

(Refer Slide Time: 04:56)

## Quicksort in practice

*l.sort()*

- In practice, Quicksort is very fast
  - Typically the default algorithm for in-built sort functions
  - Spreadsheets
  - Built in sort function in programming languages

As a result of this because though it is worst case order  $n^2$ , but an average order  $n \log n$ , quicksort is actually very fast. What we saw is it addresses one of the issues with merge sort because by sorting the rearranging in place we do not create extra space. What we have not seen in which you can see if you read up another book somewhere is that we can even eliminate the recursive part we can actually make quicksort operate iteratively over the intervals on which we want to solve.

So, quicksort as a result of this has turned out to be in practice one of the most efficient sorting algorithms and when we have a utility like a spread sheet where we have a button, which says sort this column then more often they are not the internal algorithm that is implemented is actually quicksort we saw that python has a function l dot sort which allows us to sort a list built in. You might ask, for example, what sort is sorting algorithm is python using; very often it will be quicksort. Although, in some cases some algorithm will **decide** on the values in the list and apply different sorting algorithm according to the type of values, but default usually is quicksort.

So, before we proceed let us try and validate our claim that quicksort's worst case behavior is actually tied to the description of the worst case input as in already sorted list.

(Refer Slide Time: 06:22)

```
madhavan@dolphinair:.../week4/python/quicksort$ more quicksort.py
def Quicksort(A,l,r): # Sort A[l:r]
    if r - l <= 1: # Base case
        return()
    # Partition with respect to pivot, a[l]
    yellow = l+1
    for green in range(l+1,r):
        if A[green] <= A[l]:
            (A[yellow],A[green]) = (A[green],A[yellow])
            yellow = yellow + 1
    (A[l],A[yellow-1]) = (A[yellow-1],A[l]) # Move pivot into place
    Quicksort(A,l,yellow-1) # Recursive calls
    Quicksort(A,yellow,r)
madhavan@dolphinair:.../week4/python/quicksort$ more randomize.py
import random
def randomize(l):
    for i in range(len(l)//2):
        j = random.randrange(0,len(l),1)
        k = random.randrange(0,len(l),1)
        (l[j],l[k]) = (l[k],l[j])
madhavan@dolphinair:.../week4/python/quicksort$
```

Here, we have as before our python implementation of quick sort in which we have just repeated the code **we wrote before**. Now, we are going to write another function which will do the following. It will shuffle the elements of a list by a repeatedly picking two indexes and just swapping them. This will allow us to take care of range output just sorted and create a suitably random shuffle of it; here is the code for it.

(Refer Slide Time: 06:52)

```
madhavan@dolphinair:.../week4/python/quicksort$ more randomize.py
import random
def randomize(l):
    for i in range(len(l)//2):
        j = random.randrange(0,len(l),1)
        k = random.randrange(0,len(l),1)
        (l[j],l[k]) = (l[k],l[j])
madhavan@dolphinair:.../week4/python/quicksort$ 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 quicksort import *
>>> from randomize import *
>>> import sys
>>> sys.setrecursionlimit(100000)
>>> l = list(range(7500,0,-1))
>>>
```

It is very simple, you use a python library called random which allows you some functions to generate random numbers and one of the things that this library has is this function randrange. A randrange generates an integer in the range 0 to length of l minus 1. So, we pass it a list and we repeatedly pick two indexes j and k in the range 0 to length of l minus 1 and we exchange lj and lk and how many times we do this? Well we just do it a large number of times in this case say we have 10000 elements in a list, we will do this 5000 times we do it length of l by 2 times.

Let us see how this works, we load as usual the python interpreter and then we import quicksort and then we import randomize. So, you can do this, you can write python functions in multiple files and import them one after the other and they will all get loaded. Now as before we will say l for instance could be the list. So, let us also include sys and finish off that recursion limit process because we know this is gonna kill us. So, set a large recursion limit and now we set up a fairly large list we had done last for an instance 7500 down to 0 right.

(Refer Slide Time: 08:19)

```
>>> Quicksort(l,0,len(l))
>>> Quicksort(l,0,len(l))
>>> randomize(l)
>>> Quicksort(l,0,len(l))
>>>
>>> Quicksort(l,0,len(l))
>>> l = list(range(15000,0,-1))
>>> randomize(l)
>>> Quicksort(l,0,len(l))
>>> |
```

And what we saw was that, if we try to quick sort this list it takes a long time because it is 7500 and it is a worst case in. What we are going to try and do now is and the same thing will happen even after it is sorted because even after it is sorted is still a worst case

input expect now it is an ascending order. So, both descending order and ascending order take a long time. Now, supposing we randomize l. So, if you look at l now you can see that the numbers are no longer in order. So, you see some 6000 between the 7500 and 2000 and so on.

Our claim is that this will go faster and indeed you can see that if you run quicksort on this it returns almost immediately and it is not because quicksort has become any faster, it is because of order of input, because again if we have quicksort on sorted list again it is going to be slow. This just demonstrates in a very effective way that if we randomize the list and we run quicksort it comes out immediately, but if we do not randomize it and if we actually ask to sort the sorted list then it takes a long time.

So, we could actually check that, for instance, if we go back to this list and we make it say even something bigger like 10000 maybe 15000 and then we randomize it and then we sort it right it comes little fast. So, this validates our claim that quicksort on an average is fast, it is only when you give it these very bad inputs which are the already sorted once that it behaves in a poor manner.

(Refer Slide Time: 09:54)

## Stable sorting

- Sorting on multiple criteria
- Assume students are listed in alphabetical order
- Now sort students by marks
  - After sorting, are students with equal marks still in alphabetical order?
- Stability is crucial in applications like spreadsheets
  - Sorting column B should not disturb previous sort on column A

Now, there is one more criterion that one has to be aware of when one is sorting data. So,

very often this sorting happens in stages on multiple attributes, for example, you might have a list of students who are listed in alphabetical order in the roll list after a quiz or a test, they all get marks. Now, you want to list them in order of marks, but where there are ties where two or more students have the same marks you **want to** continue to have them listed in alphabetical order. So, in another words you have an original order in alphabetic order and then you take another attribute namely marks and sorting by a marks should not disturb the sorting that already exists in alphabetical order.

What it amounts to saying is that if we have two list two items in the original list which are equal then they must retain the same order as they had after the sorting. So, I should not take two elements that are equal and, sort, swap them while sorting and this would be crucial when you are using something like a spreadsheet because if you sort by one **column** you do not want to disturb the sorting that you did by another column.

(Refer Slide Time: 11:04)

## Stable sorting ...

- Quicksort, as described, is not stable
  - Swap operation during partitioning disturbs original order
- Merge sort is stable if we merge carefully
  - Do not allow elements from right to overtake elements from left
  - Favour left list when breaking ties

Unfortunately, quicksort the way we have described it is not stable because whenever we extend a partition in this partition stage or move the pivot to the center what we end up doing is disturbing the order of elements which were already there in the unsorted list. So, we argued earlier that disturbing this order does not matter because any way we are going to sort it, but it does matter if the sorting has to be stable. If there was a reason

why these elements were in particular order not for the current attribute, but for the different attribute and we move them around then we are destroying the original sorted order.

On the other hand, merge sort we can see is actually stable if you are careful to make sure that we always pick from one side consistently if the values are equal. So, when we are merging left and right when we have the equal to case we have to either put the element from left into the final list or right. If we consistently choose the left then it will always keep elements on to the left to the left of the ones in the right and therefore, it will remain a stable sort.

Similarly, insertion sort will also be stable if you make sure that we move things backwards only if they are strictly smaller when we go backwards and we find something which is equal to the current value we stop the insertion. So, insertion sort merge sort as stable sort quicksort as we have described it is not stable though it is possible to do a more careful implementation and make it stable.

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

**Week - 04**  
**Lecture - 05**  
**Tuples and Dictionaries**

(Refer Slide Time: 00:01)

The slide has a light gray background with a white rectangular content area. The title 'Tuples' is at the top left in a blue font. Below it is a bulleted list of three items:

- Simultaneous assignments  
`(age, name, primes) = (23, "Kamal", [2,3,5])`
- Can assign a “tuple” of values to a name  
`point = (3.5, 4.8)  
date = (16, 7, 2013)`
- Extract positions, slices  
`xcoordinate = point[0]      3.5  
monthyear = date[1:]      (7, 2013)`

We have seen this kind of simultaneous assignment, where we take three names on the left and assign them to three values in the right, and we enclose these in these round brackets. So, this kind of a sequence of values with the round bracket is called a Tuple.

Normally we talk about pairs, triples, quadruples, but in general when it goes to values of k we call them k tuples. On python, tuples are also valid values. You can take a single name; and assign it a tuple of values. For instance, we can take a two-dimensional point with x coordinates 3.5 and 4.8 and say that point has the value 3.5 comma 4.8, and this is not a list, but a tuple. And we will see in a minute what a tuple is.

Similarly, we can say that a date is made up of three parts a day, a month, and a year; and we can encloses into a three value or triple. So, tuple behaves like a list, so it is a kind of sequence. So, like strings and list, in a tuple you can extract one element of a sequence. So, we can say that the 0th value in point is the x coordinate. This would assign the value 3.5 to the value x to the name x coordinate, or we can take a slice we can say that if we

want only 7 and 2013, we take date and take the slice from one to the end then we will get 7 comma 2013. So, this behaves very much like a different type of sequence exactly like strings and lists we have seen so far, but the difference between a tuple and a list is that a tuple is immutable.

(Refer Slide Time: 01:44)

## Tuples

- Simultaneous assignments  
`(age, name, primes) = (23, "Kamal", [2,3,5])`
- Can assign a “tuple” of values to a name  
`point = (3.5,4.8)  
date = (16, 0, 2013)`
- Extract positions, slices  
`xcoordinate = point[0]  
monthyear = date[1:]`
- Tuples are immutable  
`date[1] = 8` is an error

So, tuple behaves more like a string in this case, we cannot change for instance this date to 8 by **saying** date at position one should be replaced by the value 8. This is possible in a list, but not in a tuple. So, tuples are immutable sequences, and you will see in a minute why this matters.

(Refer Slide Time: 02:10)

## Generalizing lists

- $l = [13, 46, 0, 25, 72]$
- View  $l$  as a function, associating values to positions
  - $l : \{0,1,\dots,4\} \rightarrow \text{integers}$
  - $l(0) = 13, l(4) = 72$
- $0,1,\dots,4$  are **keys**
- $l[0], l[1], \dots, l[4]$  are corresponding **values**

Let us go back to lists. A list is a sequence of values, and implicitly there are positions associated to this sequence starting at 0 and going up to the length of the list minus 1. So, an alternative way of viewing a list is to say that it maps every position to the value; in this case, the values are integers.

We can say that this list  $l$  is a **map** or function in a mathematical sense from the domain 0, 1, 2, 3, 4 to the range of integers; and in particular, it assigns  $l[0]$  to be 13,  $l[4]$  to be 72 and so on where we are looking at this as a function value. So, the program language way of thinking about this is that 0, 1, 2, 3, 4 are what are called **keys**. So, these are the values with which we have some items associated. So, we will search for the item associated with 1 and we get back 46. We have keys and the corresponding **entries** in the list are called **values**. So, a list is one way of associating keys to values.

(Refer Slide Time: 03:19)

## Dictionaries

- Allow keys other than `range(0, n)`
- Key could be a string
  - `test1["Dhawan"] = 84`
  - `test1["Pujara"] = 16`
  - `test1["Kohli"] = 200`
- Python **dictionary** *Associative array*
  - Any immutable value can be a key

We can generalize this concept by allowing keys **from** a different set of things other than just a range of values from 0 to  $n$  minus 1. So, a key for instance could be a string. So, we might want a list in which we index the values by the name of a player. So, for instance, you **might** keep track of the score in a test match by saying that for each player's name what is the value associated. So, Dhawan's score is 84, Pujara's score is 16, Kohli's score is 200, we store these all in a more generic list where the list values are not indexed by position, but by some more abstract key in this case the name of the player.

This is what python calls a dictionary, in some other programming languages this is also called an associative array. So, you might see this in the literature. So, here is a store of values which are accessed through a key which is not just a position, but some arbitrary index and python's rule is that any immutable value can be a key.

(Refer Slide Time: 04:26)

## Dictionaries

- Allow keys other than range( $0, n$ )
- Key could be a string
  - test1["Dhawan"] = 84
  - test1["Pujara"] = 16 72
  - test1["Kohli"] = 200
- Python **dictionary**
  - Any immutable value can be a key
  - Can update dictionaries in place —mutable, like lists

This means that you can use strings which are immutable. And here for instance you can use tuples, but you cannot use **lists as we will see**. And the other feature of a dictionary is that like a list, it is mutable; we can take a value with a key and replace it. So, we can change Pujara's score, if you want by an assignment to 72, and this will just take the current dictionary and replace the value associated to Pujara from 16 to 72. So, dictionaries can be updated in place **and hence are mutable exactly like lists**.

(Refer Slide Time: 04:59)

## Dictionaries

- Empty dictionary is {}, not []
- Initialization: test1 = {}
- Note: test1 = [] is empty list, test1 = () is empty tuple
- Keys can be any immutable values
  - int, float, bool, string, tuple
  - But not lists, or dictionaries

We have to tell python that some name is a dictionary and it is not a list. So, we signify an empty dictionary by curly braces. So, remember we use square brackets for list. So, if you want to initialize that dictionary that we saw earlier then we would first say test 1 is the empty dictionary by giving it the braces here and then we can start assigning values to all the players that we had **before** like Dhawan and Pujara and so on. So, notice that all these three sequences and types of things that we have **are** different, so for strings of course, we use double codes or single codes; for list we use square brackets; for tuples, we use round brackets; and for dictionary, we use braces.

So, there is an unambiguous way of signaling to python what type of a collection we are associating with the name, so that we can operate on it with the appropriate operations that are defined for that type of collection. So, once again for a dictionary, the key can be any immutable value; that means, **your key could be** an integer, it could be a float, it could be a bool, it could be a string, it could be a tuple, what it cannot be is a list or a dictionary. So, we cannot have a value indexed by a list itself or by a dictionary.

(Refer Slide Time: 06:21)

## Dictionaries

- Can nest dictionaries

```
score["Test1"]["Dhawan"] = 84
score["Test2"]["Kohli"] = 200
score["Test2"]["Dhawan"] = 27
```

- Directly assign values to a dictionary

```
score = {"Dhawan":84, "Kohli":200}
score = {"Test1":{"Dhawan":84,
    "Kohli":200}, "Test2":{"Dhawan":50}}
```

So, we can have multiple just like we have **nested** list where we can have a list containing list and then we have two indices take the 0th list and then their first position in the 0 list, we can have two levels of keys. If you want to keep track of scores across multiple test matches, instead of having two dictionaries is we can have one dictionary where the first key is the test match test 1 or test 2, and the second key is a player.

With the same first key for example, with the same different first key for example, test 1 and test 2; you could keep track of two different scores for Dhawan. So, the score in test 1 and the score in test 2. And we can have more than one player in test 2 like we have here; we have both Kohli and Dhawan this one.

If you try to display a dictionary in python, it will show it to you in this bracket in this kind of curly bracket notation, where each entry will be the key followed by the values separated by the colon and then this will be like a list separated by commas. And if we have multiple keys then essentially this is one whole entry in this dictionary, and for the key test 1, I have these values; for the key test 2, I have these values. And internally they are again dictionaries, so they have their own key value.

(Refer Slide Time: 07:39)

```
>>> score = {}
>>> score["Test1"]["Dhawan"] = 76
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Test1'
>>> score["Test1"] = {}
>>> score["Test2"] = {}
>>> score["Test1"]["Dhawan"] = 76
>>> score["Test2"]["Dhawan"] = 27
>>> score["Test1"]["Kohli"] = 200
>>> score
{'Test1': {'Dhawan': 76, 'Kohli': 200}, 'Test2': {'Dhawan': 27}}
>>> |
```

Let us see how it works we start with an empty dictionary say score. And now we want to create keys, so suppose we will say score test 1, Dhawan equal to 76. Now this is going to give us an error, because we have not told it that score test 1 is suppose to be a dictionary. So, it does not know that we can further index with the word Dhawan. So, we have to first tell it that not only score is a dictionary, so is score test 1 and presumably since we will use it, so is score test 2.

Now we can go back and set Dhawan's score in the first test to 76 and maybe you can set the second test to 27 and maybe we can set Kohli's score in the first test to 200. Now, if you ask me to show what scores looks like, we see that it has an outer dictionary with

two keys test 1, test 2 each of which is a nested dictionary. In a nested dictionaries, we have two keys Dhawan and Kohli with scores 76 and 200 as the values. In test 2, has one dictionary entry with Dhawan as a key and 27 is the score.

(Refer Slide Time: 08:52)

## Operating on dictionaries

- d.keys() returns sequence of keys of dictionary d  

```
for k in d.keys():
    # Process d[k]
```
- d.keys() is not in any predictable order  

```
for k in sorted(d.keys()):
    # Process d[k]
```
- sorted(l) returns sorted copy of l, l.sort() sorts l in place
- d.keys() is not a list —use list(d.keys())

If you want to process a dictionary then we would need to run through all the values; and one way to run through value all the values is to extract the keys and extract each value by turn. So, there is a function d dot keys which returns a sequence of keys of a dictionary d. And the typical thing we would do is for every key in d dot keys do something with d square bracket k. So, pick up all the keys.

This is like saying for every position in a list do something the value at that position. This is something for every key in a list do something with a value associated to that. Now one thing we have to keep in mind which I will show in a minute is that d dot keys not in any predictable order. So, dictionaries are optimized internally to return the value with a key quickly. It may not preserve the keys in the order in which they are inserted. So, you cannot predict anything about how d dot keys will be presented to us. One way to do this is to use the sorted function.

We can say for k in sorted d dot keys, process d k, and this will give us the keys in sorted order according to the sort function. So, sorted l is a function we have not seen so far; sorted l returns a sorted copy of l, it does not modify. What we have seen so far is l dot

sort, which is the function which takes a list and updates it in place. So, sorted 1 takes an input list, leaves it unchanged, but it returns a sorted version.

The other thing to keep in mind is that though it is tempting to believe that d dot keys is a list, it is not a list; it is like range and other things. It is just a sequence of values that you can use inside of for, so we must use the list property to actually create a list out of d dot keys.

(Refer Slide Time: 10:46)

```
>>> d = {}
>>> for l in "abcdefghijklmnopqrstuvwxyz":
...     d[l] = l
...
>>> d["a"]
'a'
>>> d["i"]
'i'
>>> d.keys()
dict_keys(['e', 'i', 'g', 'b', 'f', 'd', 'a', 'c', 'h'])
>>> 
```

So, let us validate the claim that keys are not kept in any particular order. So, let us start with an empty dictionary. And now let us create for each letter and entry which is the same as that letter. So, we can say for l in a, b, c, d, e, f, g, h, i, d i, d l is equal to l. So, what it is this saying, so when you say for l in a string it goes to each letter in that string, so want to say d with key a is the value a, d with the key b is the value b and so on right. So, now, if I ask you what is d a, you can a, what is d i, it is i.

Now notice that the keys are inserted in the order a, b, c, d, e, f, g, h, i but if I ask for d dot keys it produces it in some very random order. So, e is first and a is way down and so on. There is no specific order that you can get from this. So, this is just to emphasize that the order in which keys are inserted into the dictionary is not going to be the order in which they are presented to through the keys function. So, you should always ensure that if you want to process the keys in a particular order make sure that you preserve that

order when you extract the keys you cannot assume that the keys will come out in any given order.

(Refer Slide Time: 12:06)

## Operating on dictionaries

- Similarly, d.values() is sequence of values in d

```
total = 0
for s in test1.values():
    total = total + test1 s
```

*for x in l:*

In other way to run through the values in a dictionary is to use d dot values. So, d dot keys returns the key is in some order, d dot values gives you the values in some order. So, this is for example like say for x in l. So, you just get the values you do not get the positions. Here you just get the values you do not get the keys. So, if you want to add up all the values for instance from a dictionary, you can start off by initializing total to 0, and for each value, you can just add it up yes right. So, you can pick up each s in test 1 dot values and add it to the total.

(Refer Slide Time: 12:50)

## Operating on dictionaries

- Similarly, `d.values()` is sequence of values in `d`

```
total = 0
for s in test1.values():
    total = total + test1

• Test for key using in, like list membership
for n in ["Dhawan", "Kohli"]:
    total[n] = 0
    for match in score.keys():
        if n in score[match].keys():
            total[n] = total[n] + score[match][n]
```

So, you can test for a key being in a dictionary by using the `in` operator, just like list when you say `x in l` for a list it tells you true if `x` belongs to `l` the value `x` belongs to `l`, it tells you false otherwise. The same is true of keys. So, if I want to add up the score for individual batsmen, but I do not know, if they have batted in each test match. So, I will say for each of the keys, in this case, Dhawan and Kohli, initialize the dictionary which i have already set up not here I would have set that total is a dictionary. So, total with key Dhawan is 0, total with key Kohli is 0.

Now for each match in our nested dictionary, if Dhawan is entered as a batsman in that match, so if a name Dhawan appears as the key in score for that match then and only **then** you add a score, because if it does not appear it is illegal to access that match. So, this is one way to make sure that when you access a value from a dictionary, the key actually exists, you can use the `in` function.

(Refer Slide Time: 14:00)

## Dictionaries vs lists

- Assigning to an unknown key inserts an entry

```
d = []
d[0] = 7 # No problem, d == {0:7}
```

- ... unlike a list

```
l = []
l[0] = 7 # IndexError!
```

Here is a way of remembering that a dictionary is different from the list. If I start with an empty dictionary then I assign a key, which has not been seen so far, in a dictionary there is no problem it is just equivalent to inserting this key in the dictionary with that value, if `d[0]` already exists it will be updated. So, either you update or you insert. This is in contrast with the list, where if you have an empty list and then try to insert at a position which does not exist, you get an index error.

(Refer Slide Time: 14:42)

## Summary

- Dictionaries allow a flexible association of values to keys
  - Keys must be immutable values
- Structure of dictionary is internally optimized for key-based lookup
  - Use `sorted(d.keys())` to retrieve keys in predictable order
- Extremely useful for manipulating information from text files, tables ... — use column headings as keys

In a dictionary, it flexibly expands to accommodate new keys or updates a key depending on whether the key already exists or not.

To summarize, a dictionary is a more **flexible** association of values to keys **than** you have in a list; **the only constraint** that python imposes is that all keys must be immutable values. You cannot have keys, which are mutable values. So, we cannot use dictionaries or lists themselves as keys, but you can have nested dictionaries with multiple levels of these.

The other thing is that we can use d dot keys to cycle through all the keys in the dictionary, and similarly d dot values, but the order in which these keys emerge from d dot keys is not predictable. So, we need to sort **it** to do something else if we want to make sure to process them in a predictable order.

