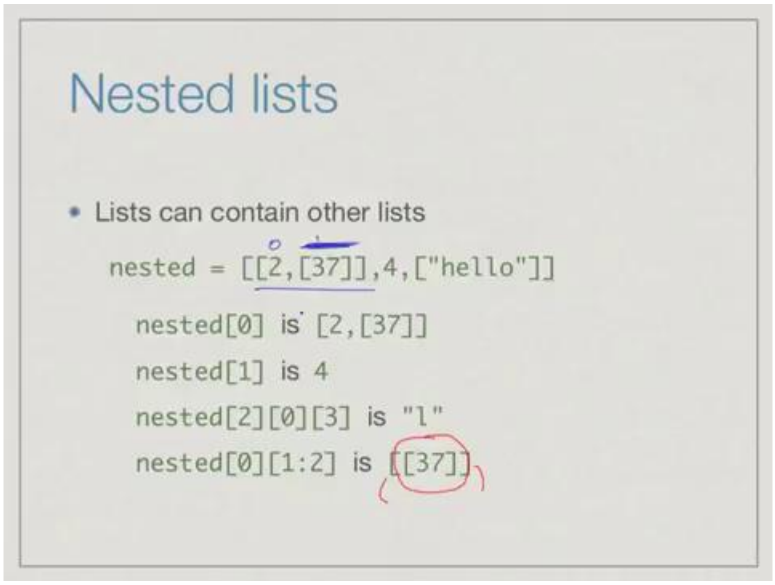
Nested lists

- Lists can contain other lists

```
nested = [[2,[37]],4,["hello"]]
```

Now, nested list can contain other list, so this is called nesting. For example, we can have a nested list. This contains a single value at the beginning which is another list. This is position 1. This is position, sorry position 0. This is position 1 and this is position 2. Position 1 is a single simple value an integer 0 an integer 4 position 0 is a list, which in turn as itself two position 0 and 1. And the value position 1 is itself another list. So, it is a third level of nested list which as a single value 37. Similarly, the value at position 2 is itself a string and therefore, this has seq is this is a sequence and it as its own position.

(Refer Slide Time: 06:33)



Nested lists

- Lists can contain other lists

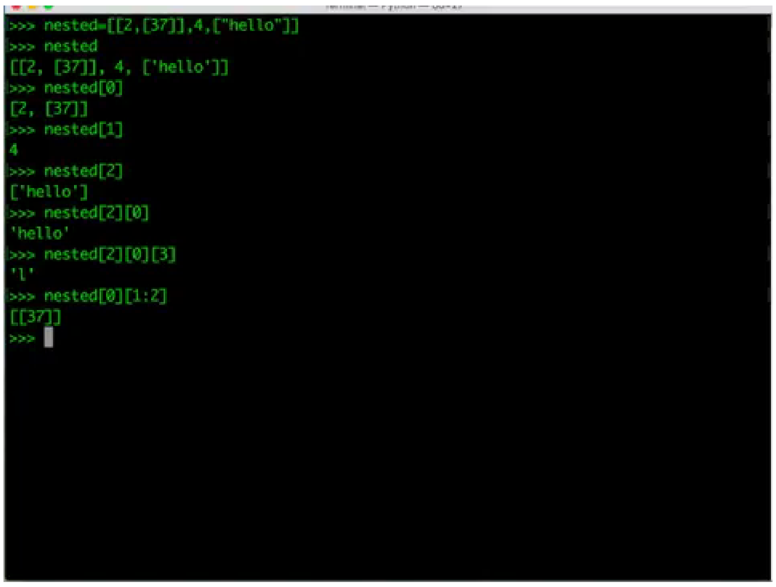
```
nested = [[2,[37]],4,["hello"]]

  nested[0] is [2,[37]]
  nested[1] is 4
  nested[2][0][3] is "l"
  nested[0][1:2] is [[37]]
```

If we look at this example then we can see that if we want to look at the 0th position in nested then as we said we get this value and this value consist of a list itself containing 2 and the list containing 37. On the other hand, if we ask for the first position number 1 then we get the value 4. And now if we look at the position 2, which is this list then, in that we look at the 0th position which is this string and in that we look for the third character which is 0, 1, 2, 3 this l right.

Nested of 2 takes us to the last value in the list nested in that we look at position 0, which is the first value in the nested list. And in that we look at position 3 which is the third character in the sequence contained in that position and we get the character l or the string l actually.

In the same way we can also take slices. So, we can take the 0th position which is this list then we ask for the slice starting at 1 and going up to, but not including 2 so that means, we start with this value. And so we get the list containing the list 37. Notice that the inner list is the value, right. This is the value that lies between position 1 and up to position 2 and the outer list is because, when we take a slice of a list we get a list. This is sub list of this list 2 comma list 37 which gives us just the list 37 we have dropped the value 2, but we get a sub list. This is what we mentioned before for list a slice gives us back a list.

(Refer Slide Time: 08:18)

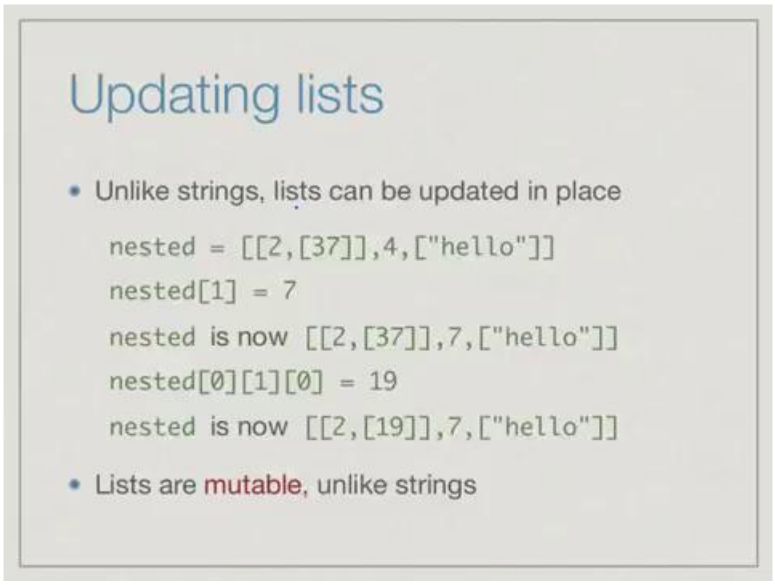


```
>>> nested=[[2,[37]],4,["hello"]]
>>> nested
[[2, [37]], 4, ['hello']]
>>> nested[0]
[2, [37]]
>>> nested[1]
4
>>> nested[2]
['hello']
>>> nested[2][0]
'hello'
>>> nested[2][0][3]
'l'
>>> nested[0][1:2]
[[37]]
>>> 
```

Let us just conform that these things behave as we said. Here we have just loaded the python interpreter with that example. Nested is this list and if you say now nested 0 you get 2, 37 if say nested 1 we get a value 4. Now if we say nested 2 we get this list. We say nested 2, 0 then it drops the list and just gives us a string and if we say nested 2, 0, 3 then we get the string l as we said before.

