



IIT Madras

ONLINE DEGREE

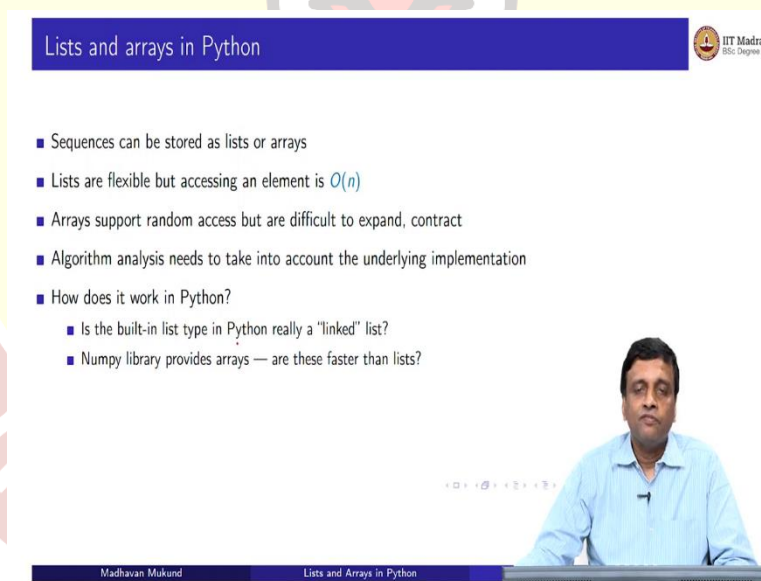
Programing: Data Structures and Algorithms using Python
Professor Madhavan Mukund
Implementation of Lists in Python

So, we saw, that a list is a flexible structure, which is distributed in memory, but which allows us to manipulate things locally so that we can add and delete items in the middle of a list at low cost, as opposed to an array, which actually happens as a contiguous block of memory. So, it is efficient to randomly access the i th element of an array.

But if you have to insert or remove things in an array, then you have to shift everything up or down by one position, so that can be an expensive task. So, arrays are not flexible, but they support random access, lists are flexible, but to get to the i th element, you have to start from the beginning and spend time proportional to the position you want to reach.

So, now in Python, we have this built-in list and we also have a library called numpy, which provides arrays. So, let us try and understand how lists actually work in Python.

(Refer Slide Time: 0:59)



Lists and arrays in Python

- Sequences can be stored as lists or arrays
- Lists are flexible but accessing an element is $O(n)$
- Arrays support random access but are difficult to expand, contract
- Algorithm analysis needs to take into account the underlying implementation
- How does it work in Python?
 - Is the built-in list type in Python really a "linked" list?
 - Numpy library provides arrays — are these faster than lists?

Madhavan Mukund Lists and Arrays in Python

So, we saw that, sequences can be stored either as lists or as arrays, so lists are flexible. You can insert and delete items in the middle of a list quite easily, but getting to the i th element takes time proportional to i . So, in general, if I have to walk across a list to get to the n th element will take me order n time.

On the other hand, arrays are declared contiguously in memory. So, we are starting from the first position, we can compute the offset of the i th element and get there in one step. So, these

arrays support random access, but because they are contiguous and must remain contiguous, we cannot have holes in them. So, if we want to insert an element, we have to push everything. If we delete an element, we have to shrink the array, and this can take again order n time.

So, what we said is, when we do an algorithm like sorting or searching, we do these operations, of looking up $A[i]$, $A[j]$ exchanging them, and so on, so we need to be conscious about which implementation we are using of this list or sequence in order to make sure that the analysis we do is accurate.

So, these are the classical understanding of listen arrays, but now we have a built-in list type in Python. So, the name, of course, suggests that it should be a flexible list. And we do have operations which allow us to insert items at the beginning of the list, to replace a smaller slice by a bigger slice or a bigger slice by a smaller slice, so we can shrink and grow lists quite arbitrarily.

So, the question is, is the built-in list type in Python, really a flexible linked list of the type of implementation that we saw? We also have separately a library called numpy, which provides arrays in Python. So, are these numpy arrays actually faster than lists in Python?