So, it turns out that you will see that dictionaries are actually something that make python a really useful language for manipulating information from text files or tables, if you have what are called comma separated value tables, it is taken out of spreadsheet because then we can use column headings and **accumulate** values and so on. So, you should understand and **assimilate** dictionary into your programming skills, because this is what makes python really a very powerful language for writing scripts to manipulate it.

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

**Week - 04**  
**Lecture - 06**  
**Function Definitions**

We have seen that we pass values to functions by substituting values for the argument set **when defining** the function.

(Refer Slide Time: 00:02)

## Passing values to functions

- Argument value is substituted for name

```
def power(x,n):  
    ans = 1  
    for i in range(0,n):  
        ans = ans*x  
    return(ans)  
power(3,5)
```

The diagram shows the state of variables during the execution of the `power` function. Red arrows point from the arguments `x = 3` and `n = 5` to their respective assignments in the function body. A green arrow points from the local variable `ans = 1` to its initial value. Another green arrow points from the final value of `ans` back to the `return` statement.

- Like an implicit assignment statement

And, this is effectively the same as having an implicit assignment. So, when we say `power x n`, and we call it values **with** 3 and 5, then we have this assignment `x` equal to 3 and `n` equal to 5. It is not really there, but it is as though, this code is executed by preceding this assignment there and of course, the advantage of calling it as the function is that, we do not have to specify `x` and `n` in the function definition; it comes with the call. So, for different values of `x` and `n`, **we** will execute the same **code**.

(Refer Slide Time: 00:02)

## Pass arguments by name

```
def power(x,n):  
    ans = 1  
    for i in range(0,n):  
        ans = ans*x  
    return(ans)  
• Call power(n=5,x=4)
```

The first thing that python allows us to do flexibly, is to not go by the order; it is not that, the first is  $x$ , and the second is  $n$ ; we can, if you do not remember the order, but we do know the values, the names assigned to **them**, we can actually call them by using the name of the argument.

So, we can even reverse the thing, and say, call power. And I know that,  $x$  is the bottom value I know it is  $x$  to the power  **$n$** , but I do not remember whether  $x$  comes first, or  $n$  comes first. I can say, let us just play safe and say power **of**  $n$  equal to 5,  $x$  equal to 4 and this will correctly associate the value according to the name of the argument and not according to the position.

(Refer Slide Time: 01:24)

The slide has a light gray background with a white rectangular content area. At the top, the title 'Default arguments' is centered in a blue font. Below the title, there is some Python code:

```
def f(a,b,c=14,d=22):  
    ...  
    • f(13,12) is interpreted as f(13,12,14,22)  
    • f(13,12,16) is interpreted as f(13,12,16,22)  
    • Default values are identified by position, must  
      come at the end  
    • Order is important
```

The code shows a function definition with default values for parameters c and d. The bullet points explain how Python interprets function calls with fewer arguments than defined.

Another nice feature of python is that, it allows some arguments to be left out and implicitly have default values. Recall that, we had defined this type conversion function int of s, which will take a string and try to represent it as an integer, if s is a valid representation of an integer. So, we said that, if we give it the string “76”, then, int would convert it to the number 76. If on the other hand, we gave it a string like “A5”, since A is not a valid number, “A5” would actually generate an error.

Now, it turns out that, int is actually not a function of one argument, but two arguments; and the second argument is the base. So, we give it a string and convert it to a number in base b, and if we do not provide b, then, by default b has value 10. So, what is happening in the earlier int conversions is that, it is as though we are saying, int “76” with base 10, but since, we do not provide the 10, python has a mechanism to take the value that is not provided, and substitute to the default value 10.

Now, if we do provide it a value, then, for instance, we can even make sense of “A5”. If you have base 16, if you have studied base 16 ever in school, you would know that, you have the digit zero to 9, but base 16 has numbers up to 15. So, the numbers beyond 9 are usually written using A, B, C, D, E, F. So, A corresponds to, what we would think of is the number 10 in base 10. So, if you write “A5” in base 16, then, this is the sixteenth position and this is the ones

position. So, we have 16 times 10, because the A is 10, plus 5. In numeric terms, this will return 165 correctly.

How does this work in python. This would be how internally, if you were to write a similar function, you would write it. So, you provide the arguments, and for the argument for which you want an optional default argument, you provide the value in the function definition. So, what this definition says is that, int takes 2 arguments s and b and b is assumed to be 10, and is hence, optional; if the person omits the second argument, then it will automatically take the value 10. Otherwise, it will take the value provided by the function call. The default value is provided in the function definition and if that parameter is omitted, then, the default value is used instead. But, one thing to remember is that, this default value is something that is supposed to be available when the function is defined. It cannot be something which is calculated, when the function is called.

So, we saw various functions like Quick sort and Merge sort and Binary search, where we were forced to pass along with the array the starting position and the ending position. Now, this is fine for the intermediate calls, but, when we want to actually sort a list, the first time we have to always remember to call it with zero, and the length of the list. So, it would be tempting to say that, we define the function as something which takes an initial array A as the first argument, and then, by default takes the left boundary to be zero, which is fine, and the right boundary to be the length of A.

But, the problem is that, this quantity, the length of A, depends on A itself. So, when the function is defined, there will be, or may not be a value for A and whatever value you have chosen for A, if there is one, that length will be taken as a default. It will not be dynamically computed each time we call Quicksort. So, this does not work, right. So, when you have default values, the default value has to be a static value, which can be determined when the definition is read for the first time, not when it is executed.

Here is a simple prototype. Suppose we have a function with 4 arguments a, b, c, d and we have, c has the default value 14, and d has a default value 22. Then, if you have a call with just 2 arguments, then, this will be associated with a and b, and so, this will be interpreted as f 13, 12, and for the missing argument c and d, you get the defaults 14 and 22. On the other hand, you

might provide 3 arguments, in which case, a becomes 13, b becomes 12 as before, and c becomes 16, but d is left unspecified; so, it picks up the default value.

This is interpreted as f of 13, 12, 16 and the default value 22. So, the thing to keep in mind is that, the default values are given **by position**. There is no way in this function to say that, 16 should be given for d, and I want the default value for c; you can only drop values by position from the end. So, if I have 2 default values, and if I want to only specify the second of them, it is not possible; I will have to redefine the function to reorder it.

Therefore, you must make sure that, when you use these default values, they come at the end, and they are identified by position. And do not mix it up, and do not confuse yourself by combining these things randomly. So, the order of the arguments is important.

(Refer Slide Time: 06:43)

## Function definitions

- Can assign a function to a new name

```
def f(a,b,c):  
    ...  
g = f
```

g(a,b,c)
- Now g is another name for f

A function definition associates a function body with a name. It says, the name f will be interpreted as a function which takes some arguments and does something. In many ways, python interprets this like any other assignment of a value to a name. For instance, this value could be defined in different ways, multiple ways, in conditional ways. So, as you go along, a function can be redefined, or it can be defined in different ways depending on how the computation proceeds. Here is an example of a conditional definition. You have a condition; if it

is true, you define f one way; otherwise, you define f another way. So, depending on which of these conditions held when this definition was executed, later on the value of f will be different.

Now, this is not to say that, this is a desirable thing to do, because you might be confused as to what f is doing. But, there are situations where you might want to write f in one way, or another way, depending on how the computation is proceeding; and python does allow you to do this. Probably, at an introductory take to python, this is not very useful; but, this is useful to know that such a possibility exists and in particular, you can go on and redefine f as you go ahead.

Another thing you can do in python, which may seem a bit strange to you, is you can take an existing function, and map it to a new name. So, we can define a function f, which as we said, associates with the name f, the body of this function; at a later stage, we can say g equal to f. And what this means is now that, we can also use g of a, b, c and it will mean the same as f of a, b, c. So, if you use g in the function, it will use exactly the same function as a, its exactly like assigning one list to another, or one dictionary to another and so on. Now, why would you want to do this? So, one useful way in which you can do this, use this facility is to pass a function to another function.

(Refer Slide Time: 08:40)

## Can pass functions

f = square

- Apply f to x n times

```
def apply(f,x,n):
    res = x
    for i in range(n):
        res = f(res)
    return(res)

def square(x):
    return(x*x)

apply(square,5,2)
square(square(5))
```

625

Suppose, we want to apply a given function  $f$  to its argument  $n$  times, then we can write a generic function like this called `apply`, which takes 3 arguments. The first is the function, the second is the argument, and the third is the number of times, the repetitions. So, we start with the value that you are provided, and as many times as you are asked to, you keep iterating function  $f$ . So, let us look at a concrete example.

Supposing, we have defined a function `square` of  $x$ , which just returns  $x$  times  $x$ ; and now we can say, apply `square` to the value 5 twice. So, what this means is, apply `square` of 5, and then, `square` of that; so, do `square` twice. Therefore, you get 5 squared 25; 25 squared 625. So, what is happening here is that, `square` is being assigned to  $f$ , 5 is being assigned to  $x$ , and 2 is being assigned to  $n$ . This is exactly as we said like, before, like, saying  $f$  is equal to `square`. So, in this sense, being able to take a function name and assign it to another name is very useful, because, it allows us to pass functions from one place to another place, and execute that function inside the another function, without knowing in advance what that function is.

(Refer Slide Time: 10:08)

## Passing functions

- Useful for customizing functions such as `sort`
- Define `cmp(x,y)` that returns -1 if  $x < y$ , 0 if  $x == y$  and 1 if  $x > y$ 
  - `cmp("aab", "ab")` is -1 in dictionary order
  - `cmp("aab", "ab")` is 1 if we compare by length
- `def sortfunction(l, cmpfn=defaultcmpfn):`

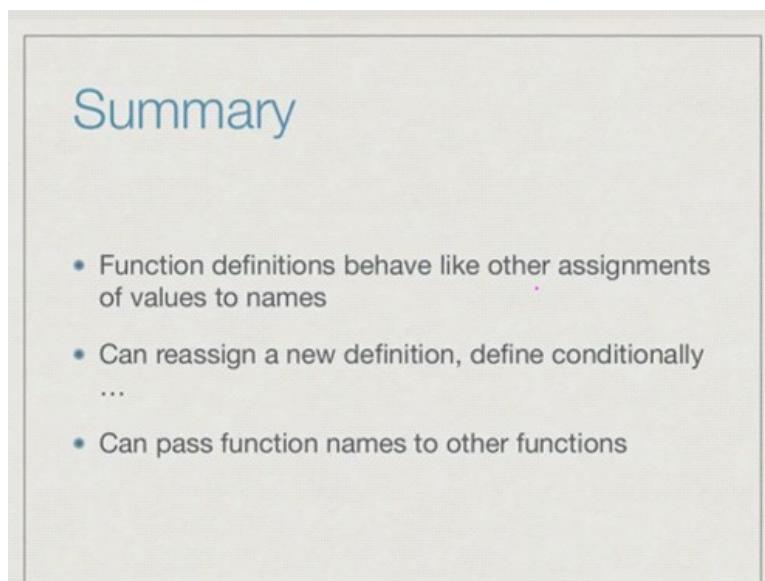
One practical use of this is to customize functions such as `sort`. Sometimes, we need to sort values based on different criteria. So, we might have an abstract compare function, which returns minus 1 if the first argument is smaller, zero if the 2 arguments are equal, and plus 1 if the first argument is bigger than the second. So, when comparing strings, we may have 2 different ways

of comparing strings in mind, and we might want to check the difference, when we sort by these 2 different ways.

We might have one sort in which we compare strings in dictionary order. So, string like aab will come before ab, because, the second position a is smaller than b. So, this will result in minus 1, because, the first argument is smaller than the second argument. If, on the other hand, we want to compare the strings by length, then, the same argument would give us plus 1, because, aab has length 3 and is longer than ab. So, we could write a sort function, which takes a list, and takes a second argument, which is, how to compare the **entries in the list**.

The sort function itself does not need to know what the elements in a list are; whenever it is given a list of arbitrary values, it is also told how to compare them. So, all it needs to do is, apply this function to 2 values, and check if their answer is minus 1, zero, or plus 1 and interpret it as less than, equal to or greater than. Then, if you want, you can combine it with the earlier feature, which is, you can give it a default function. If you do not specify a sort function, there might be an implicit function that the sort function uses; otherwise it will use the comparison function that you provide.

(Refer Slide Time: 11:50)



The slide has a light gray background with a title 'Summary' at the top left. Below the title is a bulleted list of four items:

- Function definitions behave like other assignments of values to names
- Can reassign a new definition, define conditionally  
...  
• Can pass function names to other functions

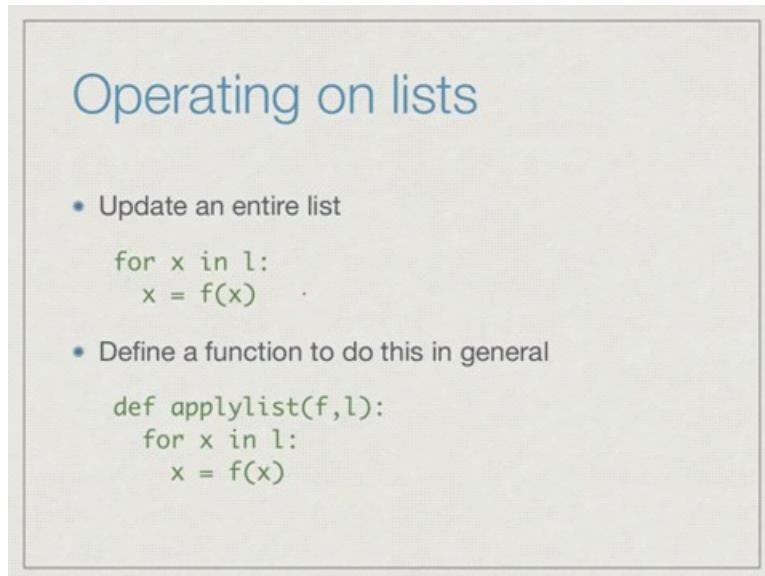
To summarize, function definitions behave just like other assignments of values to names. You can reassign a new definition to a function. You can define it conditionally and so on. Crucially, you can use one function and make it point, name point to another function, and this is implicitly used when we pass functions to other functions and in situations like sorting, you can make your sorting more flexible by passing your comparison function which is appropriate to the values we will sort.

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

**Week - 04**  
**Lecture - 07**  
**List Comprehension**

Quite often, we want to do something to an entire list.

(Refer Slide Time: 00:02)



**Operating on lists**

- Update an entire list

```
for x in l:  
    x = f(x) .
```
- Define a function to do this in general

```
def applylist(f,l):  
    for x in l:  
        x = f(x)
```

For instance, we might want to replace every item in the list by some derived value  $f$  of  $x$ . So, we would write a loop as follows, for every  $x$  in it, replace  $x$  by  $f$  of  $x$ . Now, we could write a function to do this, which does this for different lists and different values of  $l$ . We could say, define `applylist`, which takes the function  $f$  and the list  $l$ , and for every  $x$  and  $l$ , you just replace this  $x$  by  $f$  of  $x$ ; and since  $l$ , list is a **mutable** item, this will update list in the calling function as well.

(Refer Slide Time: 00:45)

## Built in function `map()`

- `map(f, l)` applies `f` to each element of `l`
- Output of `map(f, l)` is not a list!
  - Use `list(map(f, l))` to get a list
  - Can be used directly in a `for` loop

```
for i in list(map(f, l)): ...
```
- Like `range(i, j)`, `d.keys()`

Python has a built-in function map, which does precisely this. So, map f l applies f, in turn to each element of l. Now, although you would think that, if you take a list, say, x 1, x 2, and you apply map, and you get f of x 1, f of x 2, that the output of map should be another list, unfortunately, in python 3, and this is another difference between python 3 and python 2, the output of map is not a list. So, you need to use the list function like we did before. So, you need to say list of map f l to get a list, and you can however, use the output of map directly in a for loop, by saying, for i in list map f l or you can even say for i in map f l, this will work.

So, you do not need to use the list notation, if you just wanted to index menu, but if you want to use it as a list, you must use the list function to convert it. And, this is pretty much what happens, with functions like range and d dot keys and so on. These are all things which give us sequences of values. These sequences are not absolutely lists; they can be used in for functions but if you want to use them as lists, and manipulate them as lists, you must use list to convert them from their sequence to the list form.

(Refer Slide Time: 02:10)

## Selecting a sublist

- In general

```
def select(property,l):  
    sublist = []  
    for x in l:  
        if property(x):  
            sublist.append(x)  
    return(sublist)
```

- Note that `property` is a function that returns `True` or `False` for each element

Another thing that we typically want to do is to take a list and extract values that satisfy a certain property. So, we might have a list of integers called number list, and from this, we might want to extract the list of primes. We start off by saying that, the list of primes we want is empty, and we run through the number list, and for each number in that list, we apply the test, is it a prime; if it is a prime, then we append the list to our output list. So, we start with a list x 1, x 2 and so on.

And then, we apply the test and some of them will pass, and some of them will succeed, some of them will fail, and at the end, wherever the things pass, those items will emerge in the output. So, in general, we could write a select function which takes the property and a list, and it creates a sub list by going through every element in the list, checking if the property holds, and for those elements which the property holds, appending it to the sub list. The difference between select and our earlier map function is that, `property` is not an arbitrary function; it does not manipulate **at all**, all it does is, it checks whether the property is true or not. The property will be a function which takes an element in the list, and tells us true or false; if it is true, it gets copied to the output; if it is false, it gets discarded.

(Refer Slide Time: 03:46)

## Built in function filter()

- `filter(p,l)` checks `p` for each element of `l`
- Output is sublist of values that satisfy `p`

There is a built-in function for this as well. It is called filter. So, filter takes a function `p`, which returns true or false for every element, and it pulls out precisely that sublist of `l`, for which every item in `l`, which falls into the sublist satisfies `p`. Let us look at a concrete example.

(Refer Slide Time: 04:00)

## Combining map and filter

- Sum of squares of even numbers from 0 to 99

```
list(map(square, filter(iseven, range(100)))  
def square(x):  
    return(x*x)  
  
def iseven(x):  
    return(x%2 == 0)
```

Supposing, we have the list of numbers from 0 to 99. We want to first pull out only the even numbers in the list. That is a filter operation; and then, for each of these even numbers, we want

to square them. So, here, we take the even numbers, right, by using the filter, and then, we map square. Then, we get a list.

And then, of course, having got this list, then we can add it up. The sum is not the part of this function. If we want to first extract the squares of the even numbers, and that can be done using a combination of filter, and then, map. Filter, first gives us the even numbers and then map gives us the squares and the square is defined here and this even is defined here.

(Refer Slide Time: 04:46)

## List comprehension

- Squares of even numbers below 100

```
[square(x) for i in range(100) if iseven(x)]
```

map            generator            filter

There is a very neat way of combining map and filter, without using that notation. Let us get to it, through a simpler mathematical example. So, you might have studied in school, from right hand, right angled triangles that, by Pythagoras' theorem, you know that, if x, y and z are the lengths of the two sides and the hypotenuse respectively, then, x square plus y square will be z square. So, Pythagorean triple is a set of integers, say 3, 4 and 5, for example, such that, x square plus y square is z square; 3 square is 9; 4 square is 16; 5 square is 25. Let us say, we want to know all the integer values of x, y and z, whose values are below n, such that, x, y and z form a Pythagorean triple **instance**.

In conventional mathematical notation, you might see this kind of expression. It says, give me all triples x, y and z, such that this bar stands for such that; such that, x, y and z, all lie between 1

and n. And, in addition, x square plus y square is equal to z square. This is, in some sense, where we get the values from; this is an existing set. We have x ranging from 1 to n, y ranging from 1 to n, z ranging from 1 to n, and we put together all possible combinations, then we take out those combinations to satisfy a given property, x square plus y square is equal to z square, and those are the ones that we extract out.

In set theory, this is called set comprehension. This is the way of building a new set by applying some conditional things to an old set. This is also implicitly applying a kind of a tripling operator; it takes 3 separate sets, x from 1 to n, y from 1 to n, z from 1 to n, combines them into triples. There is a filtering process by which you only pull out those triples, where x square plus y square is z square; and then, there is a manipulating step, where you combine them into a single triple, x comma y comma z.

But, in general, the main point is that you are building a new set from existing sets. So, what python does and many other languages also, from which python is inspired to, is allow us to extend this notation to lists. This actually comes from a style of programming called functional programming, which, from where this kind of a notation is there, and python has borrowed it and it works quite well.

Here is how you will write our earlier thing, which we had said, the squares of the even numbers below 100. Earlier, we had given a map filter thing. So, we had said, we will take a range, and we will filter it progressively, and then, we would do a map of square. In python, there is an implicit perpendicular line below, before the 'for' from the set notation. It just takes a square of x, for i in range 100, such that, iseven of x - we have here 3 parts. So, we have a generator, which tells us where to get the values from. Remember that, list comprehension or set comprehension, pulls out values from an existing set of lists, so we first generate a list. In this case, the list range 100, but we could use our other lists; we could use for i in any one, just like a 'for'.

Then, we will apply a filter to it, which are the values in this list, which you are going to retain. And then, for each of those values we can do something to it. In this case, we squared and that will be our output. This is how we generate a list using map and filter without using the words map and filter in between, you just use the 'for' for the generator, 'if' for the filter, and the map is implicit by just applying a function to the output of the generator in the filter.

(Refer Slide Time: 08:52)

## Multiple generators

- Pythagorean triples with x,y,z below 100

```
[ $(x,y,z)$  for  $x$  in range(100)  
    for  $y$  in range(100)  
        for  $z$  in range(100)  
            if  $x*x + y*y == z*z]$ 
```
- Order of x,y,z is like nested for loop

```
for  $x$  in range(100): 0  
    for  $y$  in range(100): 0,  
        for  $z$  in range(100): 0,1..99
```

x	y	z
0,0,0		
0,0,1		
	0,0,99	
	0,1,0	
	0,1,99	

Let us go back to the Pythagorean triple example. We want all Pythagorean triples with x, y, z below 100. This, as we said, requires us to cycle through all values of x, y, and z in that range. It is a little bit more complicated than the one we did before, where we only had a single generator, all the values in range 0 to 100. It is simple enough to write it with multiple forms. So, we say, I want x comma y comma z, for x in range 100, for y in range 100, for z in range 100, provided, x squared plus y squared is equal to z squared. That is written with the 'if'. Now, just to fit on the slide, I have split it up into multiple lines, but actually, this will be a single line of python code.

In what order will these be generated? Well, it will behave exactly like a nested loop. Imagine, we had written a loop in which we had said, for x in range 100, for y in range 100, for z in range 100, and so on. So, what happens here is that, first - a value of 0 will be fixed for x, and then a value of 0 will be fixed for y, then 0 for z. In the first pair, triple that comes out is 0, 0, 0. Then, the value of z will change, the innermost loop changes next. The next one will be 0, 0, 1. This is x, this is y, this is z.