And then, we said that we can now update for instances nested, none update sorry we can look at nested 0 and take this slice 1 colon 2 and this goes to the first list and gives us the list containing the list containing 37. So, the outer list is because it is a slice and inner one is because the value in position one of the first item in the list nested is itself a list containing 37.

(Refer Slide Time: 09:25)



Updating lists

- Unlike strings, lists can be updated in place

  ```
  nested = [[2,[37]],4,["hello"]]
  nested[1] = 7
  nested is now [[2,[37]],7,["hello"]]
  nested[0][1][0] = 19
  nested is now [[2,[19]],7,["hello"]]
  ```

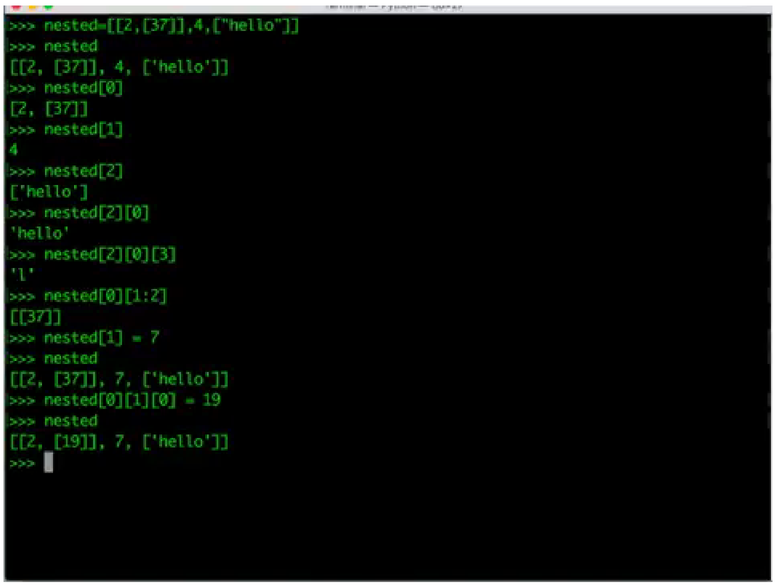
- Lists are mutable, unlike strings

One fundamental difference between list and string or indeed any of the values we have seen before is that a list can be updated in place. So, if we have a list nested as we had before and we say nested of 1 is 7. Remember when we try to change a position in a string we got an error. We cannot change the second l in hello to p just by saying that we want position three to be replaced by p, but for a list this is allowed. If we want to 4 to be replaced by 7, we can just say nested one equal to seven and this will give us the list 2, 37, 7 and then hello and we can do this inside as well.

We can say that we want to go into this list which is nested 0 then we want to go into this list which is nested 0, 1 then we want to go into this value and change this value. We want to change the value at the position 0 of the nested list at position 1 of this initial value. We say nested 0, 1, 0 equal to 19 and this changes that thirty seven into nineteen, so this is allowed. What we say in python notation is that lists are mutable, so mutation is to change.

A list can be transformed in place we can take a list and change its structure unlike a string if we try to change a string we have to actually construct a new string and re assign the name, but in a list with the same name we can update parts of it without affecting the other parts.

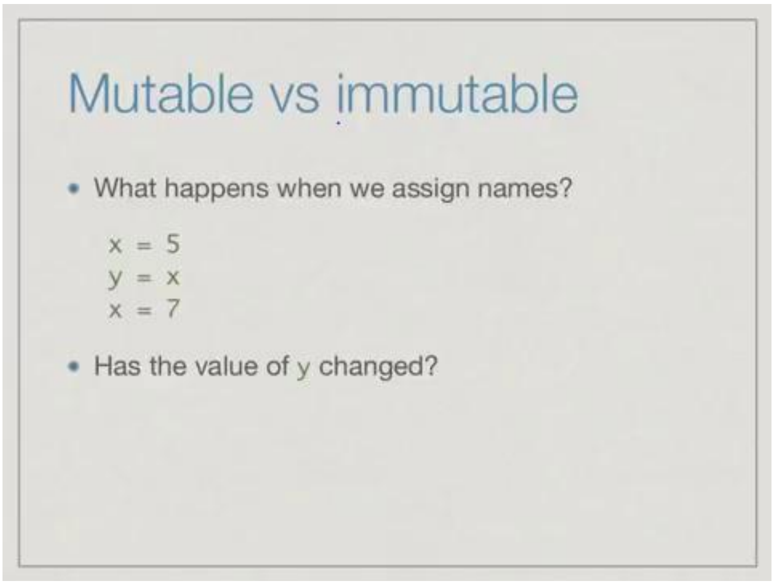
(Refer Slide Time: 11:03)



```
>>> nested=[[2,[37]],4,["hello"]]
>>> nested
[[2, [37]], 4, ['hello']]
>>> nested[0]
[2, [37]]
>>> nested[1]
4
>>> nested[2]
['hello']
>>> nested[2][0]
'hello'
>>> nested[2][0][3]
'l'
>>> nested[0][1:2]
[[37]]
>>> nested[1] = 7
>>> nested
[[2, [37]], 7, ['hello']]
>>> nested[0][1][0] = 19
>>> nested
[[2, [19]], 7, ['hello']]
>>>
```

Once again let us check that what we have done actually works. So, if I say nested of 1 is equal to say 7. Then the list nested the same name now as a 7 in place of the value 4 if I say nested of 0, which is the first list at 1, which is the second nested list at 0 is equal to 19, right. So, this says go and turn to 37 into a 19 and indeed this does happen right. So, this is a difference between list and strings. Lists are mutable values we can go and change values at given position without affecting the name in the rest of the list.

(Refer Slide Time: 11:45)



Mutable vs immutable

- What happens when we assign names?