(Refer Slide Time: 02:47)

Lists in Python

- Python lists are not implemented as flexible linked lists
- Underlying interpretation maps the list to an array
 - Assign a fixed block when you create a list
 - Double the size if the list overflows the array
- Keep track of the last position of the list in the array
 - `l.append()` and `l.pop()` are constant time, amortised — $O(1)$
 - Insertion/deletion require time $O(n)$

Handwritten diagrams: A horizontal array with elements being shifted to the right to make space for a new element at the beginning. A vertical array with indices $0, 1, \dots, n-1$ and a new element being added at the top.

Madhavan Mukund Lists and Arrays in Python

So, let us answer the first question. In fact, lists in Python are not implemented as flexible linked lists, rather, they are actually implemented as arrays. So, when you define a list in

Python, Python actually allocates a contiguous block of memory, which is much larger than the list is because the list, of course, when you start, it is typically empty.

So even if you declare an empty list, you get a large block of memory. And till that memory fills up, the list will be accommodated in the array. And once your list grows beyond the size of the array that you have, then you will get a new array, which is double the size. So, effectively, when you're working on a list and the size does not change much, you are manipulating it within an array, and when you get a new list, you grow the list beyond the size of the array, then you get a new array, which is double the size.

So, inside this array, so now we have this array, which we think of is a block of memory. So, this is my L0, L1 and so on. And in general, there will be at the end of the array, some unused space, because the array that is given to me is bigger than the list that I am actually storing in it. So, what Python does is, it keeps track of this position.

So, it keeps track of the last position. So, now, if you want to add an element to the array, it is very simple. You just remove, you put a new element here, you move this pointer here, and now your free space starts at this point. So, extending an array by adding an element to the right is easy. So, extending a list in Python by adding a limit to the right is easy provided, of course, you do not overflow that boundary.

So, Python also provides an operation called pop, which returns the last element. So, it removes the last element and then returns it back to you and shrinks the array by one. And for the same reason it is also easy, because I just have to move this pointer up, declared this to be free and return whatever was there as the return value. So, append and pop are constant time.

Now, we said that it is constant time, but what happens when we have to expand the array and create a new array? So, as I said, at some point, you will have to double the size. So, doubling the size might require you to move, you may not have space right here, so you might have to double the size somewhere else.

So, you might go from an order n size array here to an order $2n$ size array somewhere else in the memory. So, you might have to copy all these n elements and then extend it to the n plus oneth element. But that is a linear operation, that is order n , and that happens once. So, it happens once, when you do this extension beyond that.

So, if I amortize the cost, in order to reach that order n operation, I must have grown the array n times. So, those n adds could have been distributed among the n appends. So, effectively each append we can think of as having taken two steps on an average. So, I do one plus one plus one n times, and then I do one single n . So, this is what happens when I have to expand beyond the array that is given to me.

But if I think of this whole thing as a chunk, this is around two n . And how many operations have I done? I have at least n operations. So, $2n$ steps have been spent over n operation, so each step, each operation roughly took two steps instead of one step. So, that is what we mean by saying that this append and pop.

Because they happened at the end of the array, and only occasionally they cause you to spend a little bit of time transferring the contents from a smaller array to double the size, these are amortized 0 order 1. Order 1 means, remember, order 1 means it is less than c times one for some C , so it just means it takes a constant amount of time.

So, there is no dependence on the input. On the other hand, as you would expect, because this is an array, and exactly the same problem, happens in an array. So, when I have a sequence, and I want to delete this element, I have to shrink everything from the right to the left or if I want to insert an element here, I have to push everything to the right.

So, insertion and deletion in the middle of an array will be expensive, because I have to push everything one position to the right or pull everything back one position to the left. So, insert and delete actually require order n . So, effectively, lists in Python are more like arrays in a normal programming language than lists in a normal programming language.

So, a list in a normal programming language will behave like the flexible linked list structure that we described earlier, where you have these nodes, which point to the next node and so on, and you start from the head, and you have to navigate your way through the list, which will be scattered in memory. That is not how a Python list works.