So, in this way, we will keep going until it will be 0, 0, 99. So, when this hits 99, then this for loop will exit and we go to 1. So, I will get 0, 1, 0, and to 0, 1, 90, 0, 1, 99, and so on. The innermost for, so z will cycle first, then y, and then x will cycle slowest. So, just remember that.

(Refer Slide Time: 10:35)

```
(87), (0, 88, 88), (0, 89, 89), (0, 90, 90), (0, 91, 91), (0, 92, 92), (0, 93, 93), (0, 94, 94), (0, 95, 95), (0, 96, 96), (0, 97, 97), (0, 98, 98), (0, 99, 99), (1, 0, 1), (2, 0, 2), (3, 0, 3), (3, 4, 5), (4, 0, 4), (4, 3, 5), (5, 0, 5), (5, 12, 13), (6, 0, 6), (6, 8, 10), (7, 0, 7), (7, 24, 25), (8, 0, 8), (8, 6, 10), (8, 15, 17), (9, 0, 9), (9, 12, 15), (9, 40, 41), (10, 0, 10), (10, 24, 26), (11, 0, 11), (11, 60, 61), (12, 0, 12), (12, 5, 13), (12, 9, 15), (12, 16, 20), (12, 35, 37), (13, 0, 13), (13, 84, 85), (14, 0, 14), (14, 48, 50), (15, 0, 15), (15, 8, 17), (15, 20, 25), (15, 36, 39), (16, 0, 16), (16, 12, 20), (16, 30, 34), (16, 63, 65), (17, 0, 17), (18, 0, 18), (18, 24, 30), (18, 80, 82), (19, 0, 19), (20, 0, 20), (20, 15, 25), (20, 21, 29), (20, 48, 52), (21, 0, 21), (21, 20, 29), (21, 28, 35), (21, 72, 75), (22, 0, 22), (23, 0, 23), (24, 0, 24), (24, 7, 25), (24, 10, 26), (24, 18, 30), (24, 32, 40), (24, 45, 51), (24, 70, 74), (25, 0, 25), (25, 60, 65), (26, 0, 26), (27, 0, 27), (27, 36, 45), (28, 0, 28), (28, 71, 35), (28, 45, 53), (29, 0, 29), (30, 0, 30), (30, 16, 34), (30, 40, 50), (30, 72, 78), (31, 0, 31), (32, 0, 32), (32, 24, 40), (32, 60, 68), (33, 0, 33), (33, 44, 55), (33, 56, 65), (34, 0, 34), (35, 0, 35), (35, 12, 37), (35, 84, 91), (36, 0, 36), (36, 15, 39), (36, 27, 45), (36, 48, 60), (36, 77, 85), (37, 0, 37), (38, 0, 38), (39, 0, 39), (39, 52, 65), (39, 80, 89), (40, 0, 40), (40, 9, 41), (40, 30, 50), (40, 42, 58), (40, 75, 85), (41, 0, 41), (42, 0, 42), (42, 40, 58), (42, 56, 70), (43, 0, 43), (44, 0, 44), (44, 33, 55), (45, 0, 45), (45, 24, 51), (45, 28, 53), (45, 60, 75), (46, 0, 46), (47, 0, 47), (48, 0, 48), (48, 14, 50), (48, 20, 52), (48, 36, 60), (48, 55, 73), (48, 64, 80), (49, 0, 49), (50, 0, 50), (51, 0, 51), (51, 68, 85), (52, 0, 52), (52, 39, 65), (53, 0, 53), (54, 0, 54), (54, 72, 90), (55, 0, 55), (55, 48, 73), (56, 0, 56), (56, 33, 65), (56, 42, 70), (57, 0, 57), (57, 76, 95), (58, 0, 58), (59, 0, 59), (60, 0, 60), (60, 11, 61), (60, 25, 65), (60, 32, 68), (60, 45, 75), (60, 63, 87), (61, 0, 61), (62, 0, 62), (63, 0, 63), (63, 16, 65), (63, 60, 82), (64, 0, 64), (64, 48, 80) )
```

Let us see how this works in python. Let us first begin by defining square; a square of x return x times x; then, we can define iseven x, to check that the remainder of x divided by 2 is 0. So, we have square 8, 64; iseven 67 should be false; iseven 68 should be true and so on. Now, we have list comprehension. Let us look at the set of square x, for x in range 100, such that x is even. So, we see now that, 0 is there. So 0 square, 2 square, 4 square, 6 square, and so on. This is our list comprehension.

Now, let us do the Pythagorean triple one. We said, we want x, y, z, for x in range 100, y in range 100, for z in range 100. This is our 3 generators, with the condition that x times x, plus y times y, is equal to z times z. Now, you see a lot of things which have come. In particular, you should see in the early stages somewhere, things which we are familiar with, like 3, 4, 5, and so on. But, you also see some nonsensical figure, that 4, 0, 4.

So, we should probably have done this better, but you will not worry about that but, what I want to emphasize is that, you see things like, say, you see 0, 77, 77, which is a stupid one; but, let us see, for instance, you say, you see 3, 4, 5. So, we saw 3, 4, 5, somewhere - so 3, 4, 5. But, you will also see, later on 4, 3, 5. Now, one might argue that, 3, 4, 5, and 4, 3, 5, are the same triplets. So, how do we eliminate this duplicate?

(Refer Slide Time: 12:46)

## Multiple generators

- Later generators can depend on earlier ones
- Pythagorean triples with x,y,z below 100, no duplicates

```
[(x,y,z) for x in range(100)
           for y in range(x,100)
           for z in range(y,100)
           if x*x + y*y == z*z]
```

So, we can have a situation, just like we have in a 'for loop', where the later loop can depend on an earlier loop; if the outer loop says, i to some, i goes from something to something, the later loop can say that, j starts from i, and goes forward. For instance, we can now rewrite our Pythagorean triples to say that, x is in range 100, but y does not start at 0; it starts from x onwards. So, y is never smaller than x, and z is never smaller than y. So, z is also never smaller than x, because y itself is never smaller than x, and this version will actually eliminate duplicates.

(Refer Slide Time: 13:21)

```
x*x + y*y == z*z ]  
[(0, 0, 0), (0, 1, 1), (0, 2, 2), (0, 3, 3), (0, 4, 4), (0, 5, 5), (0, 6, 6), (0  
, 7, 7), (0, 8, 8), (0, 9, 9), (0, 10, 10), (0, 11, 11), (0, 12, 12), (0, 13, 13  
, (0, 14, 14), (0, 15, 15), (0, 16, 16), (0, 17, 17), (0, 18, 18), (0, 19, 19),  
(0, 20, 20), (0, 21, 21), (0, 22, 22), (0, 23, 23), (0, 24, 24), (0, 25, 25), (0  
, 26, 26), (0, 27, 27), (0, 28, 28), (0, 29, 29), (0, 30, 30), (0, 31, 31), (0,  
, 32, 32), (0, 33, 33), (0, 34, 34), (0, 35, 35), (0, 36, 36), (0, 37, 37), (0, 3  
, 38), (0, 39, 39), (0, 40, 40), (0, 41, 41), (0, 42, 42), (0, 43, 43), (0, 44,  
, 44), (0, 45, 45), (0, 46, 46), (0, 47, 47), (0, 48, 48), (0, 49, 49), (0, 50, 5  
0), (0, 51, 51), (0, 52, 52), (0, 53, 53), (0, 54, 54), (0, 55, 55), (0, 56, 56)  
, (0, 57, 57), (0, 58, 58), (0, 59, 59), (0, 60, 60), (0, 61, 61), (0, 62, 62),  
(0, 63, 63), (0, 64, 64), (0, 65, 65), (0, 66, 66), (0, 67, 67), (0, 68, 68), (0  
, 69, 69), (0, 70, 70), (0, 71, 71), (0, 72, 72), (0, 73, 73), (0, 74, 74), (0,  
, 75, 75), (0, 76, 76), (0, 77, 77), (0, 78, 78), (0, 79, 79), (0, 80, 80), (0, 81  
, 81), (0, 82, 82), (0, 83, 83), (0, 84, 84), (0, 85, 85), (0, 86, 86), (0, 87,  
, 87), (0, 88, 88), (0, 89, 89), (0, 90, 90), (0, 91, 91), (0, 92, 92), (0, 93, 93  
, (0, 94, 94), (0, 95, 95), (0, 96, 96), (0, 97, 97), (0, 98, 98), (0, 99, 99),  
(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9, 12, 15), (9,  
, 40, 41), (10, 24, 26), (11, 60, 61), (12, 16, 20), (12, 35, 37), (13, 84, 85), (1  
4, 48, 50), (15, 20, 25), (15, 36, 39), (16, 30, 34), (16, 63, 65), (18, 24, 30  
, (18, 80, 82), (20, 21, 29), (20, 48, 52), (21, 28, 35), (21, 72, 75), (24, 32  
, 40), (24, 45, 51), (24, 70, 74), (25, 60, 65), (27, 36, 45), (28, 45, 53), (30  
, 40, 50), (30, 72, 78), (32, 60, 68), (33, 44, 55), (33, 56, 65), (35, 84, 91),  
(36, 48, 60), (36, 77, 85), (39, 52, 65), (39, 80, 89), (40, 42, 58), (40, 75,  
, 85), (42, 56, 70), (45, 60, 75), (48, 55, 73), (48, 64, 80), (51, 68, 85), (54,  
, 72, 90), (57, 76, 95), (60, 63, 87), (65, 72, 97)]
```

Here is our earlier definition of Pythagoras, where we had x, y, and z unconstrained. So, what I do is, I go back, and I say that, y is not in range 100, but y is in range x to 100, and z is in range y to 100. And now, you will see a much smaller list and in particular you will see that, in every sequence that is generated, x is less than or equal to y is less than equal to z; you only get one copy of things like 3, 4, 5. So, you see 3, 4, 5, but you do not see 4, 3, 5; 3, 4, 5 is here. Next one is 5, 12, 13; 4, 3, 5 is eliminated. The key thing is that, generators can be dependent on outer generators - inner generators can be dependent on outer generators.

(Refer Slide Time: 14:06)

## Useful for initialising lists

- Initialise a  $4 \times 3$  matrix
  - 4 rows, 3 columns
  - Stored row-wise

```
l = [ [ 0 for i in range(3) ]
      for j in range(4)]
```

*↳ for each row*

This list comprehension notation is particularly useful for initializing lists, for example, for initializing matrices, when we are doing matrix like computations. Supposing, I want to initialize a 4 by 3 matrix to all zeros. So, 4 by 3 matrix has 4 rows and 3 columns, and I am using the convention that, I store it row-wise. So, I have to store the first row. So, it will be 3 entries for the first row; then, 3 entries for the second row, and so on.

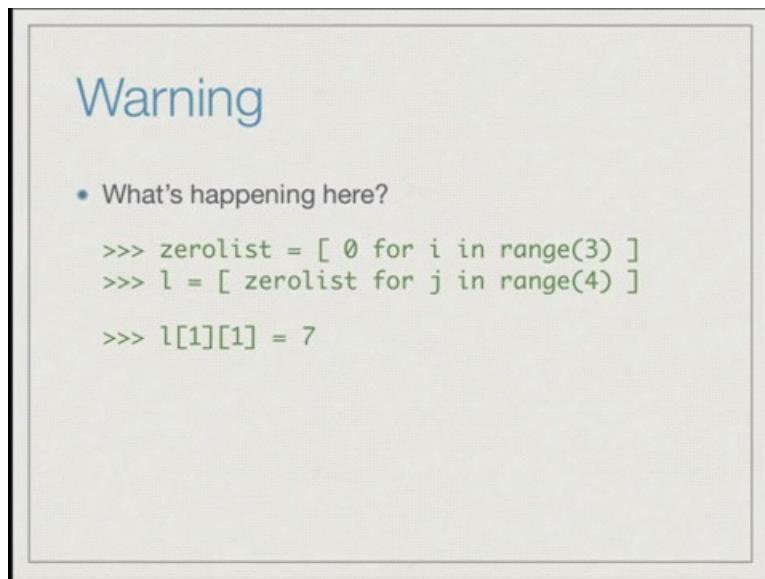
Here is an initialization, which says, l consists of something for the outer things says for, this is for each row. It is something for each row. For 4 rows 0, 1, 2, 3, I do something; and what is that something? I create a list of zeros of that size 3. Each row j, from 0 to 3, consists of columns 0, 1, 2, which are zeros. This will actually generate the correct sequence that we saw at, that would we need to initialize the generators.

(Refer Slide Time: 15:18)

```
>>> l = [ [0 for i in range(3) ] for j in range(4) ]
>>> l
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Here is that list comprehension notation for initializing the matrix. So, it says, for every  $j$  in range 4, right, we create a list, and that list itself has zero for  $i$  in range 3, and if you do this, and look at it, then, correctly it has 3 zeros, and 3 zeros, and 3 zeros, 4 times. These are the four rows.

(Refer Slide Time: 15:37)



Suppose, instead, we split this initialization into 2 steps; we first create a list of 3 zeroes called zerolist, which says zero for i in range 3. This creates a list of 3 zeros; and then, we copy this list 4 times, in the four rows. We say that the actual matrix l has 4 copies of zerolist. Now, we go and change one entry; say, we change entry 1 in row 1. From the top, it is actually second row. It is the second row, second column, if you want to think in normal terms. So, we take up list 1, which is the second list. Now, what you expect is the output of this.

(Refer Slide Time: 16:22)

```
>>> zerolist = [ 0 for i in range(3) ]
>>> l = [ zerolist for j in range(4) ]
>>> l
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> l[1][1] = 7
>>> l
[[0, 7, 0], [0, 7, 0], [0, 7, 0], [0, 7, 0]]
>>> 
```

There we have the zero lists, and then, we say, l is 4 copies of `zerolist`, for j in range 4. So, superficially, l looks exactly the same. Now, we say l 1 1 is equal to 7, and if you look at l now, we will find that we have not one 7, but 4 copies of 7. This is apparently something that we did not expect.

(Refer Slide Time: 16:56)

## Warning

- What's happening here?

```
>>> zerolist = [ 0 for i in range(3) ]
>>> l = [ zerolist for j in range(4) ]
>>> l[1][1] = 7
>>> l
[[0,7,0],[0,7,0],[0,7,0],[0,7,0]]
```
- Each row in `l` points to **same** list `zerolist`

The output after `l[1][1] = 7` is – `[0, 7, 0], [0, 7, 0], [0, 7, 0], [0, 7, 0]` and why is this happening; well, that is because by making a single `zerolist`, and then making 4 copies of it, we have effectively created 4 names with the same list. So, whether we access it through `l[0]` or `l[1]` or `l[2]` or `l[3]`, all 4 of them are pointing to the same `zerolist`. So, any one of those updates would actually update all 4 lists.

If you want to create a 2 dimensional matrix and initialize it, make sure you initialize it in one shot using a nested range, and not in 2 copies like this, because these 2 copies will unintentionally combine 2 rows into copies of the same thing, and updates to `one` row will also update another `row`.

(Refer Slide Time: 17:45)

## Summary

- `map` and `filter` are useful functions to manipulate lists
- List comprehension provides a useful notation for combining `map` and `filter`

To summarize, `map` and `filter` are very useful functions to manipulating lists, and python provides, like many other programming languages, based on the function programming, the notation called list comprehension, to combine `map` and `filter`. And, one of the uses that we saw for list comprehension is to correctly initialize 2 dimensional or multi dimensional lists to some default values.

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

**Module - 05**  
**Lecture - 01**  
**Expecting Handling**

(Refer Slide Time: 00:03)

## When things go wrong

- \*  $y = x/z$ , but  $z$  has value  $0$
- \*  $y = \text{int}(s)$ , but string  $s$  is not a valid integer
- \*  $y = 5*x$ , but  $x$  does not have a value
- \*  $y = l[i]$ , but  $i$  is not a valid index for list  $l$
- \* Try to read from a file, but the file does not exist
- \* Try to write to a file, but the disk is full

Let us see what to do when things go wrong with our programs. Now there are many different kinds of things that can go wrong. For instance we might have an expression like  $x$  divided by  $z$ , and  $z$  has a value zero. So, this expression value cannot be computed, or we might be trying to convert something from a string to an integer where the string  $s$  is not a valid representation of an integer.

We could also be trying to compute an expression, using a name whose value has not been defined, or we could try to index a position in a list which does not exist. As we go forward we will be looking at how to read and write from files on the disc. So, we may be trying to read from a file, but perhaps there is no such file or we may be trying to write to a file, but the disc is actually full. So, there are many situations in which while our program is running we might encounter an error.

(Refer Slide Time: 00:59)

## When things go wrong ...

- Some errors can be anticipated
- Others are unexpected
- Predictable error — **exception**
  - Normal situation vs exceptional situation
- Contingency plan — **exception handling**

Some of these errors can be anticipated whereas, others are unexpected. If we can anticipate an error we would prefer to think of it not as an error, but as an exception. So, think of the word exceptional. We encounter a normal situation, the way we would like our program to run and then occasionally we might encounter an exceptional situation, where something wrong happens and what we would like to do is provide a plan, on how to deal with this exceptional situation and this is called exception handling.

(Refer Slide Time: 01:36)

## Exception handling

- If something goes wrong, provide “corrective action”
- File not found — display a message and ask user to retype filename
- List index out of bounds — provide diagnostic information to help debug error
- Need mechanism to internally trap exceptions
- An untapped exception will abort the program

So, exception handling may ask, when something goes wrong how do we provide corrective action. Now the type of corrective action could depend on what type of error it is. If for instance we are trying to read a file and the file does not exist perhaps we had asked the user to type a file name. So, you could display a message and ask the user to retype the file name, saying the file asked for does not exist.

On the other hand if a list is being indexed out of bounds there is probably an error in our program, and we might want to print out this value the value of the index to try and diagnose what is going wrong with our program. Sometimes the error handling might just be debugging our error prone program. For all this what we require is a way of capturing these errors within the program as it is running without killing the program. So, as we have seen when we have spotted errors while we have been using the interpreter if an error does happen and we do not trap it in this way then the program will actually abort and exit. So, we want a way to catch the error and deal with it without aborting the program.

(Refer Slide Time: 02:53)

## Types of errors

- \* Python notifies you of different types of errors
- \* Most common error, invalid Python code  
`SyntaxError: invalid syntax`
- \* Not much you can do with this!
- \* We are interested in errors that occur when code is being executed

Now, there are many different types of errors and some of these we have seen, but we may not have noticed the subtlety of these for example, when we run python and we type something which is wrong, then we get something called a syntax error and the message that python gives us is syntax error invalid syntax.

(Refer Slide Time: 03:19)

```
>>> k = [ 5; 2]
      File "<stdin>", line 1
        k = [ 5; 2]
              ^
SyntaxError: invalid syntax
>>> 
```

For example, supposing we try to create a list and by mistake we use a semicolon instead of a comma. Then immediately python points to that semicolon and says this is a syntax error **it is** invalid python syntax.

Of course, if we have invalid syntax; that means, the program is not going to run at all and there is not much we can do. So, what we are really interested in is errors that happen in valid programs. The program is syntactically correct it is something that the python interpreter can execute, but while the code is being executed some error happens.

(Refer Slide Time: 04:01)

## Types of errors

Some errors while code is executing (run-time errors)

- Name used before value is defined  
**NameError: name 'x' is not defined**
- Division by zero in arithmetic expression  
**ZeroDivisionError: division by zero**
- Invalid list index  
**IndexError: list assignment index out of range**

These are what are called run time errors these are errors that happen while the program is running and here again we have seen these errors and they come with some diagnostic information. For instance, if we use a name whose value is undefined then we get a message from python that the name is not defined and we also get a code at the beginning of the line saying this is a name error.

This is python's way of telling us what type of error it is similarly, if we have an arithmetic expression where we end up dividing by a value 0 then, we will get something called a zero division error and finally, if you try to index a list outside its range then we get something called an index error.

(Refer Slide Time: 04:54)

```
>>> y = 5 * x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> y = 5/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> l = [1,2]
>>> l[3] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> 
```

Let us look at all this error. Just be sure that we understand. Supposing we say y is equal to 5 times x and we have not define anything for x then, it gives us an index error a name error and it says clearly that, the name x is not defined.

On the other hand, if we say is equal to 5 divided by 0 then we get a 0 division error and along with the message division by 0 and finally, if we have a list say 1, 2 and then we ask for the position three then it will say that there is no position three in this list. So, this is an index error. So, these are three examples of the types of error that the python interpreter tells us and notice that there is an error name at the beginning index error name error zero division error plus a diagnostic explanation after that.

(Refer Slide Time: 05:45)

## Terminology

- \* Raise an exception
  - \* Run time error → signal **error type**, with **diagnostic information**  
**NameError: name 'x' is not defined**
- \* Handle an exception
  - \* Anticipate and take corrective action based on error type
- \* Unhandled exception aborts execution

Let us first quickly settle on some terminology. So, usually the act of **signalling an error** is called **raising an exception**. So, when the **python interpreter** detects an error it gives us information about this error and as we saw it comes in two parts, there is the type of the error give what kind of error it is. So, it is name error or an index error or a zero division error and. Secondly, there is some diagnostic information telling us where this error occurs. So, it is not enough to just tell us oh some value was not defined it tells us specifically the name x is not defined.

This gives us some hint as to where the error might be now when, such an error is signaled by python what we would like to do is from within our program handle it right. So, we would like to anticipate and take corrective action based on the error type. So, we may not want to take the same type of action for every error type. That is why it is important to know whether it is a name error or an index error or something else.

And depending on what the error is, we might take appropriate action for that type of error and finally, if we do get an error or an exception which we **have** not explicitly handled then the python interpreter has no option, but to abort the program. So, if we do not handle an exception if an exception is unhandled then aborts the execution aborts.

(Refer Slide Time: 07:15)

## Handling exceptions