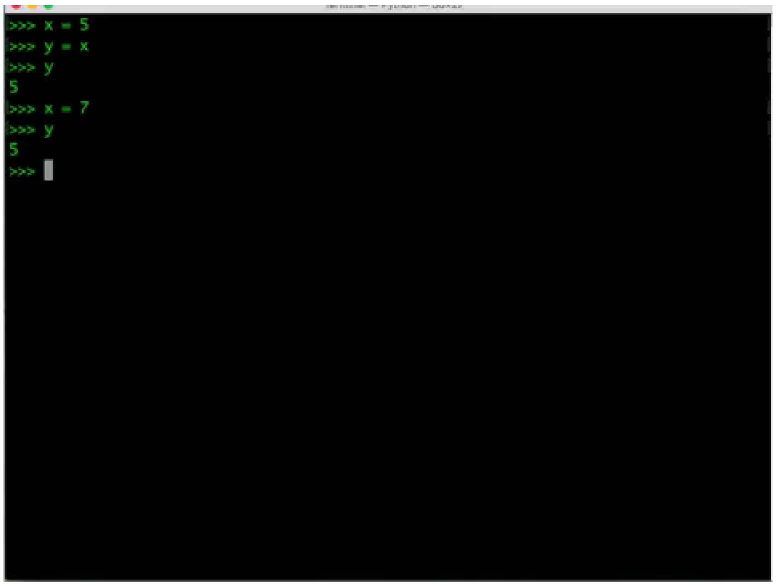
```
x = 5
y = x
x = 7
```

- Has the value of y changed?

It is important to understand the distinction between mutable and immutable values, because this plays an important role in assignment. And as you will later see, it also plays a major role in what happens when we pass values to functions as arguments. Let us look at what happens when we assign names.

Suppose we go through the following sequence of assignments. We initially assign the value 5 to the name x then we assign the name the value and the name x to the value y and then we reassign x to seven. We started with x being 5 then we said y is also 5, because y is a value of x. And now we changed x to 7. So, the question we would like to ask is has the value of y changed. Let us do this and see what happens to that right.

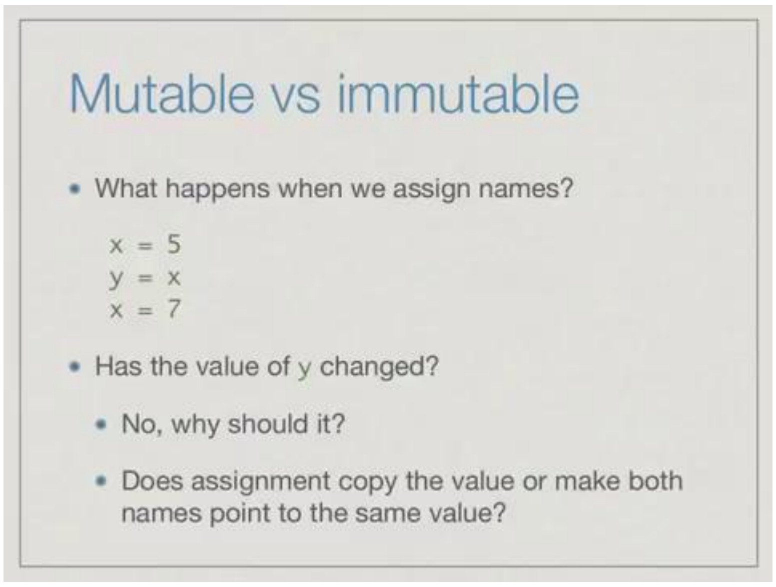
(Refer Slide Time: 12:38)



```
>>> x = 5
>>> y = x
>>> y
5
>>> x = 7
>>> y
5
>>> 
```

Let us start with x equal to 5, y equal to x. So, if we ask for the value of y at this point it is 5 as we expect. Now we change x to seven the question is it is y 5 or y 7 and indeed y is still 5 and this is perfectly natural as far as our understanding goes that what we did, when we set the value of y to the value of x. So, let we make it 5; we did not say make it the same value was x forever hence forth.

(Refer Slide Time: 13:07)



Mutable vs immutable

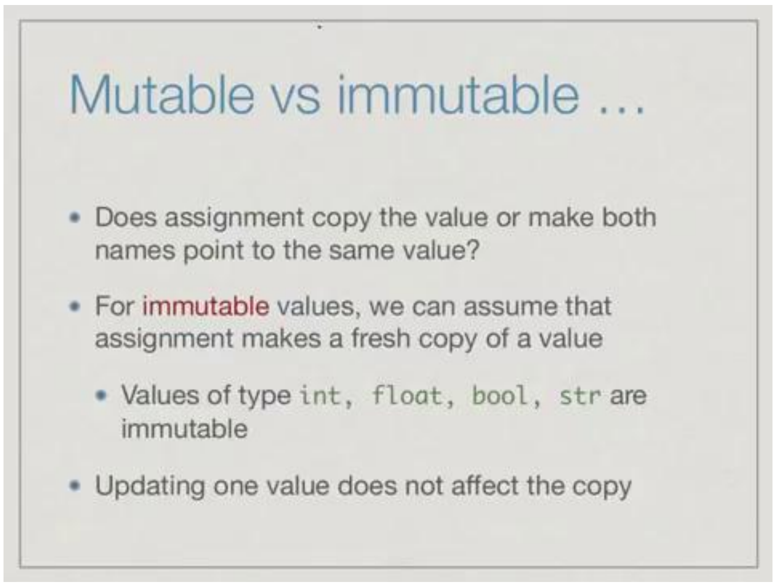
- What happens when we assign names?

```
x = 5
y = x
x = 7
```

- Has the value of y changed?
  - No, why should it?
  - Does assignment copy the value or make both names point to the same value?

As saw the value of y actually did not change and the question is why it should change. After all it seems natural that when we assign a value to the value of another name then what we are actually doing is saying copy that value and make a fresh copy of it. So, x is 5 will make why the same value as x currently is it does not mean that make y and x point to the same value it means make y also 5. So, if x gets updated to 7 it as no effect on y.