So, we have programming notation, which makes it look like we are dealing with a flexible structure, as I said, because you can take a slice and shrink it and you can take a slice and increase it. You can insert in the middle of an array, you can pull them apart, put them together. So, it looks like these lists are actually flexible, but the implementation because

most often we use them like arrays, the decision in Python is to keep these as arrays and the underlying implementation.

(Refer Slide Time: 07:48)

Arrays vs lists in Python

- Arrays are useful for representing matrices
- In list notation, these are nested lists

$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ $[[0, 1], [1, 0]]$

Madhavan Mukund Lists and Arrays in Python

Arrays vs lists in Python

- Arrays are useful for representing matrices
- In list notation, these are nested lists
- Need to be careful when initializing a multidimensional list

```
zerolist = [0, 0, 0]
zeromatrix = [zerolist, zerolist, zerolist]
```

$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

Madhavan Mukund Lists and Arrays in Python

So, arrays are useful for representing matrices. So, we saw that, in the maths course, that one of the reasons you will use a matrix, which represent a graph, and we will come across these in this course as well. So, matrices will play a role in terms of dealing with graphs. So, if we take a two-dimensional matrix, for example, which has a row with 01, and another row with 10, the typical way you will represent it using list is to write it out row by row.

I mean, you could also do it column by column, but the convention is row by row. So, you write out the first row. So, this first row is this one, and then, this second row is another list,

and this whole thing is inside a nested list on the outside. So, it is a list of lists. So, each inner list is a row, and the outer list corresponds to the set of columns. This is of course, a square matrix, but in general, you could have a rectangular matrix, so each row will have a fixed length, and the number of columns will be fixed, but the two need not be equal to each other.

So, what happens if we use lists in Python to represent arrays? The problem comes because lists in Python are mutable. So, let us look at this example. Supposing we want to create a three-dimensional array, which looks like this. It has a 3x3 matrix, which has all zeros. So, it might be tempting to first create because it is easy to write down one row.

So, we have one row, which is represented as we said by a list, so each row is a list. So, we write down the first row and call it 000 and then we just write three copies of that, and say that, this is the first row, second row and third row, so now this is an array, which has three rows and three columns. So, the length of each list inside is the number of columns, and the total number of lists inside is the number of rows in my array. So, this is a zero matrix, but there is a problem.

(Refer Slide Time: 09:48)

Arrays vs lists in Python

- Arrays are useful for representing matrices
- In list notation, these are nested lists $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ `[[0,1], [1,0]]`
- Need to be careful when initializing a multidimensional list

```
zerolist = [0,0,0]
zeromatrix = [zerolist,zerolist,zerolist]

zeromatrix[1][1] = 1
print(zeromatrix)
```

Handwritten diagram of a 3x3 matrix:

0	0	0
1	0	0
2	0	0

The diagram shows a 3x3 matrix with rows indexed 0, 1, 2 and columns indexed 0, 1, 2. A green circle highlights the element at row 1, column 1, with an arrow pointing to it from the text '1' in the code snippet.

Madhavan Mukund Lists and Arrays in Python

- Arrays are useful for representing matrices

■ In list notation, these are nested lists $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ `[[0,1], [1,0]]`

- Need to be careful when initializing a multidimensional list

```
zerolist = [0,0,0]
zeromatrix = [zerolist,zerolist,zerolist]

zeromatrix[1][1] = 1
print(zeromatrix)
```

- Mutability *aliases* different values



◀ ▶ 🔍 ↺ ↻

Madhavan Mukund

Lists and Arrays in Python

So, supposing I now take an element in this matrix. So, I take this matrix, and I take the center element. So, the center element, so remember, the columns and rows are numbered from 0, so This is 0 1 2 and this is 0 1 2. So, I take this element, and I assign it to be 1. So, if I now do this, so I just take 0 matrix 1, 1, that is the list number 1, and entry number 1, and assign it to 1.