```
try:  
    . . . ← Code where error may occur  
except IndexError:  
    . . . ← What to do if IndexError occurs  
except (NameError, KeyError):  
    . . . ← Common code to handle multiple errors  
except:  
    . . . ← Catch all other exceptions  
else:  
    . . . ← Execute if try terminates normally, no errors
```

This is done using a new type of block which we have not seen before called try. So, what we have is try block. So, when we have code, here in which we anticipate that there may be some error we put it inside a try. This is our usual block of code where we anticipate that something may go wrong and now we provide contingencies for all the things that could go wrong depending on the type of error and this is provided using this except statement. It says try this and if something goes wrong, then go to the appropriate except one after the other.

The first one says what happens if an index error occurs. So, this is the code that happens if an index error occurs on the other hand maybe I could get a name error or a key error for both of which I do the same thing, so this is the next except block. So, you could have as many except blocks as you have types of errors which you anticipate errors for it is not obligatory to handle every kind of error, only those which you anticipate and of course, now you might want to do something in general for all errors that you do not anticipate. So, you can have a pure except block.

So, kind of a naked except block in which you do not specify the type of error and by default such an except statement would catch all other exceptions. The important thing to remember is that this happens in sequence. If I have three errors for example, if I have an index error and a name error and a zero division error then, what will happen - is that, it will first go here and find that there is an index error. This code will execute. The name

error code will not execute on the other hand, if I had only a name error and if I **had a zero** division error for example, then because there is a name error first it will first come here and will find that there is no index error then will come here and say there is a name error and will execute this code.

The zero division error will not be explicitly handled, the program will not abort, but there will be no code executed for the zero division error; it is not that it tries each one of these in turn it will try whichever except matches the error and it will skip the rest. So, finally, if I had only a zero division error in this particular example then, since it is not an index error and it is not a name error, it would try to go through these in turns that would come here **find** this is not a type of error. It is not a type of error and it will go to the default except statement and catch all other exceptions.

Finally, python **offers** us a very useful alternative **clause** called else. So, this else is in the same spirit as the else associated with a 'for' or a 'while' remember that a **for** or a while that does not break that terminates normally then executes the else if there is a break the else is **skipped** in the same way, if the try executes normally that is there is no error which is found then the else will execute otherwise the else is **skipped** right.

So, we have an else block which will execute if the try terminates normally with no errors. This is the overall structure of how **we** handle exceptions we put the code that we want inside a try block then we have a sequence of except blocks which catch different types of exceptions we can catch more than one type of exception **by** putting a sequence in a tuple of exceptions, we can have a default except with no name associated with it to catch all un other exceptions which are not handled and finally, we have an else which will execute if the try terminates normally.

(Refer Slide Time: 10:58)

## “Positive” use of exceptions

- \* Add a new entry to this dictionary  
`scores = {'Dhawan':[3,22], 'Kohli':[200,3]}`
- \* Batsman `b` already exists, append to list  
`scores[b].append(s)`
- \* New batsman, create fresh entry  
`scores[b] = [s]`

Now, while we normally use exception handling to deal with errors which we do not anticipate. We can actually use it to change our style of programming. So, let us look at a typical example. We saw recently that we can use dictionaries `in` python. So, dictionaries associate values with keys here we have two keys Dhawan and Kohli and with each key which is a name we have a list of scores. So, this score is a dictionary whose keys are strings and whose values are lists of numbers. Now suppose we want to add a score to this.

The score is associated with a particular batsman `b`. So, we have a score `s` for a batsman `b` and we want to update this dictionary. Now there are two situations one is that we already have an entry for `b` in the dictionary in which case we want to append `s` to the existing list `scores` of `b` the other situation is that this is a new batsman, there is no key for `b` in which case we have to create a key by setting `scores[b]` equal to the list containing `s` right. We have two alternative modes of operation it is an error to try an append to a non `existent` key, but if `there is an` existing key we do not want to lose it by reassigning `s`. So, we want to append it. So, we want to distinguish these two cases.

(Refer Slide Time: 12:21)

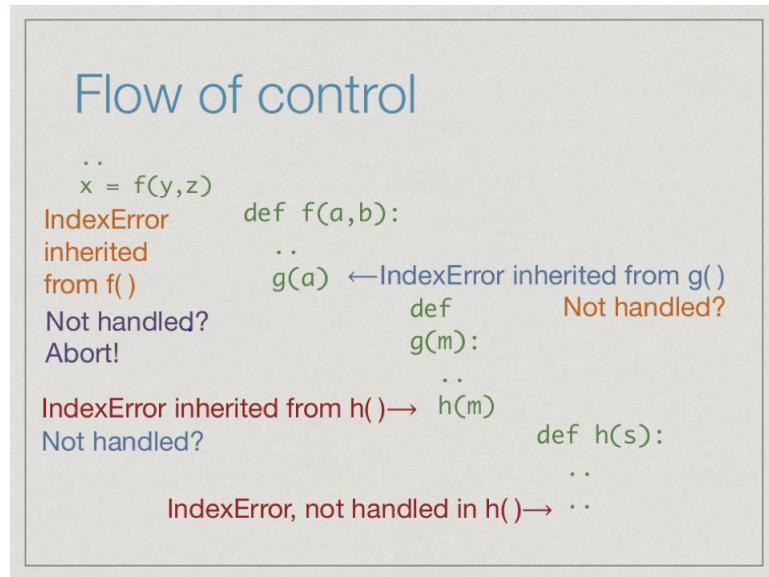
## “Positive” use of exceptions

<ul style="list-style-type: none"><li>* Traditional approach</li></ul> <pre>if b in scores.keys():     scores[b].append(s) else:     scores[b] = [s]</pre>	<ul style="list-style-type: none"><li>* Using exceptions</li></ul> <pre>try:     scores[b].append(s) except KeyError:     scores[b] = [s]</pre>
--	---

A standard way to do this in using what we have already seen, is to use the command the the statement `in` to check whether the value `b` already occurs as a key in `scores`. So, we say if `b` is in the `scores`, dot `keys` if we have `b` as an existing key then we append the score otherwise we create a new entry. So, this is fine, now we can actually do this using exception handling as follows; we try to append it right we assume by default that the `b` batsman `b` already exists as a key in this dictionary `scores` and we try `scores[b].append(s)`. What would happen if `b` is not there? Well python will signal an error saying that this is an invalid key and that is called a key error.

So, we can then revert to this `except` statement then say oh if there is key error when I try to append `s` to `scores[b]` then create an entry `scores[b] = [s]`. So, this is just a different style, it is not saying that one is better than the other, but it is just emphasizing that once we have exception handling under our control we may be able to do things differently from what we are used to and sometimes these may be more clear it is a matter of style you might be further left or the right, but both are valid pieces of python code.

(Refer Slide Time: 13:48)



Let us examine what actually happens when we hit an error. So, suppose we start executing something and we have a function call to a function `f`, with parameters `y` and `z` this will go and look up a definition for the function and inside the definition perhaps. So, this call results in executing this code and this definition might have yet another function called in it call `g`. This will in turn transfer us to a new definition sorry this should be on the same line `g` and this might in turn have another function `h` and finally, when we go to `h` perhaps this where the problem happens.

Somewhere inside `h` perhaps there is an index error and where we used this list for example, in `h` we did not put it on a try block and so, the error is not handled. So, what happens we said is when an error is not handled the program aborts, but the program **does** not directly abort; this function will abort and it will transfer the error back to whatever called it. So, what will happen here is that this index error will go back to the point where, `h` was invoked in `g`.

Now, it is as though `g` has generated an index error calling `h` has generated an index error. So, an index error is actually now within `g` because `h` did not do anything with that error we just passed it back with the error. Now, we have two options either `g` has a try block, but if `g` does not have a try block then this error will cause `g` to abort.

So, what will happen next is that if `g` does not handle it then this will go back to **where `g` was called in `f`** and likewise if now `f` does not handle it then it will go back to **where `f`**

was called in the main program. So, we keep back going back across the sequence of function calls passing back the error.

If we do not handle it in the function where we are right now, the error goes back this function aborts it goes back and finally, when it reaches the main thread of control the first function of the first python code, that we are executing there if we do not handle it then **definitely** the overall python program aborts. So, it is not as **though** the very first time we find an error which is not handled it **will** abort it will merely pass control back to where it was called from and across the sequence of calls hierarchically we can catch it at any point. So, we do not have to catch the error at the point where its handled we can catch it higher up from the point that is calling us.

(Refer Slide Time: 16:23)

## Summary

- \* Exception handling allows us to gracefully deal with run time errors
- \* Can check type of error and take appropriate action based on type
- \* Can change coding style to exploit exception handling
- \* When dealing with files and input/output, exception handling becomes very important

To summarize exception handling allows us to gracefully deal with run time errors. So, python when it flags an error tells us the type of error and some diagnostic information. Using a try and except block, we can check the type of error and take appropriate action based on the type. We also saw with that inserting a value into a dictionary example that we can exploit exception handling to develop new styles of programming and finally, what we will see is that, as we go ahead and we start dealing with input output and **files exceptions** will be rather more common as we saw earlier one of the examples we mentioned was a file is not found or a **disk is full**.

Input and output inherently involves a lot of interaction with outside things outside the program and hence is much more **prone** to errors and therefore, is useful to be able have this mechanism within our bag of tricks.

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

**Week - 05**  
**Lecture - 02**  
**Standard Input and Output**

Till now, all the programs that you have been asked to write in your assignments have actually been just functions.

(Refer Slide Time: 00:02)

### Interacting with the user

- Program needs to interact with the user
  - Receive input
  - Display output
- Standard input and output
  - Input from keyboard
  - Output to screen

These are functions, which are called from other pieces of python code and return values to them. Now, when you have a stand-alone python program, it must interact with the user in order to derive inputs and produce outputs. Let us see, how python interacts with its environment. The most basic way of interacting with the environment is to take input from the keyboard and display output to the screen.

(Refer Slide Time: 00:54)

## Reading from the keyboard

- Read a line of input and assign to `userdata`  
`userdata = input()`
- Display a message prompting the user  
`userdata = input("Enter a number")`
- Add space, newline to make message readable  
`userdata = input("Enter a number: ")`  
`userdata = input("Enter a number:\n")`

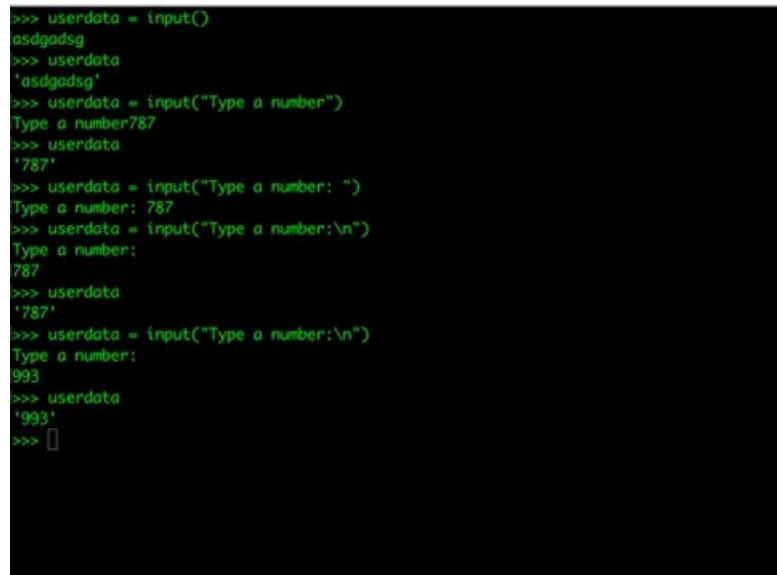
Traditionally, these modes are called standard input and standard output. So, standard input just means, take the input from the keyboard or any standard input device like that and standard output just means display the output directly on the screen. The basic command in python to read from the keyboard is `input`. If we invoke the function `input` with no arguments and assign it to a name, then, the name `user data` will get the value that is typed in by the user at the `input` command.

Remember that, it reads a line of input. The way that the user signals that the input is over, is by hitting the return button on the keyboard and the entire sequence of characters up to the return, but not including the return, is transmitted as a string to user data. Now, of course, if the program is just waiting for you for input, it can be very confusing. So, you might want to provide a prompt, which is a message to the user, telling the user, what is expected. So, you can provide such a thing by adding a string as an argument to the `input`. If you put an argument to `input` like this, then, it is a string which is displayed when the user is supposed to input data.

Now, this string is displayed as it is. So, you can make appropriate adaptations to make it little more user-friendly. We will see an example in a minute, but you might want to leave a space or you might want to insert a new line. Basically, you use the `input` command to read one line of input from the user and you can display a message to tell the user what is expected of him. So, here is what happens if I just say `userdata` is equal

to input(), the python program will just wait and now, as a user who does not know what is expected, we do not know whether it is processing something or it is waiting for input.

(Refer Slide Time: 02:30)



```
>>> userdata = input()
asdgadsg
>>> userdata
'asdgdsg'
>>> userdata = input("Type a number")
Type a number787
>>> userdata
'787'
>>> userdata = input("Type a number: ")
Type a number: 787
>>> userdata = input("Type a number:\n")
Type a number:
787
>>> userdata
'787'
>>> userdata = input("Type a number:\n")
Type a number:
993
>>> userdata
'993'
>>> []
```

Now, it so turns out that, if we type something and press enter, it will come back. Now, if I ask for the contents of the name userdata, it will be impact me the string **of things** that I had typed. So, providing an input prompt without a message can be confusing for the user. So, what we said was, we might want to say something like, provide an input like this. Now, it provides us with a message, but the number that we type for instance, is stuck to the message. It is not very readable. So, userdata is indeed not a number, now, it is a string as we will see in a minute.

But the fact is that, we did not get any space or anything else. It looks a bit ugly. So, what we said is that you can actually, for instance, put a colon and a space so that the message comes like this. Now, this is a slightly nicer prompt and you could also pfirefut a new line if you want, which is **signalled** by this special character, backslash n. Now, the message comes and then, you type on a new line and in all cases, the outcome is the same; userdata, the name to which you are reading the input, becomes set to the string that is typed in by the user. If I do it again and if I type in something else like 993, for example, then userdata becomes the string “993”. So, you can use input with a message and make the message as readable as you can.

(Refer Slide Time: 04:16)

## Reading from the keyboard

- Use exception handling to deal with errors

```
while(True):
    try:
        userdata = input("Enter a number: ")
        usernum = int(userdata)
    except ValueError:
        print("Not a number. Try again")
    else:
        break
```

As we saw, when we were playing with the interpreter, the input that is read by the function input is always a string. Even if you say, enter a number and the user types in a number, this is not actually a number. If you want to use it as a number, you have to use this type conversion. Remember, we have **these** functions int, str and so on. So, we have to use the int function to type convert whatever the user has typed, to an integer. Now, of course, remember that, if the user types some garbage, then you get an error, right. If the user does not type some, something valid, then you will get an error. So, what we can do is, we can use exception handling to deal with this error.

So, what we can say is, try userdata. This is the code that we had before: those 2 lines. So, what we want to say is, we will try these lines, but if the user types something which is not a number, then, we are going to ask him to type it again and it will turn out that, that type of error in python is called a value error.

(Refer Slide Time: 05:20)

```
>>> int('ssdfs')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'ssdfs'
>>> 
```

So, you can verify this by going to the python interpreter and checking. In the python interpreter, if we try to apply **int** to some nonsensical things, then we get a value error.

(Refer Slide Time: 05:32)

## Reading from the keyboard

- Use exception handling to deal with errors

```
while(True):
    try:
        userdata = input("Enter a number: ")
        usernum = int(userdata)
    except ValueError:
        print("Not a number. Try again")
    else:
        break
```

So, what we are trying to say is that, if we get a value error, we want to take appropriate action. So, we have this try block and if we see a value error, what we do is, we print a message. Now, we are going to see print just after this, but we print a message to the user, saying this is not a number, try again and this is now, the whole thing is enclosed inside a while loop and this while loop has a condition **True**.

In other words, the condition is never going to become false; this while loop is going to keep on asking for a number. So, how do we get out of this? Well, if I come here and I get a value error, it will go back and the while will execute again. but if there is no error, remember, if there is no error here, then it will come to the else. This else is executed if there is no error and what the else does is, it gets us out of this vicious cycle.

In other words, we are in an infinite loop, where we keep on trying to get one more piece of data from the user, until we get something that we like and when we get that, we break out of the loop. This is another kind of idiomatic way to use exceptions, in the context of input and output. As we said in the last lecture, input and output is inherently error prone, because, you are dealing with an uncertain, interacting environment, which can do things, which you cannot anticipate or control. So, you must take appropriate action to make sure that the interaction goes in the direction that you expect.

The other part of interaction is displaying messages, which we call standard output or printing to the screen.

(Refer Slide Time: 06:59)

## Printing to screen

- Print values of names, separated by spaces  
`print(x,y)  
print(a,b,c)`
- Print a message  
`print("Not a number. Try again")`
- Intersperse message with values of names  
`print("Values are x:", x, "y:", y)`

And, this is achieved using the print statement, which we have seen occasionally, informally, without formally having defined it. The basic form of the print statement is to give a sequence of values, separated by commas. So, print x, y will display the value of x, then, a space, then, the value of y. ‘print a, b, c’ will display 3 values; the values of a,

b and c, separated by spaces. Now, the other thing that we can do is, directly print a string or a message.

Like we saw in the previous example, we can say, print the string “Not a number. Try again”. This will display this string on the screen. Now, you can combine these two things in interesting ways. So, print takes, in general, a sequence of things of arbitrary links, separated by commas. These things could be either messages or names. So, we can say things, supposing, we want to print the value of x and y, but we want to indicate to the output, which is x, which is y.

Instead of saying, just print x comma y, which produces 2 values on the screen, with no indication as to which is x and which is y, we could have this more elaborate print statement, which prints 4 things. The first thing it prints is a message saying, the values are x colon. This will print x colon; it will leave a space; then, it will print the value of, current value of x; then, it will print y colon after a space and then, it will print the current value of y.

(Refer Slide Time: 08:39)

## Fine tuning `print()`

- Items are separated by space by default

```
(x,y) = (7,10)
print("x is",x,"and y is",y,".")
x is 7 and y is 10 .
```
- Specify separator with argument `sep="..."`

```
print("x is ",x," and y is ",y,".", sep="")
x is 7 and y is 10.
```

So, we can intersperse messages with values, with names, to produce meaningful output that is more readable. By default, print appends a new line whenever it is executed. In other words, every print statement, we just print the way we have done so far, appears on a new line because the previous print statement implicitly moves the output to a new line. Now, if we want to control this, we can use an optional argument called end. So, we

can provide a string saying, this is what should we put at the end of the print statement; by default, the value here is this new line character. So, we can replace this by something else.

Here is an example. Supposing, we write these 3 statements. The first statement says, ‘print “Continue on the”’; this is just a string; but set end to a space. The second line says, ‘print “same line”’ and then, it says, set end to a full stop and a new line. And then, the third statement says, ‘print “Next line.”’. So, what we are doing is, in the first 2 statements, we are changing the default.

The default would have been to print a new line, but the first statement is not printing a new line. If we print this, what we see is that, the first 2 statements continue on the same line, come on a single line, because, we have disabled the default print new line and we have explicitly put a new line here and this has forced the next one to come on the next line. If we break this up and see what happens here, we see that, in the first statement, we had this end, which says insert a space and this is why we get a space between the word ‘the’ on the first line and the word ‘same’ coming in the second line.

Otherwise, ‘the’ and ‘same’ would have been fused together as a single word, right. So, end equal to space is effectively separating this print from the next print by a space. The next print statement inserts a full stop and a new line. Implicitly, although the word same line ends without a full stop, we produce a full stop and after we produce a full stop, it produces a new line and finally, after this new line, the next line comes in the new line and of course, because here we did not say anything; if we print after that, we implicitly would print on a new line.

The other thing that we might want to control is how the items are separated on a line. We said that, if we say print x comma y, then x and y will be separated by a space by default, right. If we do this, print x, y, we set x equal to 7, y equal to 10 and we say x is x and y is y and then, we want to end with a full stop. This is what you want; you want to write a string, ‘x is’, then the value of x and ‘y is’, then the value of y and then, a full stop. Now, because everything is separated by a space, what we find is that, we find a space over here; do you see this? This is fine. So, we get a space here, because, that is from this comma; we get a space here, which is from this comma; we get a space here, which is from this comma.

And then, we get an unwanted space between 10 and the full stop. So, how do we get rid of this space, the second space, right? We do not want a full stop to come after the space. So, just like we have the optional argument end, we have an optional argument sep, which specifies what string should be used **to separate**. So, for example, if we take the earlier thing, we can say, do not separate it with anything.

Now, of course, do not separate it with anything, it changes, because, then, this x is 7 will get fused and this and this will get fused. So, what we do instead is, we put the space explicitly here. Earlier, we had no space here, at the end, just around the quotes. Now, we put spaces where we want them and we say do not put any other spaces. So, what this will say is that, x is space, I give this space; do not put the space, put the value of x; do not put a space, now, I give a space. So and y is, give a space and then, now do not put a space here. These commas do not contribute any space, because I have set separator should be empty and in particular, what this means is that, this last comma, the comma between the y and the full stop, will not generate a space.

And in fact, if you execute this, then, you will get the output, x is 7 and y is 10 and the way it works is that, this is the first block. This is everything up to here. Then, this is the second block, this is this. Then, this is the third block, which is this whole thing, with the spaces given and then, this is the value of y and finally, this is the full stop. This is one way to control the output of a print statement.

(Refer Slide Time: 13:28)

## Formatting print

- May need more control over printing
- Specify width to align text
- Align text within width — left, right, centre
- How many digits before/after decimal point?
- See how to do this later

So, with the optional arguments end and sep, we can control when successive prints continue on the same line and we can control to some limited extent, how these values are separated on a line. But we may actually want to do a lot more. We might want to put a sequence of things, so that, they all line up, right.

Supposing, we want to print out a table using a print statement, we want to make sure that the columns line up. So, we might want to say that, each item that we want to print, like we have printing a sequence of numbers, line by line, because, the numbers may have different widths; some may be 3 digits, some may be 5 digits; we might say print them all to occupy 7 characters width, right. This is a thing that we might want to do, align text.

Now, within this alignment, we might want to align things left or right. If we have a default with, say 10 characters; if they are numbers, we might want them right aligned, so that the units digit is aligned up; if they are names, we might want them left aligned, so that we can read them from left to right, without it looking ragged. And if we are doing things like calculating averages or something, we may not want the entire thing to be displayed; we might want to truncate it to 2 decimal points; say, it is currency or something like that. These are all more intricate ways of formatting the output and we will see in the next lecture how to do this. But right now, you can use end and sep to do minimal formatting.

(Refer Slide Time: 14:53)

**Summary**

*print n,y*

- Read from keyboard using `input()`
  - Can also display a message
- Print to screen using `print()`
  - Caveat: In Python 2, () is optional for `print`
  - Can control format of `print()` output
    - Optional arguments `end="..."`, `sep="..."`
    - More precise control later

To summarize, you can use the input statement with an optional message, in order to read from the keyboard. You can print to the screen using the print statement. Now, we mentioned at the beginning that, there are some differences between python 2 and 3 and here is one of the more obvious differences that you will see, if you look at python 2 code, which is available from various sources.

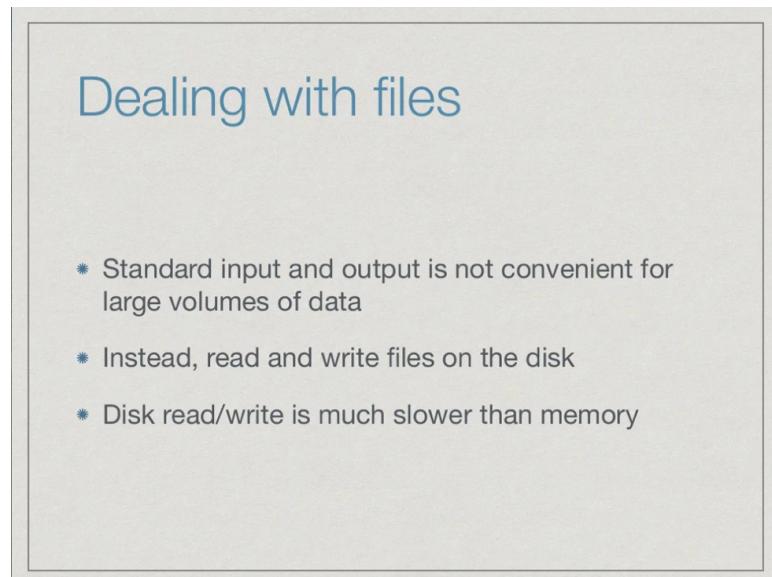
In python 2, you can say print space and then, give the, what is given as the arguments to print in python 3. Python 3 insists on it being called like a function, with brackets; in python 2 the brackets are optional. So, you will very often see, in python 2 code, something that looks like print x, y, given without any brackets. This is legal in python 2; this is not legal in python 3. Just be careful about this.

And what we saw is that, with the limited amount of control, we can make print behave as we want. So, we can specify what to put at the end. In particular, we can tell it not to put a new line. So, continue printing in the same line and we can separate the values by something other than the default space character.

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

**Week - 05**  
**Lecture - 03**  
**Handling files**

(Refer Slide Time: 00:02)



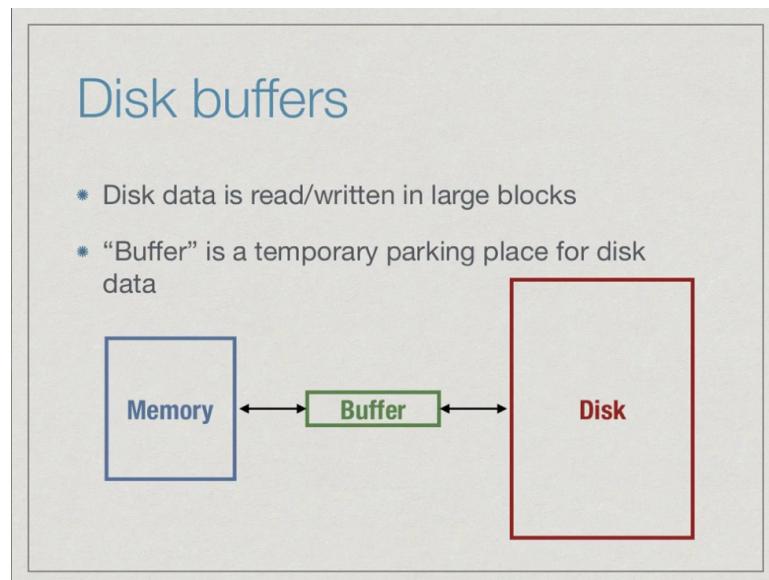
**Dealing with files**

- \* Standard input and output is not convenient for large volumes of data
- \* Instead, read and write files on the disk
- \* Disk read/write is much slower than memory

In the last lecture we saw how to use the input and print statements to collect input from the standard input that is the keyboard, and to display values on the screen using print.

Now, this is useful for small quantities of data, but we want to read and write large quantities of data. It is impractical to type them by hand or to see them as a scroll pass in this screen. So, for large data we are forced to deal with files which reside on the disk. So, we have to read a large volume of data which is already written on a file in the disk and the output we compute is typically return back into another file on the disk. Now, one thing to keep in mind when dealing with disks is that disk read and write is very much slower than memory read and write.

(Refer Slide Time: 00:48)



To get around this most systems will read and write data in large blocks. Imagine that you have a large storage facility in which you store things in big cartons. Now, when you go and fetch something you bring a carton at a time even if you are only looking for, say one book in that carton, you do not go and fetch one book out of the carton from the storage facility, **you** bring the whole carton and then when you want put things back again you assemble them in a carton and put them back.

In the same way, the way that data flows back and forth between memory and disk is in chunks called blocks. So, even if you want to read only one value or only one line it will actually a fetch large volume of data from the disk and store it **in** what is called a buffer and then you read whatever you need from the buffer. Similarly, when you want to write to the disk you assemble your data in the buffer when the buffer is enough quantity to be written on the disk then one chunk of data or **block is** written back on the disk.

(Refer Slide Time: 01:49)

## Reading/writing disk data

- \* Open a file — create **file handle** to file on disk
  - \* Like setting up a buffer for the file
- \* Read and write operations are to file handle
- \* Close a file
  - \* Write out buffer to disk (**flush**)
  - \* Disconnect file handle

When we read and write from a disk the first thing we need to do is connect to this buffer. This is called opening a file. So, when we open a file we create something called a file handle and you can imagine that this is like getting access to a buffer from which data from that file can read into memory or **written** back.

Now, having opened this file handle everything we do with the file is actually done with respect to this file handle. So, we do not directly try to read and write from the disk, instead we read and write from the buffer that we have opened using this file handle and finally, when we are done with our processing we need to make sure that all the data that we have written goes back. So, this is done by closing the file.

So, closing the file has two effects; the first effect is to make sure that all changes that we intended to make to the file. Any data we want to write to the file is actually taken out to the buffer and put on to the disk and this technically called flushing the buffer. So, closing a file flushes the output buffer make sure that all rights go back to the file and do not get lost and the second thing it does is that it in some sense makes this buffer go away. So, it disconnects the file handle that we just set up. Now, this file is no longer connected to us if we want to read or write it again we have to again open it.

(Refer Slide Time: 03:11)

## Opening a file