(Refer Slide Time: 13:42)



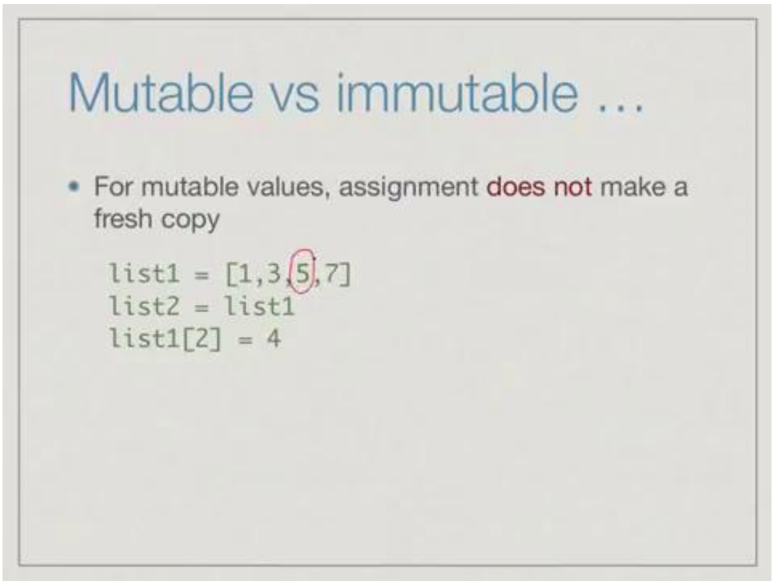
Mutable vs immutable …

- Does assignment copy the value or make both names point to the same value?
- For immutable values, we can assume that assignment makes a fresh copy of a value
  - Values of type `int`, `float`, `bool`, `str` are immutable
- Updating one value does not affect the copy

This question actually is not so simple while our intuition says that assignment should always copy the value. In some cases it does happens that both names end up pointing to the same value. So, for immutable values we can assume what we are intuition says that whenever we assign a name a value we get a fresh copy of that value.

This applies to all the types we have seen before today's lecture namely int float bool and string these are all immutable. If we do the kind of assignment we did before where we assign something to x then make y is the same value was x and then update x, y will not change. Updating one value does not affect the copy, because we have actually copied the value.

(Refer Slide Time: 14:30)



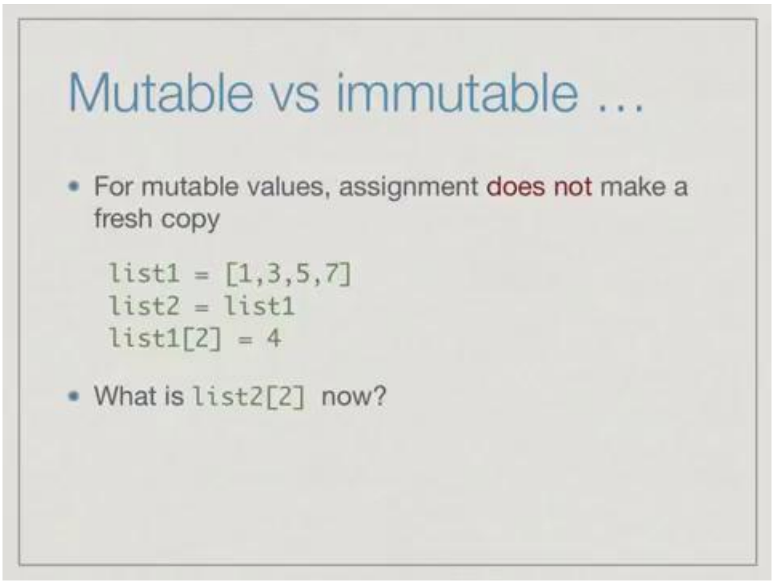
Mutable vs immutable …

- For mutable values, assignment does not make a fresh copy

```
list1 = [1,3,5,7]
list2 = list1
list1[2] = 4
```

However as we are pointed out lists are difference beast from strings and list are mutable. It turns out that for mutable values assignment does not make a fresh copy. Let us look at the following example we first assign say the list 1, 3, 5,7 to the name list1, then we say that list2 is the same as list1. If we had this copy notation now you would have two copies of the list suppose we now use the mutability of list1 to change the value at position 2 namely this value to 4 right.

(Refer Slide Time: 15:08)



Mutable vs immutable …

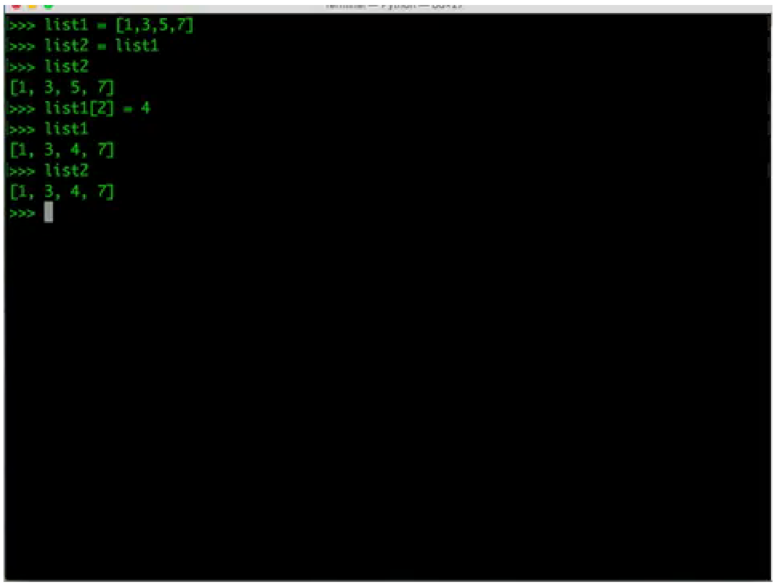
- For mutable values, assignment does not make a fresh copy

```
list1 = [1,3,5,7]
list2 = list1
list1[2] = 4
```

- What is `list2[2]` now?

The question is what has happened to list2, is list2 the same as before namely 1, 3, 5, 7 or as list2 also become 1, 3, 4, 7 like list1. So, lets us see what happens in the interpreter.

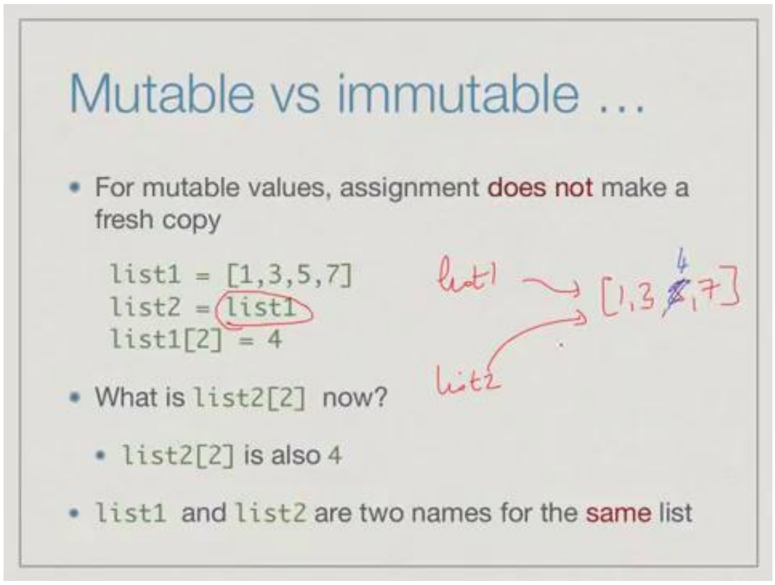
(Refer Slide Time: 15:25)



```
>>> list1 = [1,3,5,7]
>>> list2 = list1
>>> list2
[1, 3, 5, 7]
>>> list1[2] = 4
>>> list1
[1, 3, 4, 7]
>>> list2
[1, 3, 4, 7]
>>> 
```