And if I print the 0 matrix, now suddenly you find that the first row and the last row also have a 1. And the reason for this is because of this way that lists are organized. So essentially, we had this 0 list, which was pointing to 0,0,0. And then when I created 0 matrix, and I said, I have three copies of 0 list, essentially, I have all these three entries pointing to the same object, the same item in memory.

So, now, if I take this through any one of those copies, like if I go through here, and then I replace this by 1, then unfortunately, for the other copies, also that copy is replaced by 1. So, when we have this mutability we unintentionally, sometimes alias different parts of a list or different lists together.

So, when we set up a matrix, we have to do this quite often. We quite often have to initialize a matrix to an mxn or an nxn matrix of 0. And it is tempting to do this, but it is dangerous, you should not do this. So, what we can do, for instance, in Python, the most common way to do this is to use this list comprehension.

(Refer Slide Time: 11:31)

Arrays vs lists in Python




- Arrays are useful for representing matrices
- In list notation, these are nested lists $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ `[[0,1], [1,0]]`
- Need to be careful when initializing a multidimensional list

```
zerolist = [0,0,0]
zeromatrix = [zerolist,zerolist,zerolist]

zeromatrix[1][1] = 1
print(zeromatrix)
```

`[[0, 1, 0], [0, 1, 0], [0, 1, 0]]`
- Mutability *aliases* different values
- Instead, use list comprehension

```
zeromatrix = [ [ 0 for i in range(3) ] for j in range(3) ]
```



Madhavan Mukund Lists and Arrays in Python

So, what you say is that I first create a row of two independent zeros, I mean, three independent zeros, and I do this three times. So, each of these zeros is now a separate 0, and now this actually creates an entry in which all the 9 entries are pointing to distinct elements in memory. So, you need to use this kind of cumbersome notation, in order to work with matrices in this list language of Python.

So, even though the underlying representation is as an array, the notation for lists in Python does not have a convenient way of actually expressing an array as we would like. .we cannot just say I want an mxn array, I want an mxn list, I have to grow that list from an empty list.

(Refer Slide Time: 12:16)

- The Numpy library provides arrays as a basic type

```
import numpy as np
zeromatrix = np.zeros(shape=(3,3))
```

- Can create an array from any sequence type

```
import numpy as np
newarray = np.zeros(shape=(3,3))
```

newarray = np.array([[0,1],[1,0]])

*(0 1)
(1 0)*

◀ ▶ 🔍 ↺ ↻

Madhavan Mukund

Lists and Arrays in Python

So, the solution is to use numpy, which is a library, which provides among other things, arrays. So, this is the normal way you use it. You import numpy, and just to save on space, you normally import it with an alias, so you call it np. So here, for instance, is the same code that we wrote before to get a 0 matrix, which is 3x3 np numpy has a function called zeros.

So, you pass the shape that you want, so the shape tells you how many rows and how many columns. So, this is a function, which returns to you a 3x3 matrix or a 3x3 array of zeros. So, in general numpy allows you to create an array from any kind of a sequence. So, here for instance, is the same thing, which, so this is, this should not be. So, you can say for example, new array is equal to np dot array, and we can take our earlier thing. For example, remember 0,1 and 1, 0.

So, this was the, we had said earlier that this represents that matrix 0,1,1,0. So this nested thing. So, this sequence, if I pass it to this array function in in numpy, this will actually produce an array with exactly that structure. So, you can take any normal Python sequence, either a list or even a tuple, and pass it to this array function, and it will create one array, which is called the corresponding entries, but in in numpy's array format.

(Refer Slide Time: 13:56)

Numpy arrays



- The Numpy library provides arrays as a basic type

```
import numpy as np
zeromatrix = np.zeros(shape=(3,3))
```

- Can create an array from any sequence type

```
import numpy as np
zeroarray = np.zeros(shape=(3,3))
```

- `arange` is the equivalent of `range` for lists

```
row2 = np.arange(5)
```

$[0, 1, 2, 3, 4]$



Madhavan Mukund

Lists and Arrays in Python

So, one of the ways that we can construct lists from say, i to j is to use the range function, so the same thing is there in numpy. And just to distinguish, it is called a range because it is an array range function and not the range function. So, this will create an array, which has 0,1,2, a single row with 0,1,2,3,4. So, then the representation is, I mean, the notation is similar, but typically you have these extra a's hanging around to tell you that this is a numpy function.