```
fh = open("gcd.py", "r")
```

- \* First argument to `open` is file name
  - \* Can give a full path
- \* Second argument is mode for opening file
  - \* Read, "`r`": opens a file for reading only
  - \* Write, "`w  - Append, "a": append to an existing file`

The command to open a file is just ‘open’. The first argument that you give open is the actual file name on your disk. Now, this will depend a little bit on what system you are using, but usually it has a first part and `an` extension. This commands, for instance, to open the file gcd dot py. Now implicitly, if you just give a file name it will look for it in the current folder or directory where you running the script. So, you can give a file name which belongs to the different part of your directory hierarchy by giving a path and how you describe the path will depend on whether you are working on Windows or Unix, what operating system `you` are using.

Now, you see there is a second argument there, which is letter ‘r’. This tells us how we want to open the file. So, you can imagine that if you are making changes to a file by both reading it and writing it, this can create confusion. So, what we have to do is decide in advance whether we are going to read from a file or write to it, we cannot do both. There is no way we can simultaneously read from a file and modify it while it is open. So, read is signified by ‘r’.

Now, write comes in two flavors, we might want to create a new file from scratch. In this case we use a letter ‘w’. So, ‘w’ stands for write out a new file, we have to be bit careful about this because if we write out a file which already exists then opening it with more ‘w’ will just overwrite the contents that we already had. The other thing which might be useful to do is to take a file that already exists and add something to it. This is called

append. So, we have two writing modes; ‘w’ for write and ‘a’ for append. What append will do is it will take a file which already exists and add the new stuff the writing at the end of the file.

(Refer Slide Time: 05:02)

## Read through file handle

```
contents = fh.read()  
* Reads entire file into name as a single string  
  
contents = fh.readline()  
* Reads one line into name—lines end with '\n'  
* String includes the '\n', unlike input()  
  
contents = fh.readlines()  
* Reads entire file as list of strings  
* Each string is one line, ending with '\n'
```

Once we have a file open, let us see how to read. So, we invoke the read command through the file handle. This is like some of the other function that you saw with strings and so on, where we attach the function to the object. So, fh is the file handle we opened, we want to read from it. So, we say fh dot read, what fh dot read does is it **swallows** the entire contents the file as a single string and returns it and then we can assign it to any name, here we use the name contents. So, **contents** is now assigned the entire data which is in the file handle pointed by fh in one string.

Now, we can also consume a file, we are typically dealing with text files. So, text file usually consists of lines; think of python code, for example, we have lines after lines after lines. A natural unit is a bunch of texts which is ended with new line character. If you remember this is what the input command does, the input command waits for you type something and then you press return which is a new line and whatever you type up to the return is then transmitted by input as a string to the name that you assigned to the input.

So, readline is like that, but the difference between the readline and input is that, when you read a line you get the last new line character along with the input string. When you

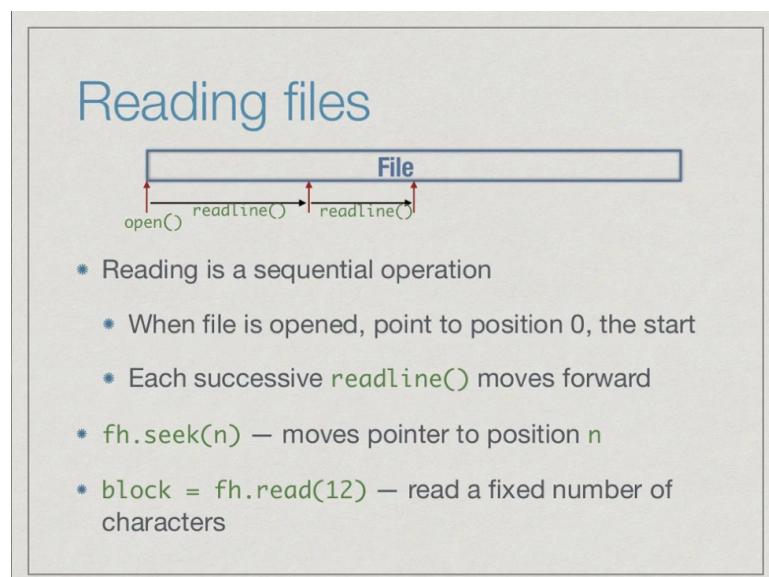
say input you only get the characters which come before the last new line the new line is not included, but in readline you do get the new line character.

So, you have to remember that you have the extra character floating around at the end of your string. So, this is conventionally the noted backslash n. The backslash n is a notation which denotes a single character even though looks two characters. This is supposed to be the new line character. Now, the actual new line character differs on operating systems from one to the other, but in python if we use backslash n and it will correctly translated in all the systems that you are using.

The third way that you can read from a file is to read all the lines one by one into a list of strings. So, instead of readline, if I say readlines then it reads the entire the files as a list of strings. Each string is one item in the list and remember again each of these lines has the backslash n included. So, read, readline and readlines, none of them will actually remove the backslash n. They will remain faithfully as part of your input.

In other words, if you are going to transfer this from one file to another, you do not want to worry reinserting the backslash n because this is already there. So, you can use this input output directly, but on the other hand, if you want to do some manipulation of the string then you must remember this backslash n is there and you must deal with it appropriately.

(Refer Slide Time: 07:49)



Reading files is inherently a sequential operation. Now, of course, if we use the basic command read, it reads the entire content. So obviously, it reads from beginning to the end, but if you are reading one line at a time then the way it works is that when we open the file we are initially at the beginning of the file. So, you can imagine a pointer like this red arrow which tells us where we are going to read next. So, initially when we open we are going to read from the beginning, now each readline takes us forward. If I do a readline at this point it will take me up to the next backslash n.

Remember a line is a quantity which is delimited by backslash n. So, we could have a line which has 100 characters, next line could have 3 characters and so on. It is from one backslash n to the next is what a line, so this is not a fixed link. So, we will move forward reading one character at a time until we have backslash n, then everything up to the backslash n will be returned as the effect to a string return by the readline and pointer move to the next character. Now, we do another readline possibly of different line again the point to move forward. So, in this way we go from beginning to the end.

In case we want to actually divert from the strategy there is a command seek, which takes a position, an integer n, and moves directly to the position n regardless of where you are. This is one way to move back or to jump around in a file other than by reading consecutively line by line.

Finally, we can modify the read statement to not to read the entire file, but to read a fix number of characters. Now, this may be useful if your character actually your file actually consists of fix blocks of data. So, you might have say, for example, pan numbers which are typically 10 characters long and you might have just stored them as one long sequence of text without any new lines knowing that every pan number is 10 characters. So, if we say fh dot read 10, it will read the next pan number and keep going and this will save you some space in the long run. So, there are situation where you might exploit this where you read a fix number of characters.

(Refer Slide Time: 09:56)

## End of file

- \* When reading incrementally, important to know when file has ended
- \* The following both signal end of file
  - \* `fh.read()` returns empty string ""
  - \* `fh.readline()` returns empty string ""

When we are reading a file incrementally, it is useful to know when the file is over because we may not know in advance how long files is or how many lines of file is. So, if you are reading a file line by line then we may want to know when the file has ended. So, there are two situations where we will know this. So, one is if we try to read using the read command and we get nothing back, we get an empty string that means the file is over, we have reached end of file.

Similarly, if we try to read a line and we get empty string it means we reached the end of file. So, read or readline if they return empty string its means that we have reached the end of the file. Remember, we are going sequential from beginning to the end. So, we reached the end of the file and there is nothing further to read in this file.

(Refer Slide Time: 10:44)

## Writing to a file

```
fh.write(s)
* Write string s to file
  * Returns number of characters written
  * Include '\n' explicitly to go to a new line
fh.writelines(l)
* Write a list of lines l to file
  * Must include '\n' explicitly for each string
```

Having read from a file then the other thing that we would like to do is to write to a file. So, here is how you write to a file just like you have read a command you have a write command, but now unlike read which implicitly takes something from the file and gives to you, here you have to provide it something to put in the file. So, write takes an argument which is a string. When you say, write s says take the string s and write it to a file.

Now, there are two things; one is this s may or may not have a backslash n, it may have more than one backslash n. So, is nothing tells you this is one line part of a line more than a line you have to write s according to the way you want it to be written on the file, if you want it to be in one line you should make sure it ends with a backslash n.

And this write actually returns the number of characters written. Now, this may seem like a strange thing to do, why should it tell you because you know from the length of s what is number of character is written, but this is useful if, for instance, the disk is full. If you try to write a long string and find out only part of the string was written and this is a indication that there was a problem with the write. So, it is useful sometimes to know how many characters actually got written out of the characters that tried to write.

The other thing which writes in bulk to a file is called writelines. So, this takes list of strings and writes them one by one into the file. Now though it says write lines these may not actually be lines. So, its bit misleading the name if you want to them in lines you

must make sure that you have each of them terminated by backslash n. If they are not then they will just cascade to form a long line thing. So, though it says writelines it should be more like write a list of strings, that should, that is a more appropriate name for this function, it just takes a list of strings and writes it to the file pointed to by the file handle.

(Refer Slide Time: 12:40)

## Closing a file

`fh.close()`

- \* Flushes output buffer and decouples file handle
  - \* All pending writes copied to disk

`fh.flush()`

- \* Manually forces write to disk

And finally, as we said once we are done with a file, we have to close it and make sure the buffers that are associated with the file, especially if you are writing to a file that they are flushed. So, fh dot close, will close the file handle fh and all pending writes at this point are copied out to the disk. It also now means that fh is no longer associated with the file we are dealing with. So, after this if we try to invoke operation on fh it is like having undefined name in python.

Now, sometimes there are situations where we might want to flush the buffer without closing the file. We might just want to make sure that all writes up to this point have been actually reflected on the disk. So, there is a command flush which does this. In case we say flush, it just say if there are any pending writes then please put them all on to the disk, do not wait for the risk drives to accumulate until the buffer is full and then write as you normally would to the disk.

(Refer Slide Time: 13:42)

## Processing file line by line

```
contents = fh.readlines()  
for l in contents:  
    ...  
* Even better  
for l in fh.readlines():  
    ...
```

Here is a typical thing that you would like to do in python, which is to process it line by line. The natural way to do this is to read the lines into a list and then process the list using for. So, you say content is fh dot readlines and then for each line and contents you do something with it. You can actually do this in a more compact way, you can get rid away with the name contents and just read directly every line return by the function fh dot readlines. So, this is the equivalent formulation of the same loop.

(Refer Slide Time: 14:25)

## Copying a file

```
infile = open("input.txt", "r")  
outfile = open("output.txt", "w")  
for line in infile.readlines():  
    outfile.write(line)  
infile.close()  
outfile.close()
```

As an example, for how to use this line by line processing, let us imagine that we want to copy the contents of a file input dot txt to a file output dot txt.

So, the first thing we **need** to do is to make sure that we open it correctly. We should actually open outfile with mode ‘w’ and in file mode ‘r’. This tells that I am going to read from infile and write to outfile. Now, for each line in returned by readlines on infile, remember that when I get line from readline the backslash n is already there, if I do not do anything to the backslash n, I can write it out the exactly the same way. So, for each line that I read from the list infile dot readlines I just write it to outfile and finally, I close **both** the files. This is one way to copy one file from input to output.

(Refer Slide Time: 15:25)

## Copying a file

```
infile = open("input.txt", "r")
outfile = open("output.txt", "w")
contents = infile.readlines()
outfile.writelines(contents)] replace for
infile.close()
outfile.close()
```

Of course, we can do it even in one shot because there is a command called writes lines, which takes the list of strings and writes them in one shot. So, instead of going line by line through the list readlines we can take the entire list contents and just output it directly through writelines, this is an alternative way where I have replaced. This is basically replacing the for. So, instead of saying for each line in infile I can just write it directly out.

(Refer Slide Time: 16:02)

## Strip new line character

- Get rid of trailing '\n'

```
contents = fh.readlines()
for line in contents:
    s = line[:-1]
```
- Instead, use `rstrip()` to remove trailing whitespace

```
for line in contents:
    s = line.rstrip()
```
- Also `strip()` — both sides, `lstrip()` — from left
  - String manipulation functions — coming up

One of the things we are talking about is this new line character which is a bit of annoyance. If we want to get with a new line character, remember this is only a string and the new line character is going to be a last character in this string. So, one way to get is just to take slice of the string up to, but not including the last character. Now, remember that we when we count backwards minus 1 is the last character. If we will take the slice from 0 up to minus 1 then it will correctly exclude the last character from the string.

So, `s` is equal to `line colon minus 1`, will take the line and the strip of the last character which is typically backslash n that we get, when we do `readlines`. Now, in general we may have other spaces. So, remember when you write out text very often we cannot see the spaces the end of the line because they are invisible to us. These are what are called white space. So spaces, tabs, new lines, these are characters which do not display on the screen, especially spaces and tabs and there at a end of line, we do not know the line ends with the last character we see there are spaces after words.

So, `r strip` is a string command which actually takes a string and removes the trailing white space, all the white spaces are at end of the line. In particular there is only a backslash n and it will strip to a backslash n. It also strips to other jump there is some spaces and tabs before the backslash n and return that. So, `s` equal to `line dot r strip` that is strip line from the right of white space. This is an equivalent thing to the previous line

**except** it is more general because strips all the white space not just by the last backslash n, but all the white spaces end of the line.

We can also strip from the left using l strip or we can strip on both sides if we just say strip without any characterization l or r. These are the string manipulation functions and we will look at some more of them, but this is just useful one which has come up immediately in the context of file handling. So, before we go ahead let us try and look at some examples of all these things that we have seen so far.

(Refer Slide Time: 18:05)

```
madhavan@dolphinair:...016-jul/week5/python/files$ ls
input.txt
madhavan@dolphinair:...016-jul/week5/python/files$ more input.txt
The quick brown
fox
jumps over the lazy
dog.
madhavan@dolphinair:...016-jul/week5/python/files$ 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.
>>> f = open("input.txt","r")
>>> for line in f.readlines():
...     print(line)
...
The quick brown
fox
jumps over the lazy
dog.

>>> for line in f.readlines():
...     print(line)
...
>>> █
```

We have created a file called input dot txt which consist of a line, the quick brown fox jumps over the lazy dog. Now, let us open the python interpreter and try to read lines from this file and print it out. So, we can say, for instance, that f is equal to open input dot txt in read. Now, I have opened the file and now I can say, for instance, for line in f dot readlines print line. Now, you will see something interesting happening here, you will see that we have now a blank line between every line our file.

Now, why is there blank lines between every line in our file that is because when we readlines we get a backslash n character from the line itself. So, the quick brown, the first line end with the backslash n, fox end with backslash n and then over and above that if you remember the print statement adds a backslash n of its own. So, actually print is putting out to blank lines for each of these.

Now, let us try and do this again, supposing I repeat this thing and now I do this again, now nothing happens the reason nothing happens is because we had this sequential reading of the file. So, the first time we did f dot readlines, it read one line at a time and now we are actually pointing to the end of the file.

(Refer Slide Time: 19:49)

```
>>> text = fh.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fh' is not defined
>>> text = f.read()
>>> text
''
>>> text = f.readline()
>>> text
''
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>> f = open('input.txt', "r")
>>> for line in f.readlines():
...     print(line,end="")
...
The quick brown
fox
jumps over the lazy
dog.
>>> █
```

If for instance, at this point we were to say text equal to fh dot read, sorry f dot read, then text will be empty string. This is the indication that we have actually reached the end of the file. Similarly, if we try to say readline again text will be empty string. So, remember we said that if read or readlines returns the empty string then we have reached the end of the file. The only way we can undo this is to start again by closing the files. So, what we say is f dot close. This closes of the file.

Now, if you try to do f dot read then we will get an error saying that this is not being defined. So, we do not have f with us anymore. So, again we have to say f is open input dot txt r and now we can say while for line in f dot readlines for each line. Supposing, we now use that trick that we had last time which is to say end equal to empty string that says do not insert anything after each print statement. Now, if you do this you see we get back to exactly the input files as it is without the extra blank lines because print is no longer creating these extra lines.

(Refer Slide Time: 21:10)