Let us run this example in python. So, we say list1 is equal to 1, 3, 5, 7 list2 is equal to list1. list2 is indeed 1, 3, 5, and 7. Now we update in place list1, 2 to be equal to 4. We say that list1 is 1, 3, 4, 7, the 5 has been replaced by 4. The question we are asking is has this affected list2 or not and contrary to our intuition that we have the values are copied in which case list1 has indirectly has effected the value of list2 as well. So, why does this happen.

(Refer Slide Time: 16:05)



Mutable vs immutable …

- For mutable values, assignment **does not** make a fresh copy

```
list1 = [1,3,5,7]
list2 = list1
list1[2] = 4
```

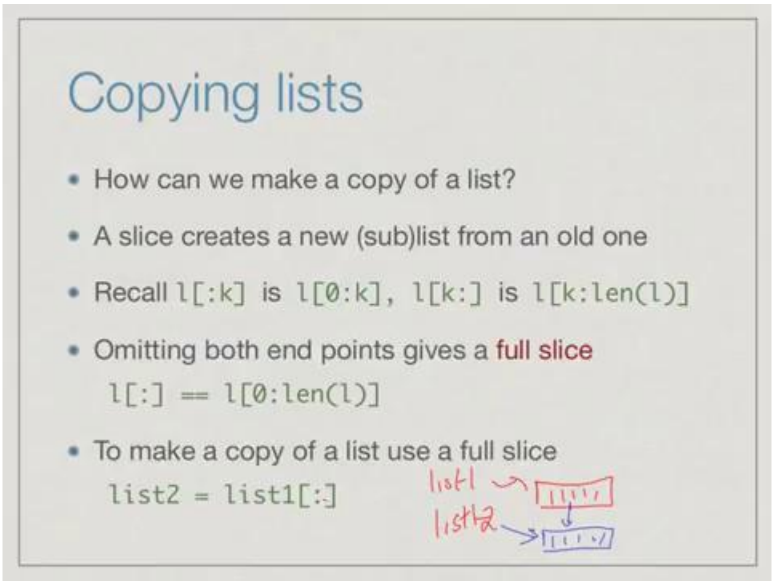
- What is `list2[2]` now?
  - `list2[2]` is also 4
- `list1` and `list2` are two names for the **same** list

list1
list2
[1,3,~~5~~ 4,7]

So, list2 to is also 4 and this that is because, when we actually make assignment like this from one name to another name and the other name holds a mutable value in this case the only mutable type that we have seen so far is a list. Then instead of saying that they are two copies, we actually just say that list1 is pointing if you like to a value of list 1, 3, 5, 7.

And now we also have another name for the same list namely list2. If we go and change this value to 4, then list2 also has same value 4 at this position. There is a fundamental difference will how assignment works for mutable and immutable types. For mutable types we can think of assignment as making a fresh copy of the value and for immutable types and for mutable types assignment does not make a fresh copy it rather makes both names point to exactly the same value. Through either name if we happened to update the mutable value the other name is also effected.

(Refer Slide Time: 17:10)



Copying lists

- How can we make a copy of a list?
- A slice creates a new (sub)list from an old one
- Recall `l[:k]` is `l[0:k]`, `l[k:]` is `l[k:len(l)]`
- Omitting both end points gives a full slice

  `l[:] == l[0:len(l)]`
- To make a copy of a list use a full slice

  `list2 = list1[:]`

list1
list2

This is something which we will see is useful in certain situations, but what if we do not want this to happen what if we want to make a real copy of the list. So, recall that a slice takes a list and returns us a sub list from one position to another position. The outcome of a slice operation is actually a new list, because in general, we take a list and we will take a part of it for some intermediate position to some other intermediate position, so obviously, the new list is different from the old list.

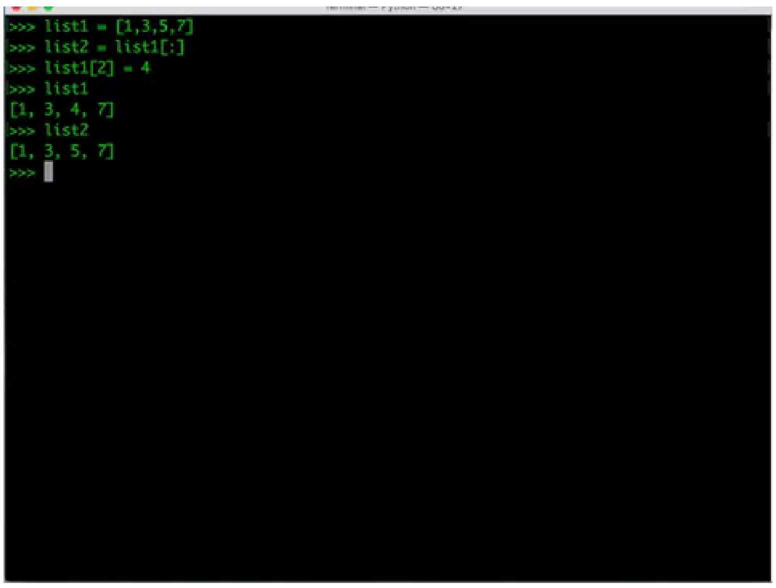
We also saw that when we looked at strings that we can leave out the first position or the last position when specifying a slice. If we leave out the first position as this then we will implicitly say that the first position is 0, so we start at the beginning. Similarly, if we leave out the last position like this, then we implicitly assume that the last position the slice is the length of this list of the string and so it goes to the last possible value.

If we leave out the first position, we get a 0; if we leave out the last position, we get the length. If we leave out both position, we just put colon with nothing before nothing after logically this becomes 0 and this becomes the length. We have both characteristics in the same thing and we call this a full slice.

Now let us combine this observation which is just a short cut notation with this observation that each slice creates a new sub list. So, what we have is that l with just a colon after it is not the same as l it is the new list created from the old list, but it as every value in l in the same sequence. This now gives us a simple solution to copy a list instead of saying list2 is equal to list1, which makes then both.