(Refer Slide Time: 14:28)

Numpy arrays



- The Numpy library provides arrays as a basic type

```
import numpy as np
zeromatrix = np.zeros(shape=(3,3))
```

- Can create an array from any sequence type

```
import numpy as np
zeroarray = np.zeros(shape=(3,3))
```

- `arange` is the equivalent of `range` for lists

```
row2 = np.arange(5)
```

- Can operate on a matrix as a whole

```
C = 3*A + B
```

```
C = np.matmul(A,B)
```

- Very useful for data science

for all k
 $C[i,j] = A[i,k] \cdot B[k,j]$

$\begin{bmatrix} 3 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 2 & 3 \\ 1 & 4 & 7 \\ 3 & 12 & 21 \end{bmatrix}$

Speaker in a video lecture window

Madhavan Mukund

Lists and Arrays in Python

So, one advantage of working with numpy, is that, you can actually do operations on arrays as a whole without writing these nested loops. If I have to operate on all elements of a nested list representation, then I have to write a for loop which runs through all the rows and all the columns and updates every ij .

Now, if you want to say, take two matrices or two arrays of the same size and add them together element by element, then normally in mathematics, you will write a plus B. And now numpy supports that. So, for example, this will take the array a and multiply every element in A by 3, so this is like a scalar multiplication of every element in A.

So, this takes, for example, if I had 1,0,0,0,2,3,1,4,7 then this will take every entry and make it 3 times. So, this will become 3, this will remain 0, this will become 0, this will become 6,9,3, 12,21. So, this will be the new array which is three times A. And then if I add b, then each element wise will become added and I get a new array C out of this.

So, this kind of a block operation can be expressed on arrays exactly as you would do with numerical variables. And you can also do matrix operations. So, this is the numpy function with just the matrix multiplication. So, if you remember matrix multiplication, you take a row of A and a column of B and then point wise you multiply them and add it up right.

So, you want A_{ik} times B_{kj} , for all k. And this will give me one entry C_{ij} . So, the ij th entry in the target is obtained by taking the i th row the j th column and then adding them up multiplying and adding them. So, these things are very useful because we use these kind of operations a lot in many of data science manipulations.

So, for this reason, numpy has become a very popular library in Python. But for us in this course, our main interest in numpy is that it allows us to write these kinds of definitions to create $N \times N$ arrays quickly without going through this cumbersome list comprehension notation and worrying about whether we are in this mutable aliased mode where something will accidentally disturb something else.

(Refer Slide Time: 16:44)

- Python lists are not implemented as flexible linked structures
- Instead, allocate an array, and double space as needed
- Append is cheap, insert is expensive
- Arrays can be represented as multidimensional lists, but need to be careful about mutability, aliasing
- Numpy arrays are easier to use



Madhavan Mukund

Lists and Arrays in Python

So, to summarize, Python lists despite the name of the data structure, Python lists are not lists in the conventional sense that computing calls lists. They are not these flexible linked lists with nodes pointing to each other, rather, they are implemented as arrays with a pointer to the end of the list within the array.

So, the list starts at the beginning of the array and it gradually grows. And when it overshoots, the amount of space allocated, then the array that is used for the list will be doubled. So, Python will automatically allocate a new array of double the size, move everything there and let you continue. So, in this process, append is cheap, because I am just adding and moving appointed to the right.

And even though I periodically overshoot and have to go across we said this amortized cost remains order 1. On the other hand, if I have to insert something as we will see, if I have to insert something, then I have to push everything. So inserting is actually an expensive operation, because it takes order n time. Then we said that arrays or matrices as we will use are actually can be represented as multi-dimensional lists, but then we have to be careful about this mutability, aliasing and all that.

S

o

,

f

o

r

t

h