```
>>> f = open("input.txt", "r")
>>> g = open("output.txt", "w")
>>> for line in f.readlines():
...     g.write(line)
...
16
4
20
5
>>> f.close()
>>> g.close()
>>> ^D
madhavan@dolphinair:...016-jul/week5/python/files$ more output.txt
The quick brown
fox
jumps over the lazy
dog.
madhavan@dolphinair:...016-jul/week5/python/files$
```

Let us say we want to copy input dot txt to a output dot txt, we say f is equal to open input dot txt r as before and we say g is equal to open output dot txt w and now we say for line in f dot readlines, g dot write line.

Now, notice you get the sequence of numbers why do we get a sequence of numbers that is because each time we write something it returns a number of character written and it will turn out, if you look at the lines quick brown fox, etcetera, for example, the second line is just fox, fox has three letters, but if you include the backslash n its wrote 4 letters. That is why quick brown was 15 letters plus a backslash n, fox was a 3 plus backslash n. So, this is line by line. Now, if I correctly close these files then come out of this, then output dot txt is exactly the same as input dot txt.

(Refer Slide Time: 22:26)

## Summary

- Interact with files through file handles
- Open a file in one of three modes — read, write, append
- Read entire file as a string, or line by line
- Write a string, or a list of strings to a file
- Close handle, flush buffer
- String operations to strip white space

To summarize what we have seen is that, if you want to interact with files we do it through file handles, which actually corresponds to the buffers that we use to interact between the memory and the file on the disk. We can open a file in one of three modes; read, write and append. We did not actually do an example with the append, but we do append what it will do, keep writing beyond where the file already existed, otherwise write will erase the file and start from the beginning.

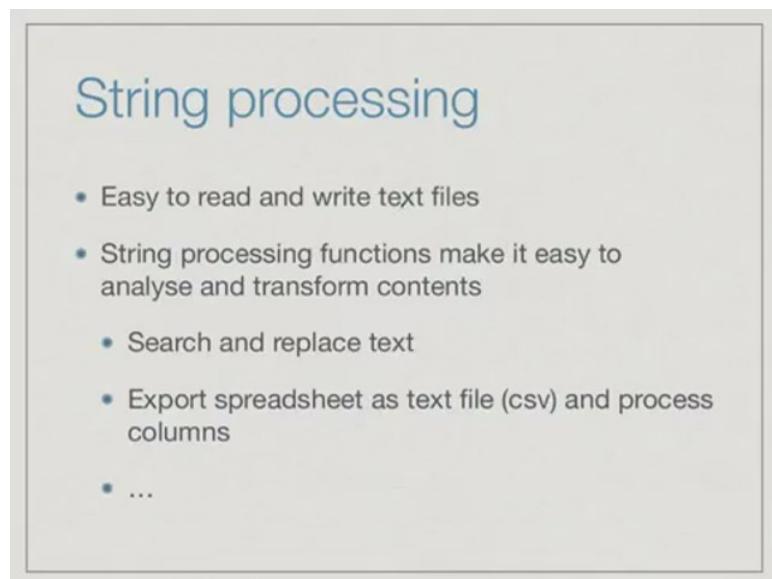
We saw that read, readline and readlines, using this we can read the entire file in one shot of the string or read it line by line. Similarly, we can either write a string or we write a list of strings too. So, we have a write command in a writelines and writelines are more correctly to be interpreters write list of strings.

Finally, we can close the handle when we have done and in between that we can flush the buffer by using flush command and we also saw that there are some string operations to strip white space and this can be useful to remove these trailing backslash n which come whenever you are processing text files.

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

**Week - 05**  
**Lecture - 04**  
**String Processing**

(Refer Slide Time: 00:02)



**String processing**

- Easy to read and write text files
- String processing functions make it easy to analyse and transform contents
  - Search and replace text
  - Export spreadsheet as text file (csv) and process columns
  - ...

The last lecture we saw how to read and write text files. And reading and writing text invariably involves processing the strings that we read and write. And so, Python has a number of string processing functions that make it easier to modify this content.

So usually, you are reading and writing files in order to do something with these files, and to do something with this you can use built in string functions which are quite powerful. Among other things, what you can do with these string functions is for example, search for text or search and replace it. A typical use of string processing for a file is when we take something like a spread sheet and export it as text. There is something called a comma separated value format CSV, where the columns are output separated by commas as text. Now, what we can do with a string file is to read such a file line by line and in each line extract the columns from the text by reading between the commas. So, we will see all this in this lecture.

(Refer Slide Time: 01:04)

## Strip whitespace

- `s.rstrip()` removes trailing whitespace  
`for line in contents:  
 s = line.rstrip()`
- `s.lstrip()` removes leading whitespace
- `s.strip()` removes leading and trailing whitespace

The first example of a string command that we already saw last time is the commands to strip white space right. We have `rstrip`, which we used for example to remove the trailing whitespace backslash n in our lines, and we had `lstrip` to remove leading whitespace, and we had `strip` which removes it on both directions. Let us see how this works.

(Refer Slide Time: 01:30)

```
>>> s = "      hello      "
>>> s
'      hello      '
>>> t = s.rstrip()
>>> t
'      hello'
>>> t = s.lstrip()
>>> t
'hello      '
>>> t = s.strip()
>>> t
'hello'
>>> []
```

Let us create a string which has whitespace before and afterwards, so let us put some spaces may be a tab and then the word hello and then two tabs. We have a string which has whitespace and you can see the tabs are indicated by backslash t and blanks. Now, if

we want to just strip from the right we say t is equal to s dot rstrip. Remember this strip command strings are immutable right it won't change s it will just return a new string, if I say t is s dot r strip it will strip to the whitespace to the right and give me t, if I look at t it has everything up to hello but not that tab and the space afterwards.

Similarly, if I say t is s dot l strip it will remove the ones to the left now t will start with hello, but it will have the whitespace at the end. Finally, if I say t is s dot strip then both sides are gone and I will just get the word that I want. This is useful because when you ask people to type things and forms for example, usually if they leave some blanks before and after, so if you want everything before the first blank to be lost and the last blank only keep the text in between then you can use the combination of lstrip, rstrip or just strip to extract the actual data that you want from the file.

(Refer Slide Time: 02:43)

## Searching for text

- s.find(pattern)
  - Returns first position in s where pattern occurs, -1 if no occurrence of pattern
- s.find(pattern,start,end)
  - Search for pattern in slice s[start:end]
- s.index(pattern), s.index(pattern,l,r)
  - Like find, but raise ValueError if pattern not found

The next thing that may we want to do is to look text in a string. There is a basic command called find. So, if s a string and pattern is another string that I am looking for in s, s dot find pattern will return the first position in s which pattern occurs. And if pattern does not occur, so the positions will there obviously be between 0 and the length of s minus 1. We already wrote some our own implementation of this earlier. So, if it does not occur it will give you minus 1.

Sometimes you may not want to search entire string, so pattern takes an optional pair of argument start and end in which case instead of looking for the pattern from the entire

string it looks at a slice from start to end with the usual convention that this is the position from start to end minus 1. There is another version of this command called index. And the difference between find and index is what happens when the pattern is not found. In find when the pattern is not found you get a minus 1, in index when the pattern is not found you get a special type of error in this case a value error. So again let us just see how these things actually work.

(Refer Slide Time: 03:54)

```
>>> s = "brown fox grey dog brown fox"
>>> s.find("brown")
0
>>> s.find("brown",5,len(s))
19
>>> s.find("cat")
-1
>>> s.index("cat")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> []
```

So, we have a string here `s` which contains the word "brown fox grey dog brown fox." Now if I ask it to look for the first occurrence of the word "brown", then it will return the position 0 because it is right there at the beginning of string. If on the other hand I do not want this position, but I wanted to say starting from position 5 and going to length of `s` for example, then it will say 19 and if you count you will find that the second occurrence of brown is at position 19.

If on the other hand I look for something which is not there like "cat" then find will return minus 1, so minus 1 is not the error but the indication that the string was not found. The difference with index is that if I give index the same thing instead of a minus 1 it gives me a value error saying the substring does not occur right this is how find and index work.

(Refer Slide Time: 04:53)

## Search and replace

`s.replace(fromstr,tostr)`

- Returns copy of s with each occurrence of `fromstr` replaced by `tostr`

`s.replace(fromstr,tostr,n)`

- Replace at most first n copies
- Note that s itself is unchanged — strings are immutable

The next natural thing after searching is searching and replacing. If I want to replace something I give it two strings what I am searching for and what I am replacing it with and it will return a copy of s with each occurrence of the first string replaced by the second string. Now this can be controlled in the following ways; supposing, I do not want to each occurrence, but I only want say the first occurrence or the first three occurrences.

So, I can give it **an** optional argument saying how many such occurrences starting from the beginning should be replaced. It says replace at most the first n copies and notice that like and strip and all that, here it's because changing this string replacing something by something else, is not that s is going to change because strings are immutable is going to return us the transform string. So let us look at an example.

(Refer Slide Time: 05:45)

```
>>> s = "brown fox grey dog brown fox"
>>> s.replace("brown","black")
'black fox grey dog black fox'
>>> s.replace("brown","black",1)
'black fox grey dog brown fox'
>>> t = "ababa"
>>> t.replace("aba","DD")
'DDba'
>>> t = "abaaba"
>>> t.replace("aba","DD")
'DDDO'
>>> 
```

Once again let us see our old example; `s` is "brown fox grey dog brown fox" and now supposing I want to replace "brown" by "black", then I get 'black fox grey dog black fox.' If I say only want 1 to be replaced then I get 'black fox grey dog' where the second brown is left unchanged. Now you may ask what happens if I have this pattern it does not neatly split up if I have the different copies of brown overlaps. Supposing, I have some stupid string like "abaaba" and now I say replace all "aba" by say "DD".

Now the question is, will it find two aba's or 1 aba, because there is an aba starting at position 0, there is also an aba in the second half of the string starting at position 2. The question is will it mark both of these and replace them by DD, well, it does not because it does it sequentially so it first takes the first aba, replaces it by DD, at this point the second aba has been destroyed. So it will not find it.

Whereas, if I had for instance two copies of this disjoint then it would have correctly found this and given me DD followed by DD. So, there is no problem about overlapping strings it just does it from right to left and it makes sure that the overlap string is first written, so it will not the second copy will not get transformed.

(Refer Slide Time: 07:17)

## Splitting a string

- Export spreadsheet as “comma separated value” text file
- Want to extract columns from a line of text
- Split the line into chunks between commas
  - columns = s.split(",")
  - Can split using any separator string
- Split into at most n chunks
  - columns = s.split(" : ", n)

The next thing that we want to look at is splitting a string. Now when we take a spreadsheet and write it out as text, usually what happens is that we will have an output which looks like this. The first column would be written followed by comma then second column, so if we had three columns then the first column set 6 second column set 7 and the third was string hello, then we write it out a text as you will get 6, 7 and "hello". Actually "hello" is a bit of problem because it has double quotes let us not use hello let us use something simpler. So let us just say that we had three numbers 6, 7 and 8 for example.

Now, what we want to do is we want to extract this information. So, we want to extract the individual 6, 7 and 8 that we had as three values. So what we need to do is look for this text between the strings, so we want to split the column into lines into chunks between the commas and this is done using the split command. So, split takes a string s and takes a character or actually could be any string and it splits the columns it gives you a list of values that come by taking the parts between the commas. So, up to the first comma is a first thing. So columns is just a name that we have used, it could be any list. The first item of the list will be up to the first comma then between the first and second comma and so on and finally after the last comma.

Comma in this case is not a very special thing you can split using any separator string. And again just like in replace we could control how many times we replaced, here we

can also control how many splits you make. So, you can say split according to this string notice that this could be any string so here we are splitting it according to space colon space. But we are saying do not make more than n chunks, if we have more than n columns or whatever chunks which come like this beyond a certain point we will just lump it as one remaining string and keep it with us. So again let us see how this works.

(Refer Slide Time: 9:22)

```
>>> csvline = "6,7,8"
>>> csvline.split(",")
['6', '7', '8']
>>> csvline.split(",",1)
['6', '7,8']
>>> csvline = "6#7#8"
>>> csvline.split("#")
['6', '7', '8']
>>> 
```

Suppose this is our line of text which I will call CSV line it is a sequence of values separated by commas notice it is a string. Now if I say CSV line dot split using comma as a separator and then I get a list of values the string 6, the string 7, the string 8. Remember this is exactly like what we said about input it does not give you the values in the form that you want you then have to convert them using int or these are still strings. So, it just takes a long string and splits it into a list of smaller strings. Now here there are three elements so if I say for example I only want position 0, 1, 2. So, if I say I only wanted to do it once then I get the first 6, but then 7 and 8 does not get split because it only splits once.

Now, if I change this to something more fancy like say hash comp question mark. So now I have a different separator it's not a single character, but hash question mark then I can say split according to hash question mark and this will give me the same thing. You can split according to any string it's just a uniform string. There are more fancy things you can do with regular expression and all that, but we won't be covering that for now.

As long as you have a fixed string which separates your thing you can split according to that fixed string.

(Refer Slide Time: 10:49)

## Joining strings

- Recombine a list of strings using a separator

```
columns = s.split(",")
joinstring = ","
csvline = joinstring.join(columns)

date = "16"
month = "08"
year = "2016"
today = "-".join([date,month,year])
```

So, the inverse operation of split would be to join strings. Supposing, we have a collection of strings and I want to combine it in to a single string separate each of them by a given separator. So as an example, supposing we take s which is some CSV output and we split it into columns on comma, and then we can take join string and set it to the value comma and then use that to join the columns.

Now this is a bit confusing, so join is a function which is associated with a string. In this case a string in concerned is a comma. So it says, more or less you are saying comma dot join columns which is use comma to join columns. So, you have just given it a name here join string is equal to comma and then CSV line is join string dot join columns.

So what this says is, use comma to separate so if at the end of this I had got like last time 6, 7 and 8, then this will now put them back as 6 comma 7 comma 8 into a single string. Here is another example, here we have a date 16 a month 08 and a year 2016 given as strings and I want to string it together into a date like we normally use with hyphens.

Here instead of giving an intermediate name to the hyphens and then saying hyphens dot join I directly use this string itself, just want to illustrate that you can directly use this joining string itself as a constant string and say use this to join this list of values. All you

need to make sure is what you have inside the join in the argument is a list of strings and what you applied to is the string which will be used to join them. Let us just check that this works the way we actually intended to do.

(Refer Slide Time: 12:38)

```
>>> date = "16"
>>> month = "08"
>>> year = "2016"
>>> "-".join([date,month,year])
'16-08-2016'
>>> 
```

Let us directly do the second example. Supposing, we say date is 16, remember these are all strings month is 08, year is 2016, and now I want to say what is the effect of joining these three things using dash as separator and I get 16 dash 08 dash 2016.

(Refer Slide Time: 13:09)

## Converting case

- Convert lower case to upper case, ...
- `s.capitalize()` — return new string with first letter uppercase, rest lower
- `s.lower()` — convert all uppercase to lowercase
- `s.upper()` — convert all lowercase to uppercase
- `s.title()`, `s.swapcase()`, ...

So there are many other interesting things you can do with strings for example, you can manipulate upper case and lower case. If you say capitalize, what it will do is it will convert the first letter to upper case and keep the rest as lower case, if you say s dot lower it will convert all upper case to lower case, if you say s dot upper it will convert all lower case to upper case and so on.

There are other fancy things like s dot title. So, title will capitalize each word. This is how it normally appears say in the title of a book or a movie. S dot swap case will invert lower case to upper case and upper case to lower case and so on. So there are whole collection of functions in the string thing which deal with upper case, lower case and how to transform between these.

(Refer Slide Time: 13:52)

## Resizing strings

- s.center(n)
  - Returns string of length n with s centred, rest blank
- s.center(n, "\*")
  - Fill the rest with \* instead of blanks
- s.ljust(n), s.ljust(n, "\*"), s.rjust(n), ...
  - Similar, but left/right justify s in returned string

The other thing that you can do with strings is to resize them to fit what you want. So if you want to have a string which is positioned as a column of certain width then we can say that center it in a block of size n. So what this will do is it will return a new string which is of length n with s centered in it.

Now by centering what we mean is that on either side there will be blanks instead of blanks you can put anything you want like, stars or minuses. You can give a character which will be used to fill up the empty space on either side rather than a blank. Now you may not want it centered or you may not want to the left or the right, so you can for example left justify during ljust or rjustify during rjust and again you can give an

optional character and so on. So, we can just check one or two of these just to see how they work.

(Refer Slide Time: 14:46)

```
>>> s = 'hello'
>>> s.center(50)
'          hello          '
>>> s.center(50,'-')
'-----hello-----'
>>> s.ljust(50,'-')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'ljust'
>>> s.ljust(50,'-')
'hello-----'
>>> s.rjust(50,'-')
'-----hello'
>>> 
```

Suppose, we take a short string like 'hello' and now we center it in a large block of say 50. We say s dot center 50, then this gives us hello with a lot of blank spaces on either side. Now we can replace those blank spaces by anything we want, so say minus sign then we will get a string of a minus signs or hyphens before that. Now we can also say that I want the thing left justified in this not a center. So if I do that then I will get hello at the beginning and a bunch of minus signs, similarly with rjust and so on.

(Refer Slide Time: 15:23)

## Other functions

- Check the nature of characters in a string  
`s.isalpha()`, `s.isnumeric()`, ...
- Many other functions
- Check the Python documentation

Some of the other types of functions which we find associated to strings are to check properties of strings. Does s consists only of the letters a to z and capital a to capital z. So that is what s dot is alpha says is it an alphabetic string, if it is true it means it is, if it is not it has at least one non alphabetic character. Similarly is it entirely digits, is numeric will tell us if it is entirely digits. So, there is a huge number of string functions and there is no point going through all of them in this thing, we will if we need them as we go along we will use them and explain them.

But you can look at the Python documentation look under string functions and you will find a whole host of useful utilities which allow you to easily manipulate strings. And this is one of the reasons that Python is a popular language because you can do this kind of easy text processing. So you can use it to quickly transform data from one format to another and to you know change the way it looks or to resize it and so on. String functions are an extremely important part of Python's utility as a glue language for transforming things from one format to another.

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

**Week - 05**  
**Lecture - 05**  
**Formatting printed output**

(Refer Slide Time: 00:01)

## Formatted printing

- Recall that we have limited control over how `print()` displays output
  - Optional argument `end="..."` changes default new line at the end of print
  - Optional argument `sep="..."` changes default separator between items

When we looked at input and print earlier in this week's lectures, we said that we could control in a limited way how print displayed the output. Now by default print takes a list of things to print, separates them by spaces and puts a new line at the end.

However, it takes an optional argument `end` equal to string which changes what we put at the end, in particular if we put an empty string it means that it does not start a new line, so the next print will continue on the same line. Similarly we can change the separator from a space to whatever we want and in particular if we do not want any spaces we can put them ourselves and just say the separator is nothing - the empty string.

(Refer Slide Time: 00:43)

## String format() method

- By example

```
>>> "First: {0}, second: {1}".format(47,11)
'First: 47, second: 11'

>>> "Second: {1}, first: {0}".format(47,11)
'Second: 11, first: 47'
```

- Replace arguments by position in message string

Now, sometimes you want to be a little bit more precise, so for this we can use the format method which is actually part of the string library. So the set of things that we can do with strings, last, in the previous lecture we looked at other things we can do string like, find, replace and all this things, so this is like that, it is in the same class.

Remember when you are doing print, you are actually printing a string. So, anything you can do to modify a string will give you another string that is what you are going to print. So, the string here is actually going to call a format method. So, the easiest way to do this is by example. We have a base string here, which is first, second and we have these two funny things in braces.

The funny things in braces are to be thought of as the equivalent of arguments in the function, these are things to be replaced by actual values and then what happens is that when you give this string and you apply the format method then the 0 refers to the first argument and 1 refers to the second argument. So what we are doing is, we are replacing by position, so if I actually take this string and I pass it to python the resulting thing is first colon 47, second colon 11, because the first argument, the brace 0 is replaced by the first argument to format 47 and the second replaces the second.

Now the positions determine the names so they do not have to be used in the same order. So, we could first print argument 1 and then print argument 0 as the second example

**shows.** Essentially, this version of format allows us to pass things into a string by their position in the format thing. So we are replacing arguments **by** position.

(Refer Slide Time: 02:34)

## format() method ...

- Can also replace arguments by name

```
>>> "One: {f}, two: {s}".format(f=47,s=11)
'One: 47, two: 11'  
X
>>> "One: {f}, two: {s}".format(s=11,f=47)
'One: 47, two: 11'
```

Now we can do the same thing by name. This is exactly like defining a function where **remember,** we said that **we** could give function arguments **and we** could pass it by name. So, in same way here we can specify names of the arguments **to format**, we can say f is equal to 47, s is equal to 11, that is first and second.

Now we can say one f and two s. Now here the advantage is not by position but by name. If I take these two things and I exchange them, so if I make **f the** second argument and s the first argument, **and I** pass it to the same string then f will be correctly caught as 47 and s as 11, so here by using the name not the position. So, the order in which you supply the things to format does not matter.

(Refer Slide Time: 03:20)

## Now, real formatting

```
>>> "Value: -{0:3d}".format(4)
```

- 3d describes how to display the value 4
- d is a code specifies that 4 should be treated as an integer value
- 3 is the width of the area to show 4

```
'Value: -4'
```

So, up to this point we have **not** done **any** **formatting**. All we have done is we have taken a string and we have told us how to replace values for place holders in **the** string. There is no real formatting **which has** happened because whatever we did **with** that we could **have** already done using the existing print statement that we saw.

Now the real formatting comes by giving additional instructions on how to display each of these place holders. So here we have 0 followed by some funny thing, we have **this** special colon and what comes after the colon is the formatting instruction. This has two parts here, we see a 3 and a d and they mean different things. So the 3d as a whole tells us how to display the value there is going to be **passed here**, that is the first thing. D is a code that specifies, I think **it** stands for decimal, so d specifies that 4 should be treated as an integer value. So, we should actually **display it as** a normal integer value namely it's a base ten integer.

Finally, 3 **says** that we must format 4 so that **it** takes three spaces. It occupies the equivalent of three spaces though it is a single digit. So if I do all **this**, then what happens is I get value, now notice that already there is one space here, then it going to take three spaces so I am going to get two blank spaces and then **a** 4, so that is why we have this long, so this **is** actually three blank spaces and then a 4. This whole thing, this part of it comes from the format. So I have a blank space, **a blank space and** a 4, because I was

told to put 4 in a width of 3 and think of it as a number, so since its number it goes to right hand.

(Refer Slide Time: 05:06)

## Now, real formatting

```
>>> "Value:{0:6.2f}".format(47.52)
• 6.2f describes how to display the value 47.523
• f is a code specifies that 47 should be treated as a
  floating point value
• 6 — width of the area to show 47.523
• 2 — number of digits to show after decimal point
"Value:47.52"
```

Let us look at another example. Supposing, I had number which is not an integer, but a floating point number, so it is 47.523. Now here first thing is that we have instead of d we have f for floating point. So, f, 6.2f, this whole thing to the right of the colon tells me how to format the values comes from here. 6.2f breaks up as follows; the f, the letter part of it always tells me what the type of value is. So, it says that 47.523 should be treated as a floating point value. And the second thing is that it says this 6 tells me how much space I have to write whatever I have to write. So it says the total value including the decimal point, everything is going to be 6 characters wide.

Finally, the 2 says how many digits to show after the decimal point. If I apply all this then first of all because I have only two digits after decimal point this 3 gets knocked off, and then because it's set to use 6 character, now if I count from the right, this is 1, 2, 3, 4, 5 characters but it's set to use 6 characters, that is why there is an extra blank here. If you notice here, there is only one blank, but here there are two blanks. The second blank comes from the format statement.

(Refer Slide Time: 06:27)

## Real formatting

- Codes for other types of values
  - String, octal number, hexadecimal ...
- Other positioning information
  - Left justify
  - Add leading zeroes
- Derived from `printf()` of C, see Python documentation for details

Unfortunately this is not exactly user friendly or something that you can easily remember, but there are codes for other values. We saw f and d, so there are **codes** like s for string, and o for octal, and x for hexadecimal. All these values can be displayed also using these formatted things. And you can also do other things you can tell it not to put it on the right put **it** on the left, so you **can left** justify the value. In **a field** of width 5 for example, if you want to put a string you might say the string should come from the left, not from the right. Then you can add leading zeroes, so you might to display a number 4 not in width 3 not as 4, but as 004. So, all these things you can do.

As I said this is **a** whole zoo of formatting things that you can do with this. These all have **their** origin from the language C and the statement called printf in C, so the exact format statements in our 0, 3d and 6.2f and all what they **mean**. It is best that you look up python documentation, **you** may not need all variations of it, the **ones that** you need you **can** look up when you need them, but it is useful to know that this kind of formatting can be done.

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

**Week - 05**  
**Lecture - 06**  
**Pass, del ( ) and None**

(Refer Slide Time: 00:01)

**Doing nothing**

- Blocks such as `except:, else:, ...` cannot be empty
- Use `pass` for a null statement