Remember if I do not have this then I will get list1 and list2 pointing to the same actual list. There will be only 1 list of values and will point to the same. But if I do there have this then the picture changes then what happens is that the slice operation produces a new list which has exactly the same length and the same values and it makes list2 point to that. Therefore, after this list1 and list2 are disjoint from each other any update to list2 will not affect list1 any update to list1 will not affect list2. Let us see how this works in the interpreter to convince ourselves this is actually the way python handles this assignment.

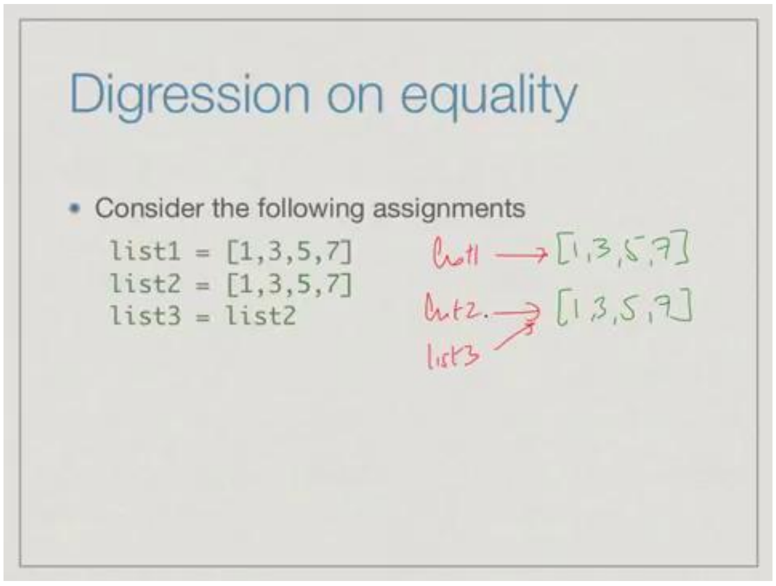
(Refer Slide Time: 19:45)



```
>>> list1 = [1,3,5,7]
>>> list2 = list1[:]
>>> list1[2] = 4
>>> list1
[1, 3, 4, 7]
>>> list2
[1, 3, 5, 7]
>>> 
```

As before let us start with list1 is 1, 3, 5, 7 and list2 now let us say is the slice. So, now, if is update list1 at position 2 to be 4 then list1 looks like 1, 3, 4, 7. But list2 which was a copy is not affected right. When we take a slice we get a new list. So, if we take the entire list as a full slice we get a full copy of the old list and we can assign it safely to a new name and not worry about the fact that both names are sharing the value.

(Refer Slide Time: 20:19)



Digression on equality

- Consider the following assignments

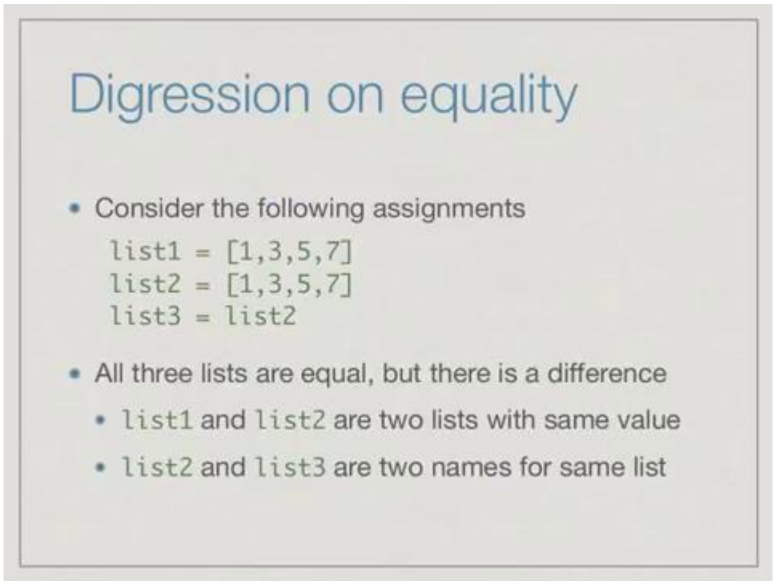
```
list1 = [1,3,5,7]
list2 = [1,3,5,7]
list3 = list2
```

list1 → [1,3,5,7]
list2 → [1,3,5,7]
list3

This leads us to a digression on equality. Let us look now at this set of python statements. We create a list 1, 3, 5, 7 and give with the name list1 and, when we create another list 1, 3, 5, 7, and give it the name list2.

And finally, we assign list3 to be the same values as list2 and this as be said suggest that list3 is actually pointing to the same thing. So, we have now pictorially we have two list of the form 1, 3, 5, 7 stored somewhere. And initially we say that list1 points to this and list2 points to this in the last assignment say that list3 also points to this.

(Refer Slide Time: 21:10)



Digression on equality

- Consider the following assignments

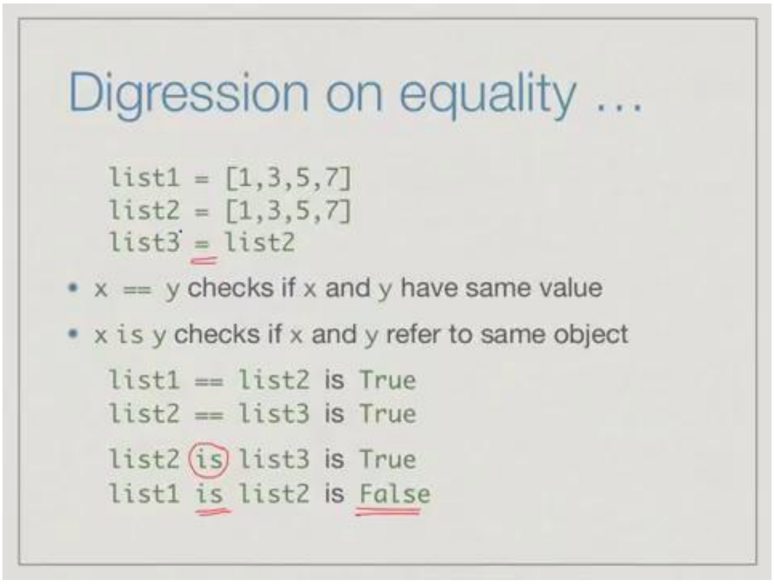
```
list1 = [1,3,5,7]
list2 = [1,3,5,7]
list3 = list2
```

- All three lists are equal, but there is a difference
  - list1 and list2 are two lists with same value
  - list2 and list3 are two names for same list

All three lists are equal, but there is a difference in the way that they are equal. So, list1 and list2 are two different lists, but they have the same value right. So, they happen to have the same value, but they are two different things and so, if we operate on one it need not preserve this equality any more.