```
while(True):
    try:
        userdata = input("Enter a number: ")
        usernum = int(userdata)
    except ValueError:
        pass
    else:
        break
```

For the last lecture this week we look at some useful things which will crop up and which do not fit anywhere else specific so we just combined a couple of them into this one single presentation.

So, we had an **example** when we are doing the input and print about how to prompt the user in case they had an invalid entry. We were trying to read a number and what we said was that we would input some string that the user provides, so give them a message saying enter the number they provide us with the string and then we try to convert it using the `int` function. And this `int` function will fail if the user has provided a string which is not a valid integer, in which case we will get a value error. And we get a value error we print out a message saying try again and we go back.

Finally, if we succeed, that is this try succeeds the `int` works then we will exit from this try block go to the else and the else will break out to of the loops. So, this we had already seen. Now the question is, what if we want to change this so that we do nothing we do

not want to do this. In other words if the user does not present a valid value instead of telling them why we just keep saying enter a number I mean we have seen this any number of times right, you go to some user interface and you type something wrong it does not tell you what is wrong it just keeps going back and back and back and asking to type again. So, how would you actually program something as unfriendly as that?

What we want to say is, if I come to this value error do nothing. Now the Problem with python is that wherever you put this kind of a colon it expects something after that, you cannot have an empty block. So if I put a colon there must be at least one statement after that. This is a syntactic rule of Python you cannot have an empty block. But here I want to do nothing, I want to recognize there is a value error and then go back here that is fine, but I do not want to do anything else.

How do I do nothing in Python? So the answer is, that there is a special statement called pass. Pass is a statement which exists only to fill up spaces which need to be filled up. So when you have blocks like except or else, if you put the block name there you cannot leave it empty. In this case you can use the word pass in order to do nothing.

(Refer Slide Time: 02:28)

Removing a list entry

- Want to remove `l[4]`?
- `del(l[4])`
- Automatically contracts the list and shifts elements in `l[5:]` left

Supposing, I have a list and I want to remove an element from the middle of the list. So one way to do this of course, is to take the slice up to that position this slice from that position then glue them together using plus and so on. **But what** if I want to directly do this. It turns out that that there is a command called del. If I say `del l[4]` and what it does is

effectively removes 1 4 from the current set of values, and this automatically contracts the list and shifts everything from position 5 on wards to the left by 5. So let us just verify with this works the way it claims to work.

(Refer Slide Time: 03:09)

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del(l[4])
>>> l
[0, 1, 2, 3, 5, 6, 7, 8, 9]
>>> []
```

Supposing, we set our list to be the range of values from 0 to 10, 0 to 9 say, and now I say del 1 4 then l becomes 0, 1, 2, 3, 5, 6, 7, 8, 9, because l 4 was the value 4 remember the position start from 0. And so it has actually deleted the value at the fourth position. This also works for dictionaries. If we want to remove the value associated with the key k then you can del d k and it will remove the key k and whatever values associated with it and removal from it. So, the key will now it considered being an undefined key.

(Refer Slide Time: 03:56)

## Undefining a value

- In general, `del(x)` removes the value associated with `x`, makes `x` undefined

```
x = 7  
del(x)  
y = x+5  
  
NameError: name 'x' is not defined
```

In general we can take a name like `x` and say `del x` and what this will do is make `x` undefined. Supposing, you wrote some junk code like this we set `x` equal to 7 and then we say `del x`, and then we ask `y` equal to `x` plus 5. Now this point since `x` has been undefined even though it had a value of 7 this expression `x` plus 5 cannot be calculated because `x` no longer has a value, and what you will end up with is error that we saw before namely a name error saying that the name `x` is not defined.

(Refer Slide Time: 04:29)

## Checking undefined name

- Assign a value to `x` only if `x` is undefined

```
try:  
    x  
except NameError:  
    x = 5
```

How would you go about checking if a name is defined, well of course you could use

exception handling. Supposing, we want to assign a value to a name x only if x is undefined, then we can say try x so this is trying to do something with the x.

Remember that names can be anything, it could be functions. So, you can just write x because if it is currently in name of a function which takes no arguments we will try to execute that function, so it is perfectly valid to just write x. But x if it has no value it will give a name error. So, you can say try x and if you happened to find a name error set it to 5 otherwise leave it untouched. If x already has a value this will do nothing to x, if x does not have a value then it will update the value of x to 5.

(Refer Slide Time: 05:15)

## The value None

- `None` is a special value used to denote “nothing”
- Use it to initialise a name and later check if it has been assigned a valid value

```
x = None          • Exactly one value None
...
if x is not None:  • x is None is same as
    y = x           x == None
```

Now, usually what happens is that we want to check whether something has been defined or not so far. It is not a good idea to just leave it undefined and then use exception handling to do it, because you might actually find **strange** things happening. So, Python provides us with a special value called `None` with the capital N, which is the special value used to define nothing - an empty value or a null value.

We will find great use for it later on when we are defining our own data structures, but right now just think of none as a special value, there is only one value in none and it denotes nothing. So, the typical use is that when we want to check whether name has a valid value we can initialize it to none and later on we can check if it is still none. So, initially we say x is equal to none and finally we go ahead and say, if x is not none then set y equal to x, Another words y equal to x is will not be executed if x is still none.

Now, notice the peculiar word is not. So, we are using is not equal to. So there is exactly one value `None` in Python in the space. All `None`s point to the same thing. Remember was checking when we say `l1` is `l2`, we are checking whether `l1` and `l2` point to the same list object. We were asking whether `x` and the constant `None` point to the same `None` object and there is only one value `None`. So, `x` is `None` as the same as `x` equal to `None`, so `x` is not `None` is the same as `x` not equal to `None`. So, `x` is not `None` is much easier to read the next not equal to `None`. That is why we will write it this way.

(Refer Slide Time: 06:58)

## Summary

- Use `pass` for an empty block
- Use `del()` to remove elements from a list or dictionary
- Use the special value `None` to check if a name has been assigned a valid value

What we have seen our three useful things which we will use from time to time as we go along. One is the statement `pass` which is the special statement that does nothing and can be used whenever you need an empty block. Then we saw the command `del` which takes the value and undefined it. The most normal way to use `del` is to remove something from a list or a dictionary, you would not normally want to just undefine name which is holding simple value.

But from a dictionary or a list we might want to remove a key or if might want to remove a position and `del` is very useful for that. Finally, we have seen that there is a special value `None` which denotes a null value, it is a unique null value and this can be used to initialize variables to check whether they have been assigned sensible values later in the code.

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

**Week - 06**  
**Lecture - 01**  
**Backtracking, N Queens**

For many problems, we have to search through a set of possibilities in order to find the solution.

(Refer Slide Time: 00:02)

## Backtracking



- Systematically search for a solution
- Build the solution one step at a time
- If we hit a dead-end
  - Undo the last step
  - Try the next option

There is no clear solution that **we** can directly reach. So, we have to systematically search for it. We keep building candidate solutions one step at a time. Now it might be that the solution that we are trying to get does not work. So, we hit a dead end, and then we undo the last step and try **the** next option. Imagine for instance if you are solving a Sudoku. So, you have a grid and then you start filling up things and there are some points you realize that there is nothing **you can put here**.

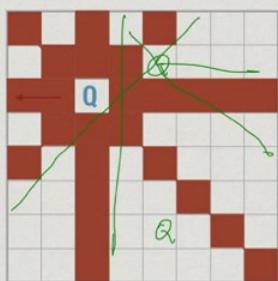
Then you go back and you have to change something you did before. So, we have to backtrack, we have to go forwards trying to solve the problem; and at some point when we realize that we are stuck we cannot solve the problem again, we have to go back and

change something we have done before and try something else.

(Refer Slide Time: 00:53)

## Eight queens

- Place 8 queens on a chess board so that none of them attack each other
- In chess, a queen can move any number of squares along a row column or diagonal



One of the classic problems of this kind is called Eight queens problem. The problem is to place 8 queens on a chess board so that none of them attack each other.

Now, if you have ever played chess, you would know that the queen is a very special piece it can move any number of squares along a row, column or diagonal - for instance, if we place the queen here, in the third row and the third column, then it could move anywhere upward down the third column anywhere left or right on the third row, and along the two diagonals on which the square three comma three lies.

Since it can move along these columns it can also capture any piece that lies along these rows. The queen is said to attack all these squares. The squares to which the queen can move are said to be attacked by the queen. So, our goal is to place queens so that, they do not attack each others, so if we have a queen here then we cannot put another queen in any of the red squares, we have to put it somewhere else. For instance we could put a new queen; say for instance here this would be ok; or here.

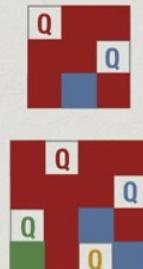
And then, if I put a queen here in turn it will attack more pieces like, it will attack these

squares and you rule out some more options so I will not be able to place queens there and so on. So, we want to see if we can place 8 queens. Now we cannot place more than 8 queens; because, there are only 8 rows if you place 9 queens, 2 will be in the same row or the same column and the same column. They will have to attack each other. So, 8 is clearly the limit, the question is whether we can actually put 8?

(Refer Slide Time: 02:28)

## N queens

- Place N queens on an  $N \times N$  chess board so that none attack each other
- $N = 2, 3$  impossible
- $N = 4$  is possible
- And all bigger N as well



So, we can generalize this question and ask not for 8, but  $N$ . Supposing, I have a chessboard in which there are  $N$  rows and  $N$  columns. Can I place  $N$  queens on such a chessboard? Now for  $N$  equal to one the question is trivial, because you only have to put 1 queen on 1 square. Now, it is easy to see that  $N$  equal to 2 is impossible because, if I have 2 squares and wherever, I put a queen say here it will attack all the remaining squares. No matter where I put the queen, every other square will be on the row, column or diagonal of that queen.

And so there is no possibility of putting a second queen. It turns out that three is also impossible. Supposing we start by putting a queen in the top left corner then we will see that it blocks out the first column, the first row and the main diagonal. This leaves two slots open for the second queen, but wherever we put, whichever of the two we put, it will block the other one.

Once we put a queen in one of those slots the other one is on the same diagonal and there is no free slot for the third queen. So, just by exhaustive analysis we can show that, **n** equal to three is actually impossible. For  $N$  equal to 4 for a 4 by 4 board, it does turn out to be possible. We should not start at the corner, but one of the corners. Supposing we put it in the second column, then we get this pattern of block squares.

Then we can find an empty slot on the second row right at the end. So, we put a queen there it blocks of certain of some more squares in the last column and in that diagonal, but this still leaves one slot in the third row, unfortunately the third queen does not block the last two slot on the fourth row and we have this kind of symmetric pattern where everything is one of the corner in which none of the queens attack each other.

Now, it turns out that once we cross  $N$  equal to 4, for 5, 6, 7, 8, you can show that there is always a solution possible. Our task is to find such a solution. How do we find a solution for  $N$  greater than or equal to 4?

(Refer Slide Time: 04:24)

## 8 queens

- Clearly, exactly one queen in each row, column
- Place queens row by row
- In each row, place a queen in the first available column
- Can't place a queen in the 8th row!

So, as you observed, the first of first thing you know is that there can be exactly one queen in each row and in each column because queens attack the column and row on which they lie. If we have two queens on the same row or the same column they will

necessarily attack each other. Since 8 is the classical size of a chessboard let us look at specifically our example for 8 queens. So, we want to place the queens now row by row. We know that there is exactly one queen in each row.

Let us first put a queen in the first row, then based on that put a queen in the second row and so on exactly as we did for the 4 by 4 case that we saw in the previous slide. So, in each row we will place the queen in the first available column, given the queens that I have already been placed so, far by available we mean a square which is not attacked so far. So, we start with an 8 by 8 board and in the first row now everything is available. By our analysis we are going to put a queen in the first available column, namely the top left once we do this; it blocks out the first row and column and the main diagonal. So, all the shaded squares are now under attack. We move to the second row and we try to put a queen in the first available column this is the third one and this in turn will attack another set of rows, columns and diagonal squares.

Now, we move to the third row and in the 5th column we can place a queen. And this one again attacks some squares. So, we have added some colors to indicate, as each new queen is placed which squares are newly under attack by the new queen, some of them are attacked by multiple queens. For instance the yellow queen attacks the blue square on the diagonal which was already attacked by the first queen.

So, we will leave it blue for now. In this way we can proceed. So, we put a 4th queen on the 4th row, and then this is a mistake this should be already attacked by this queen and then we will place a 5th queen and then a 6th one and then a 7th one and now we find that all the squares in the 8th row are actually blocked. There is no way to extend this solution to put the 8th queen. So, we have to do something about this, we cannot place a queen in the 8th row.

(Refer Slide Time: 06:36)

## 8 queens

- Can't place the a queen in the 8th row!
- Undo 7th queen, no other choice
- Undo 6th queen, no other choice
- Undo 5th queen, try next

Since we cannot put a place with queen in the 8th row we have to go back and change something we did before now. The last thing we did was to put the 7th queen right. So, we do that and we find that unfortunately for the 7th queen, we had only one choice. So, we have no other choice for the 7th queen. Though the 7th queen could not lead to a solution, it was not the choice of the 7th queen, which actually made a problem, but it was something earlier.

Then we go back and try to move the 6th queen. So, once again if you remove the 6th queen then this unblocks a few squares, but at the same time there was no other place to place the 6th queen **on** the 6th row. So, again this was a unique choice that we had made. Now if we go back to the 5th queen then we find that there is a way to place the 5th queen. In a different place namely it move it to this slot. So, we can move this 5th queen to one slot to the right and try again.

So, having gone back from the 8th square and, so 8th row which is completely blocked, to the 7th row which had **only** one choice, to the 6th row which had only one choice we come back to the 5th row and now we try the next choice for the 5th row. If we try **the** next choice **for** the 5th row - then we get this pattern of squares and now we see for example, that we cannot put a 6th thing. So, both the choices for the 5th row actually

turn out to be **bad**. So, you would now have to go back and try a different choice for the 4th row and so on.

(Refer Slide Time: 07:55)

## Backtracking

- Keep trying to extend the next solution
- If we cannot, undo previous move and try again
- Exhaustively search through all possibilities
- ... but systematically!

This is what backtracking is all about, we keep trying to extend the solution to the next step if we cannot we undo the previous move and try again, and in this way we exhaustively search through all the possible solutions, but we do it in a systematic way we do not go back and randomly reshuffle some of the choices we made before we go back precisely one step and undo the previous steps.

So, at each step we have a number of choices we go through them systematically, for each choice we try to extend the solution if the solution does not get extended we come back we try the next choice and when we exhaust all choices at this level we report back to the previous level that we have failed then they will try their next choice and so on. The key to backtracking is to do a systematic search through all the possibilities by going forwards and backwards one level at a time.

(Refer Slide Time: 08:47)

## Coding the solution

- How do we represent the board?
- $n \times n$  grid, number rows and columns from 0 to  $n-1$ 
  - $board[i][j] == 1$  indicates queen at  $(i, j)$
  - $board[i][j] == 0$  indicates no queen
- We know there is only one queen per row
- Single list  $board$  of length  $n$  with entries 0 to  $n-1$ 
  - $board[i] == j$ : queen in row  $i$ , column  $j$ , i.e.  $(i, j)$

So, how would we actually encode this kind of an approach? Specifically, for the 8 queens problem, so our first question is how to represent the board because a board is what keeps changing as we make moves and undo them. The most obvious way for an N queen solution is to represent the board literally as an N by N grid. And since python numbers list position from 0 onwards we have an N by N grid and we number the columns not 1 to N, but 0 to N minus 1, so will have rows 0 to N minus 1 and columns 0 to N minus 1. We can now put a value 1 or 0 or true or false to indicate whether or not there is a queen at the square  $i$  comma  $j$ ;  $i$  is the row,  $j$  is the column.

So, we can have a two dimensional list, board or list of lists, which has  $N$  minus 1 by  $N$  minus 1, 0 to  $N$  minus 1 and 0 to  $N$  minus 1 as valid indices and we say that board  $i$   $j$  is 1 to indicate that the queen is at  $i$  comma  $j$ .

And therefore, if it is 0 it indicates there is no queen. There are two possible values for every square. Of course, we also know that there is only one queen per row. This particular thing though it has  $N$  minus  $N$  into  $N$   $N$  square entries it will only have actually  $N$  ones at any given time. So, we can optimize this slightly by just having a single list with the entries 0 to  $N$  minus 1 where we say that the  $i$ th entry corresponds to the  $i$ th row and we record the column number. So, if board of  $i$  is equal to  $j$  it means that

in row i the queen is at column j. The queen is at position i comma j.

(Refer Slide Time: 10:36)

## Overall structure

```
def placequeen(i,board): # Trying row i
    for each c such that (i,c) is available:
        place queen at (i,c) and update board
        if i == n-1:
            return(True) # Last queen has been placed
        else:
            extendsoln = placequeen(i+1,board)
            if extendsoln:
                return(True) # This solution extends fully
            else:
                undo this move and update board
    else:
        return(False) # Row i failed
```

So, with such a data structure this is the outline of how our strategy works. So, what we have to do is place each queen one at a time. So, we are just writing a function which tries to place a queen in row i given the current state of the board. So, we pass it the current state of the board as one argument and we pass it the row number i that we are going to do. So, we would initially start with an empty board and with row 0. Now we run through each column and check whether the row column position that is the square i comma c is available, if it is available we then put a queen there and we of course, have to update the board. So, we will come back in a minute, but in our case updating our board just means setting board i equal to c if we have the one dimensional representation.

Now, if we have actually put the last queen, if I was N minus 1 then this is the last queen right. So, if it is an 8 queen problem then when we have put queen number 7 starting from 0 then we are done. So, we can return true; however, if this is not the last queen then we have to continue. So, what we need to do is now with the new board we have to place one more queen. So, we recursively call this function incrementing the row to i plus one with the updated board which we have just put and this will return true or false depending on whether it succeeds or not. So, we record it is return value in the name

extend solution. Depending on whether it succeeds or not we check if extend solution is true that is the current position reaches the end.

Now, when would it be true; if it succeeded in going all the way to level N minus 1 and N minus 1 returns true. So, when N minus 1 returns true then N minus 2 will return true and so on. Then our level I will also get the value true. Then we can also return true. So, if extend solution returns true we also return true saying that, so far I am good. On the other hand if extend solution returns false it means that given the current position that I chose for row I, nothing more could be done to extend this to a full solution. This position must be undone. So, we have to undo this move. So, we have to whatever we did earlier to update the board.

This update has to be reversed at this point. So, we have to reverse the effect of putting at i c and then, when we do this we will go back and we will try the next c and now if we have actually run through all the c's and we have not returned true at any point, then python has this else which says that the for loop terminated without coming out in between.

The for loop terminates normally it means we have run through every possible c that was available and for none of them did we return true; that means, that there is no way to currently put a queen on row i given the board that we have. So, we should return false saying that the board that we got is not a good one, then the previous row will now get a false and we try the next position and so on. This is a recursive solution that we get we will see an actual python implementation, but we have to do a little bit more work to figure out how to actually implement this.

The crucial thing in the implementation that we saw the previous one is, that we have to update the board when we place the queen and update the board when we undo it and we also have to check whether i c is available.

(Refer Slide Time: 14:01)

## Updating the board

- Our 1-D and 2-D representations keep track of the queens
- Need an efficient way to compute which squares are free to place the next queen
- $n \times n$  attack grid
  - $\text{attack}[i][j] == 1$  if  $(i, j)$  is attacked by a queen
  - $\text{attack}[i][j] == 0$  if  $(i, j)$  is currently available
- How do we undo the effect of placing a queen?
  - Which  $\text{attack}[i][j]$  should be reset to 0?

So, we had two representations, a two dimensional representation with 0s and ones and a one dimensional representation which gives us the column position for each row to keep track of the queens in the board, but in order to determine whether a square is free or not, we need to have a better way to compute how the squares are attacked by queens.

A simple way would be to just say that along with a two dimensional representation of the board we denote like we are done pictorially in the example we worked out we denote by what we have called this kind of colored square whether or not an attack a square is **attacked**. So, we say attack  $i j$  is 1, if it is attacked by queen otherwise it is 0. Now the problem with this is that a given square  $i j$  could be attacked by more than one queen right. So, when we undo a queen it will obviously, attack many squares, but not all those squares become free by removing that queen because, some of the squares are also attacked by other queens **which we had placed earlier**.

So, we need to be careful, when we remove a queen in order to mark squares which were attacked as being free. Well, one way to do this is to actually, number the queens and record the earliest queen that attacks each square.

(Refer Slide Time: 15:28)

## Updating the board

- Queens are added row by row
- Number the queens 0 to n-1
- Record earliest queen that attacks each square
  - `attack[i][j] == k` if  $(i, j)$  was first attacked by queen  $k$
  - `attack[i][j] == -1` if  $(i, j)$  is free
- Remove queen  $k$  — reset `attack[i][j] == k` to -1
  - All other squares still attacked by earlier queens

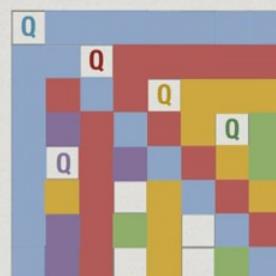
So, we say  $\text{attack } i \ j$  is  $k$  if  $i \ j$  was first attacked by queen  $k$  and  $\text{attack } i \ j$  is minus one if  $i \ j$  is free. So, when we remove queen  $k$  we reset  $\text{attack } i \ j$  with value  $k$  to minus 1 and all other squares are still attacked by earlier queens.

So, we can explain this very easily with the picture that we had before.

(Refer Slide Time: 15:58)

## 8 queens

- Can't place the a queen in the 8th row!
- Undo 7th queen, no other choice
- Undo 6th queen, no other choice



Here is how we had represented our board when we put the blue queen we marked all squares of the blue queen **attacked** with blue as blue solid squares then, when we put the red queen we only attack when we mark with red those squares new squares which **are attacked** for example, this particular square, which is attacked by both red and blue was already attacked by blue. So, we did not mark it. So, in this way with each new queen i that we put we only mark the squares which are attacked by queen i.

The colors here represent the queen numbers. The blue squares are queen 0, the red squares are queen one, the yellow squares are queen two and so on. So, when it comes to **undoing** it for a instance, now we want to undo this particular thing now this when we put it had only one white square, there was no free squares other than this. So, we did not add any new attack. So, removing it does not actually change anything regarding the attack position only makes that particular square itself free does not unattack any of the other squares.