On the other hand list2 and list3 are equal precisely because they points to the same value, there is exactly one list in to which they are both pointing. So, if we update list3 or we update list2 they will continue to remain equal. There are two different notions of equality whether the value is the same or the actual underline object that we are referring to by this name is the same. In the second case, updating the object to either name is going to result in both names continuing to be equal.

(Refer Slide Time: 21:57)



Digression on equality ...

```
list1 = [1,3,5,7]
list2 = [1,3,5,7]
list3 = list2
```

- x == y checks if x and y have same value
- x is y checks if x and y refer to same object

```
list1 == list2 is True
list2 == list3 is True

list2 is list3 is True
list1 is list2 is False
```

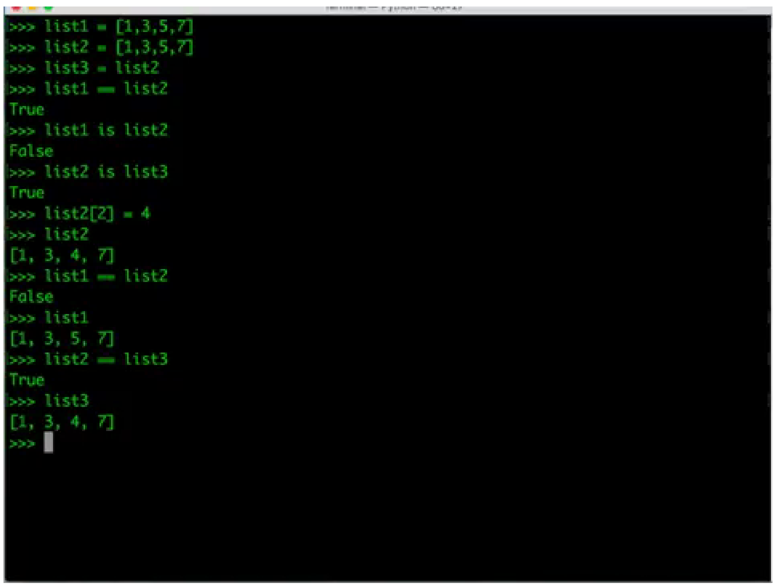
Python has we saw this operation equal to equal to, which is equivalent or the mathematically equality which checks if x and y as names have the same value. So, this will capture the fact that list1 is equal to list2 even though they are two different lists they happen to have the same value.

To look at the second type of equality that list3 and list2 are actually the same physical list in the memory. We have another key word in python called 'is'. So, when we say x is y what we are asking is, whether x and y actually point to the same memory location the same value in which case updating x will effect y and vice versa. We can say that x is y checks if x and y refer to the same object.

Going by this description of the way equal to equal to and is work; obviously, if list2 list3 are the same object they must always be equal to - equal to. So, x is y then x will always equal to equal to y, because there are actually pointing to the same thing. But in this case although list1 list2 are possibly different list they are still equal to - equal to, because the value is the same.

On the other hand if I look at the is operation then list1 list2 is list3 happens to be true, because we have seen that this assignment will not copy the list it will just make list3 point to the same thing is list2. On other hand list1 is list2 is false that is because they are two different list. So, once again its best to verify this for ourselves to convince ourselves that this description is actually accurate.

(Refer Slide Time: 23:41)

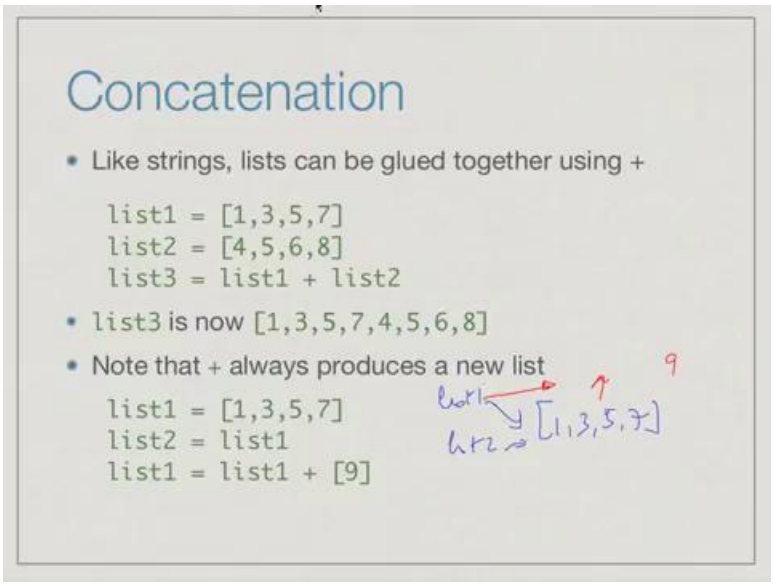


```
>>> list1 = [1,3,5,7]
>>> list2 = [1,3,5,7]
>>> list3 = list2
>>> list1 == list2
True
>>> list1 is list2
False
>>> list2 is list3
True
>>> list2[2] = 4
>>> list2
[1, 3, 4, 7]
>>> list1 == list2
False
>>> list1
[1, 3, 5, 7]
>>> list2 == list3
True
>>> list3
[1, 3, 4, 7]
>>> 
```

Let us type out those three lines of python in the interpreter. So, we say list1 is 1, 3, 5, 7 list2 is also 1, 3, 5, 7 and list3 is list2. Now, we ask whether list1 is equal to list2 and it indeed is true, but if we ask whether list1 is list2 then it says false. So, this means that list1 and list2 are pointing to the same value physically. So, we update one it will not update the other.

On the other hand, if we ask whether list2 is list3 then this is true. If for instance we change list2, 2 to be equal to 4, like we are done in the earlier example then list2 has now become 1, 3, 4, 7. So, if we ask if list1 is equal to list2 at this point as values that's false. Therefore, because list1 continues to be 1, 3, 5, 7 and list2 has become 1 3 4 7; however, if we ask whether list2 is equal to list3 is true that is the case, because list3 is list2 in the sense if they both are the same physical list and so when we updated list3 list2 will also be updated via list3.

(Refer Slide Time: 24:54)



Concatenation

- Like strings, lists can be glued together using +

```
list1 = [1,3,5,7]
list2 = [4,5,6,8]
list3 = list1 + list2
```

- `list3` is now `[1,3,5,7,4,5,6,8]`
- Note that + always produces a new list

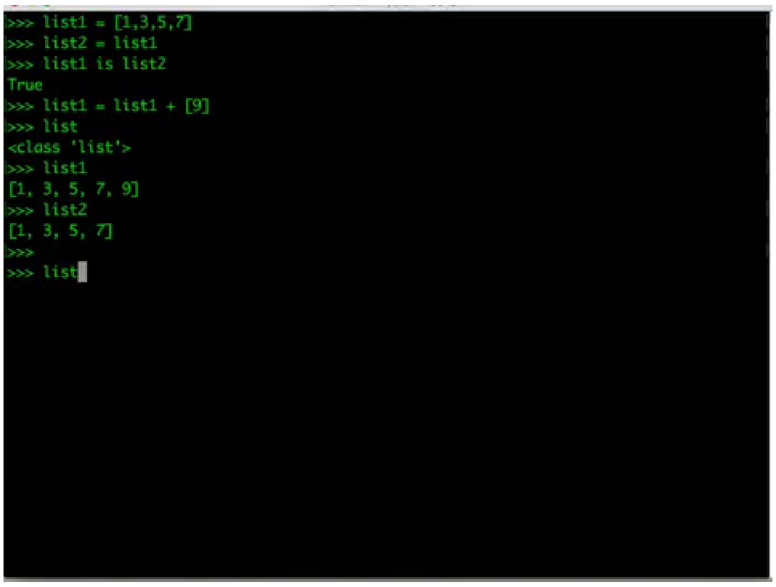
```
list1 = [1,3,5,7]
list2 = list1
list1 = list1 + [9]
```

list1
list2
[1,3,5,7]
9

Like strings, we can combine lists together using the plus operator. So, plus is concatenation. So, if we have list1 is the value 1, 3, 5, 7 list2 is the value 4, 5, 6, 8. Then list3 equal to list1 plus list2 will produce for us the value 1, 3, 5, 7, 4, 5, 6, 8. One important thing to recognise in our context of mutable and immutable values is that plus always produces a new list. If we say that list1 is 1, 3, 5, 7 and then we copy this list as a name to list2. We saw before that we have 1, 3, 5, 7 and we have two names list1 and list2.

Now if we update list1 by saying list1 plus nine this will actually generate a fresh list which has a nine at the end and it will make list1 point their and list2 will no longer be the same right. So, list1 and list2 will no longer point to the same object. Let just confirm this.

(Refer Slide Time: 26:00)

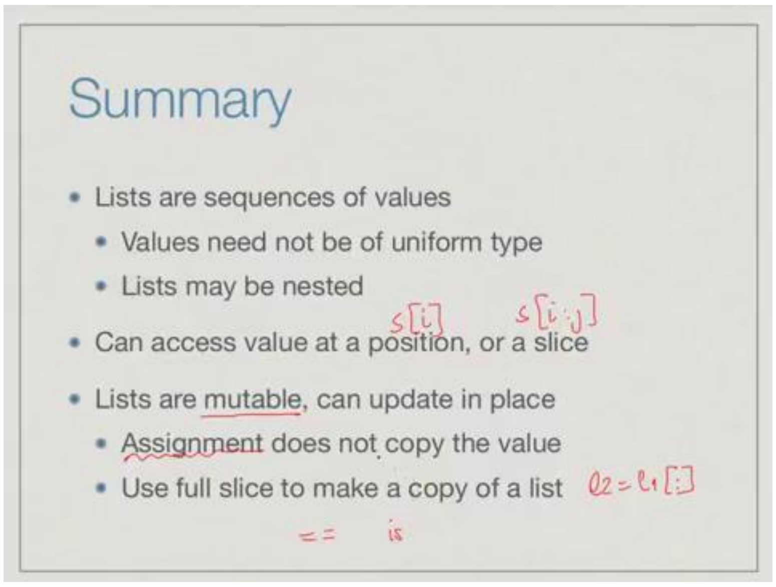


```
>>> list1 = [1,3,5,7]
>>> list2 = list1
>>> list1 is list2
True
>>> list1 = list1 + [9]
>>> list
<class 'list'>
>>> list1
[1, 3, 5, 7, 9]
>>> list2
[1, 3, 5, 7]
>>>
>>> list
```

In the python interpreter let us set up list1 is equal to 1, 3, 5, 7 and say list2 is equal to list1. Then as we saw before if we say list1 is list2 we have true. If on the other hand we reassign list1 to be the old value of list1 plus a new value 9.

This extends, list1 to be 1, 3, 5, 7, 9. Now we will see the list2 is unchanged. So, list1 and list2 have become decoupled because which time we apply plus it is like taking slice. Each time we apply plus we actually get a new list. So, list1 is no longer pointing to the list it was originally pointing to. It is pointing to a new list constructed from that old list with a 9 appended to it at the end.

(Refer Slide Time: 26:49)



Summary

- Lists are sequences of values
  - Values need not be of uniform type
  - Lists may be nested
- Can access value at a position, or a slice
- Lists are mutable, can update in place
  - Assignment does not copy the value
  - Use full slice to make a copy of a list

s[i]
s[i:j]
l2 = l1[:]
==
is

To summarise we have now seen a new type of value called list. So, list is just a sequence of values. These values need not be of a uniform type, we can have mixed list consisting or list, Boolean, integers, strings. Although almost always we will encounter list, where the underline content of a list is of a fixed type. So, all position will actually typically have a uniform type, but this is not required by python and we can nest list. So, we can have list of list and list of list of list and so on.

As with strings, we can use this square bracket notation to obtain the value at a position or we can use the square bracket with colon notation to get a sub list or a slice. One new feature of python, which we introduced with list, is a concept of a mutable value. So, a list can be updated in place we can take parts of a list and change them without effecting the remaining parts it does not create a new list in memory. One consequence is this is that we have to look at assignment more carefully.

For immutable values the types we have seen so far, int, float, bool and string when we say x equal to y the value of y is copied to x. So, updating x does not affect y and vice versa. But when we have mutable values like list we say l2 is equal to l1 then l2 and l1 both point to the same list. So, updating one will update the other, and so we be have little bit careful about this.

If we really want to make a copy, we use a full slice. So, we say l2 is equal to l1 colon with nothing before or after, this is implicitly from 0 to the length of l1, and this gives us a fresh list with exact same contents as l1. And finally, we saw that we can use equality and is as two different operators to check whether two names are equal to only in value or also are physically pointing to the same type.