Now, when we remove this orange queen, then we have to remove all the orange squares which were placed under attack only after adding this queen and that turns out to be these two on the bottom row. So, when we undo this one, we will find those two get **undone**. Similarly when we undo the purple. So, what we are done actually was precisely this more efficient implementation of how to keep things how to record what is under attack.

(Refer Slide Time: 17:22)

## Updating the board

- Queens are added row by row
- Number the queens 0 to n-1
- Record earliest queen that attacks each square
  - `attack[i][j] == k` if  $(i, j)$  was first attacked by queen  $k$
  - `attack[i][j] == -1` if  $(i, j)$  is free
- Remove queen  $k$  — reset `attack[i][j] == k` to -1
  - All other squares still attacked by earlier queens

So, we are going to now keep an attack array which says that attack  $i$   $j$  is  $k$ , if it is first attacked by queen  $k$  and when we remove queen  $k$  we reset to minus one saying that, that square is free precisely if the value is currently  $k$ . Now this would work the only difficulty is that it requires  $N$  square space, we saw that we could replace the board by a linear thing from a  $N$  by  $N$  array with 0s and ones, we could replace it by a single array which had board  $i$  equal to be  $j$ .

The question is can we replace attack by a linear array now one thing to remember is that though attack itself is an  $N$  squared array attack, undoing the attack does not require as to actually look at all the  $N$  squared entries once we fix the queen to undo, we only have to look along it is row, column and diagonal and remove all entries with the value equal to that queen on that row column and diagonal. The updates are not a problem the updates are linear, adding and removing a queen only requires us to look at a linear number of cells in this array, but the array itself is quadratic, so can we improve our representation to use only order  $N$  space.

(Refer Slide Time: 18:35)

## A better representation

- How many queens attack row i?
- How many queens attack row j?
- An individual square  $(i,j)$  is attacked by upto 4 queens
  - Queen on row i and on column j
  - One queen on each diagonal through  $(i,j)$

To do this we just have to look a little closer at the problem. So, how many queens attack row i now if we look at the row as a whole remember we place only one queen in each row and in each column. So, only the queen on row i actually attacks row i similarly only one queen is in column j. Therefore, there is only the queen in column j which attacks that column. If we look at an individual square then, if we are in the center of this for instance then this particular square can be attacked from 4 directions, can be attacked from the column in which it is or the row in which it is or it can be attacked from this main diagonal or the off diagonal.

The main diagonal is the one from top left which is called north west and the one, the off diagonal is the one from the south west. There are 4 possible queens that could be attacking this square. There are 4 directions in which a square could be under attack. It might be better to represent these 4 directions rather than the squares itself the representation we have now is to say that this particular square is attacked by queen k, but it does not tell us from which direction queen k is attacking right it does not tell us whether queen k is attacking it from the row or the column or the diagonal.

(Refer Slide Time: 20:05)

## Numbering diagonals

- Decreasing diagonal:  
column - row is invariant
- Increasing diagonal:  
column + row is invariant
- $(i,j)$  is attacked if
  - row  $i$  is attacked
  - column  $j$  is attacked
  - diagonal  $j-i$  is attacked
  - diagonal  $j+i$  is attacked

c+r=12

So, rows and columns are naturally numbered from 0 to 7, but how about diagonals.

Now if we look at a diagonal from the north west. Let us call these directions north west, south west, north east and south east. If you look at a decrease in diagonal a diagonal that goes from top to bottom like this, then what we find is that this difference the column minus the row is something that will be the same along every square on that diagonal, for instance look at this diagonal it starts here.

Here the column number is 2 and the row is 0. 2 minus 0 is 2, if we go to the next item of the diagonal is 3 minus 1 which is again 2 then 4 minus 2 is again 2 and so on. So, if we go along this diagonal for all these squares,  $c - r$  where  $c$  is the column number and  $r$  is a row number the difference is exactly 2 and you can check that nowhere else on the square on this grid is this true, as another example if you look at this particular thing. We have 0 minus four. The difference is minus 4 and similarly 3 minus 7 is also minus 4. So, everything along this particular diagonal has a difference minus 4.

Now, if we look at the diagonals going the other way then we find that the sum is an invariant here for instance we have either 6 plus 0 or 5 plus 1 or 4 plus 2 and 2 plus 3, 3 plus 3 and so on. So, along this purple diagonal  $c + r$  is equal to 6 everywhere, and along this green diagonal we have 7 plus 5, 6 plus 6 and 5 plus 7. So,  $c + r$  is equal to

12. So, we can now conclude that the square at position  $i$   $j$  is attacked, if it is attacked by queen in row  $i$  or in column  $j$  or if it is along the diagonal whose difference is  $j$  minus  $i$  or if it is along the diagonal whose difference is  $j$  plus  $i$  whose sum is  $j$  plus  $i$ .

(Refer Slide Time: 22:00)

## O(n) representation

- `row[i] == 1` if row  $i$  is attacked,  $0..N-1$

So, we can now come up with a representation which only keeps track of rows, columns and diagonals which are under attack and from that we can deduce, whether a square is under attack. So, we say that row  $i$  is 1, if row  $i$  is under attack where  $i$  ranges from 0 to  $N$  minus 1 similarly; we can have a an array which says column  $i$  is attacked and then column  $i$  is set to 1 provided column  $i$  is attacked for again  $i$  between 0 and  $N$  minus 1. Now when we look at the diagonals we have these two types of diagonals.

The north west to south east diagonal is the one where the difference is the same and if you look at the differences, if you go back then you see the differences at this diagonal here, the difference is 7 minus 0 is 7 and here the difference is 0 minus 7 is minus 7.

(Refer Slide Time: 22:48)

## Numbering diagonals

- Decreasing diagonal:  
column - row is invariant
- Increasing diagonal:  
column + row is invariant
- $(i,j)$  is attacked if
  - row  $i$  is attacked
  - column  $j$  is attacked
  - diagonal  $j-i$  is attacked
  - diagonal  $j+i$  is attacked

7-0  
=7  
0 1 2 3 4 5 6 7  
1  
2  
3  
4  
5 c+r=12  
6  
7  
0 -7 = -7

It goes from plus N minus one to minus N minus 1. On the other hand, if you go the other way then the sum at this point is 0 plus 0 is 0, and the sum over here is 7 plus 7 is 14. The sum along these diagonals are 0, 1, 2, 3, 4 and so on. This is one this is 2 this is 3 and so on.

(Refer Slide Time: 23:11)

## O(n) representation

- `row[i] == 1` if row  $i$  is attacked,  $0..N-1$
- `col[i] == 1` if column  $i$  is attacked,  $0..N-1$
- `NWtoSE[i] == 1` if NW to SE diagonal  $i$  is attacked,  $-(N-1)$  to  $(N-1)$
- `SWtoNW[i] == 1` if SW to NE diagonal  $i$  is attacked,  $0$  to  $2(N-1)$

( $i,j$ )

So, we have **these** north west to south east diagonals running from minus N minus 1 to N minus 1 this gives me the number if at. This is the difference if the difference is say 6 I know which squares are there if the difference is minus 3. I know which squares are there and the possible range of values is from minus 7 to plus 7 minus N minus 1 to plus N minus 1 and for the other direction it is from 0 to 2 times N minus 1 in our case two times N minus 1 is two times 7 which is 14.

So, 0 to 14, but if we have an N by N thing we have two times N minus 1. This gives us an order N representation of the squares under attack. Therefore, we look for if we want to see if i j squares under attack we check whether it is row i is one or column j is 1 or j minus 1, diagonal is 1 or i plus j diagonal is 1. If any of these is 1, then **it** is under attack if all of these are 0 then is not under attack right.

(Refer Slide Time: 24:07)

## Updating the board

- $(i, j)$  is free if  
 $\text{row}[i] == \text{col}[j] == \text{NWtoSE}[j-i] == \text{SWtoNE}[j+i] == 0$
- Add queen at  $(i, j)$   
 $\text{board}[i] = j$   
 $(\text{row}[i], \text{col}[j], \text{NWtoSE}[j-i], \text{SWtoNE}[j+i]) = (1, 1, 1, 1)$
- Remove queen at  $(i, j)$   
 $\text{board}[i] = -1$   
 $(\text{row}[i], \text{col}[j], \text{NWtoSE}[j-i], \text{SWtoNE}[j+i]) = (0, 0, 0, 0)$

So,  $i, j$  is free provided row  $i$  column  $j$  the north west to south east diagonal  $j$  minus  $i$  and the south west to north east diagonal  $j$  plus  $i$  are all equal to 0. When we add a queen at  $i, j$  first we update the board representation to tell us that there is, now the  $i$ th row is set to the  $j$ th column and for the appropriate row, column and diagonal corresponding to this square we have to set all of them to be under attack.

So, row i becomes under attack, column j becomes under attack the j minus one th diagonal on the decreasing diagonal and j plus i th diagonal on the increasing diagonal all get set to one; **And** undo is similarly, easy we have to first reset the board value to say that the ith queen is not placed. So, we could say minus one this is not a valid value because the values are 0 to N minus 1. So, minus 1 indicates that the ith queen is not placed at this moment and we reset this row and this column to be equal to 0 because, this row and this column are attacked only by this queen.

Remember we cannot have two queens on the same diagonal because, they would attack each other. So, at any given point each one of these rows columns and diagonals is attacked by a single queen and it must be attacked by the queen at i comma j. So, only the queen at i comma j can attack all of these because, if it was under attacked by another queen we could not placed a queen at i comma j.

The fact that this free before indicates that all of these got attacked only by the current queen. So, when we remove the current queen we must reset them back to 0.

(Refer Slide Time: 25:35)

## Implementation details

- Maintain `board` as nested dictionary
  - `board['queen'][i] = j` : Queen located at (i, j)
  - `board['row'][i] = 1` : Row i attacked
  - `board['col'][i] = 1` : Column i attacked
  - `board['nwtose'][i] = 1` : NWtoSW diagonal i attacked
  - `board['swtne'][i] = 1` : SWtoNE diagonal i attacked

One implementation detail for python is that instead of keeping these 5 different data structures, we have a board **and** a row and a column and all that we keep it as a single

nested dictionary. So, it is convenient to call it board and we will have at the top 5 key values indicating the 5 sub dictionaries. The queen position we will call the key queen. So, instead of saying board i is j, we will say board with queen as the key at position i is j then we will say instead of row i is 1 or minus 1 we will say that the board at key row is one similarly board at key column board at north west to south east and board at south west to north east.

So, we have just converted it. So, we do not have to pass around 5 different parts to each function we just have to pass a single board which is a dictionary which contains everything of interest.

Remember that this is how we try to give our solution. So, we wanted to place the queen in row i and for each column that is available we would try to update the board and so on. Now, we have now better ways to do these things right. So, we have shown that using these dictionary or these 5 different representations we can check whether a row and column is available, how to update the board when we place a queen and we undo the queen.

Here we have an actual python implementation of what we discussed. So, we have this function here which is called place queen. Place queen we said takes the row i and the board and the first thing it does it has to determine. What is the value of n? So, we just take.

Remember that board is now a dictionary. So, board of queen will tell us how many rows there are in the thing. If we take the length of the keys of board of queen we get n. This is just way of recovering N without passing it around. Now, what we do is for every possible value from 0 to N minus 1 that is for j, for all column values we check if i j is free in the current board. If it is free then we add a queen this is exactly the code that pseudo code we had if, i is N minus 1 we return true otherwise we try to extend the solution by placing a queen at i plus 1th row. If the solution does extend we return true otherwise we undo the queen.

So, undoing the queen will remove this queen and also update the board and finally, if

this loop goes all the way through for every possible column and does not return true then we means it means we cannot place the queen on the  $i$ th row. So, we return false. Now the main function that we have the main code will start off by initializing board to be an empty dictionary, it will ask the user how many queens what kind of board we have  $N$  by  $N$ . So, remember we take the input it will be a string we convert it using int and we record this as  $N$ .

So, it asks for us number converts it to an int and passes it as  $N$  then we will initialize the board with the number  $N$ . We need  $N$  because, we need to know how to set up that remember that the indices run from 0 to  $N$  minus 1 or  $N$  minus  $N$  minus 1 plus  $N$ .  $N$  is required in order to initialize the dictionary and finally, we try to place the queen. So, initialization will setup an empty board where nothing is under attack then we try to place a queen in the 0th row on this board, if it succeeds then we have a function which prints the board.

Let us see how these other functions work. Let us first look at the function which initializes the board. Initializing the board says that first of all for every key for each of these sub dictionaries queen row column north west south east south west or north east we first set up a dictionary with that key. So, this says create an empty dictionary; Now for three of these things for queen, row and column right the indices are 0 to  $N$  minus 1. For  $i$  in range we just set up the key value  $i$  to point to minus 1 in case of queen this says that the queen in row  $i$  has not been placed and for row and column these are the attacked ones which says that they are 0 if they are under not under attack and one if they are under attack. The initial thing is to say 0.

Now, similarly for the north west to south east the range goes from minus  $N$  minus minus  $N$  minus 1 to plus  $N$  minus 1. So, from the range function since we give the upper bound as  $N$ . We set every key in this to 0 similarly for 0 to 2  $N$  2 into  $N$  minus 1 we want to set the south west to north east diagonals to be 0 this is one reason here why we are using a dictionary because for the other things of course, we could use a list right 0 to  $N$  minus one is the natural list index, but here we have the strange indices which go from minus  $N$  minus one to plus  $N$  minus 1 and so on.

That is why we use a two level nested dictionary. This initializes the board; what how do we print a board well for every row we sort the rows. So, we take board dot queen dot keys will give us 0 1 upto N minus 1 in some random order we sort them and for each such row we print the row and the column number for that row. This happens when we have a successful solution. When is a position free well we check whether board the row entry is 0 the column entry is 0 the diagonal entry  $j - i$  is 0 and the diagonal entry  $j + i$  is 0 this is exactly as we said before and finally, what happens when we add a queen right when we add a queen we have to place it.

So, we set the queen entry for row  $i$  to  $j$  and then we mark the corresponding row column and diagonal to be one and when we undo a queen we set the queen entry to be minus 1 and the row column and diagonal entries to be 0; these are all exactly what we wrote in the pseudo code that has been formalized in python code. Now we can run this code and verify that it works. So, here we have this code 8 queens dot py which is the code that we just saw in the editor.

Now if I run this code as python 3.5 8 queens dot py. This is by the way if you have a python program you can run it directly without first invoking it and then importing it if you do this it will ask us how many queens we want. For instance if we give it the number 4 then we will get the solution that we saw in the earlier example it is not very printed out very neatly. So, if we give the number 8 then we will get one solution like this the it turns out that you can actually change that print board function i would not show you the code, but to print it out in a more user friendly way. So, I have another function which is called pretty if I do this then it shows me the 4 queen solution in a more readable form right.

So, you see exactly the kind of off diagonal positions and if I do for 8 queens then you see there is an extra column. There is some mistake in that, but there is an extra column, but basically you can see that if you ignore the last column which is showing the position of the queens in the first 8 queen solution. So, it is fairly straight forward once we have got the representation worked out and the structure of the code worked out, it is very easy to transform it into actual python code.

(Refer Slide Time: 32:54)

## All solutions?

```
def placequeen(i,board): # Try row i
    for each c such that (i,c) is available:
        place queen at (i,c) and update board
        if i == n-1:
            record solution # Last queen placed
        else:
            extendsoln = placequeen(i+1,board)
            undo this move and update board
```

As a final step suppose we want not one solution, but all solutions right. So, we do not wants the previous thing in the moment it find a solution then it returns true and then every previous level also returns true and eventually it print out the board. Supposing we do not want to stop at the first solution, but keep printing out it is actually much easier; then what we do is we just keep going through all possible positions and whenever we reach the final step if we actually a solution reaches the final step then we record it in our case it might print it otherwise we extend it and go to the next one.

Actually it is much simpler to print all solutions than it is print a single solution because we do not have to remember whether our solution extends or not it is really running through every possible solution. The only thing is that it will not run through every solution to the very end and then decide it does not work. It is not like we are putting all possible queen positions and then trying it out we are trying it out for smaller things, because once we get stuck at say position 5 then it would not try to extend this it will come back and so on, but this is just a much simpler loop which just prints all solutions.

So, here is the code it's exactly the same code otherwise the only thing is the place queen function is much simpler now, we just try for every j and range one to 0 to N minus 1, if it is free we add the queen, if we have reached the last row we print the board, we extend

the solution and then we undo the queen and try the next one.

For every  $j$  we are going to first add the queen, if it manages to place it extend the solution and finally, we are going to undo it and try the next  $j$ ; we're just going to blindly try every possible  $j$  and we are not going to ever come out complaining that we have not succeeded the rest is pretty much the same, the print board has been changed slightly and slight change in the print board is just that we have changed it so that, it will print the entire thing on a single row. So, we have added this thing which says, end equal to space. So, we print the positions in a single row rather than row by row, that we can see them all.

Now if we look at the function now and we try to print it say for 4 queens, then it prints two solutions, these are essentially two rotated solutions of the same thing. If we do it for 8 queens for instance then it will actually produce a vast number of solutions it turns out there are actually 92 solutions, but even these 92 solutions if you look at rotations and reflections they come out to be much less, but if you just look at a position of the square as it is given to you then, there are 92 different solutions that it prints out. This concludes our discussion of backtracking with respect to the 8 queens' problem.

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

**Week – 06**  
**Lecture – 02**  
**Global Scope, Nested Functions**

(Refer Slide Time: 00:02)

### Recall 8 queens

```
def placequeen(i,board): # Trying row i
    for each c such that (i,c) is available:
        place queen at (i,c) and update board
        if i == n-1:
            return(True) # Last queen has been placed
        else:
            extendsoln = placequeen(i+1,board)
            if extendsoln:
                return(True) # This solution extends fully
            else:
                undo this move and update board
    else:
        return(False) # Row i failed
```

We were looking at the 8 queens problem, and our solution involved representing the board, which squares **are** under attack and placing the queens one by one.

One feature of this solution is that we had to keep passing the board through the functions in order to update them or to resize them **and I** had to initialize them and so on because the board had to **kept** updated through each function. Now **the question is** can **we avoid** passing the board around all over the place?

(Refer Slide Time: 00:31)

## Global variables

- Can we avoid passing `board` explicitly to each function?
- Can we have a single **global** copy of `board` that all functions can update?

So, can we avoid passing this board explicitly or can we have a single global copy of the board that all the functions can update which will save us passing this board back and forth.

(Refer Slide Time: 00:42)

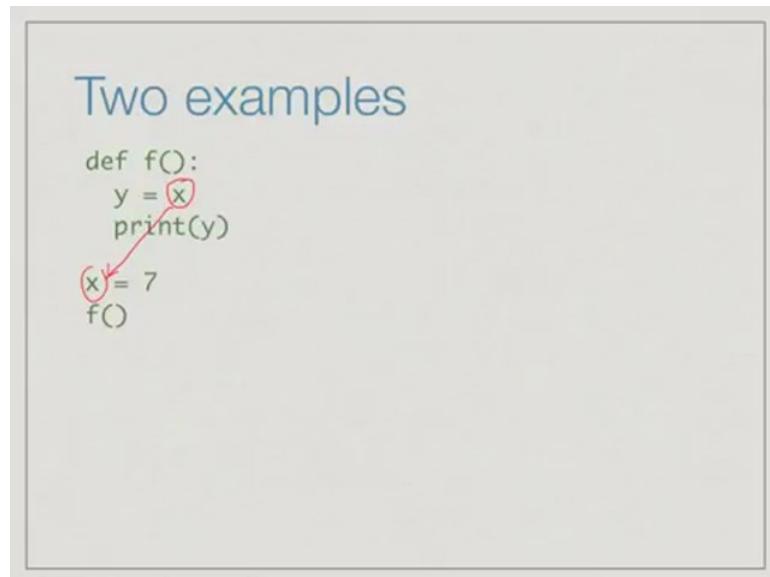
## Scope of name

- Scope of name is the portion of code where it is available to read and update
- By default, in Python, scope is local to functions
  - But actually, only if we update the name inside the function

So, this **brings us** to a concept of Scope. The scope of a name in Python is **the** portion of the code where it is available to read an update. Now by default in python scope is local to a function, we saw that if we use a name inside a function **and** that it is different from

using the same name outside the function. But actually this happens only when we update the name inside the function.

(Refer Slide Time: 01:08)



Let us look at this particular code. Here we have a function `f` which reads the values `x` and prints it by storing it in the name `y`, Now the question is: what is this `x`? Well there is an `x` here. So, will this `x` inside the function correctly reflect the `x` outside the function or not.

(Refer Slide Time: 01:33)

```
madhavan@dolphinair:...016-jul/week6/python/scope$ more f1.py
def f():
    y = x
    print(y)

x = 7
f()
madhavan@dolphinair:...016-jul/week6/python/scope$ python3.5 f1.py
7
madhavan@dolphinair:...016-jul/week6/python/scope$ []
```

So here we see that function, we have written a file f1 dot py which contains exactly that code. So, we have function f which reads an x from outside and tries to print it. If you run this, then indeed it prints the value 7 as we expect, so y gets the value 7 because the x has the value 7 outside and that x is inherited itself a function from inside f.

(Refer Slide Time: 02:03)

## Two examples

```
def f():
    y = x
    print(y)
x = 7
f()
Fine!
```

```
def f():
    y = x
    print(y)
    x = 22
x = 7
f()
```

So this works. Now what if you do this, and this is exactly the same function except that after printing the values of y it sets x equal to 22 inside f. Now what happens?

(Refer Slide Time: 02:19)

```
madhavan@dolphinair:...016-jul/week6/python/scope$ more f1.py
def f():
    y = x
    print(y)

x = 7
f()
madhavan@dolphinair:...016-jul/week6/python/scope$ python3.5 f1.py
7
madhavan@dolphinair:...016-jul/week6/python/scope$ more f2.py
def f():
    y = x
    print(y)
    x = 22

x = 7
f()
madhavan@dolphinair:...016-jul/week6/python/scope$ python3.5 f2.py
Traceback (most recent call last):
  File "f2.py", line 7, in <module>
    f()
  File "f2.py", line 2, in f
    y = x
UnboundLocalError: local variable 'x' referenced before assignment
madhavan@dolphinair:...016-jul/week6/python/scope$
```

So here is f2 dot py the code in **the middle** of the screen, so only difference with **respect to** f1 dot py is extra assignment x equal to 22 inside f. Now if you try to run f2 dot py, then it gives us an error saying that the original assignment y equal to x gives us an unbound local name there is no x which is available at this point inside f. So, somehow assigning x equal to 22 inside f **changed** the status of x, it is no longer willing to look up the outside x it will insist that there is an inside x. This gives as an error.

(Refer Slide Time: 02:59)

## Two examples

<pre>def f():     y = x     print(y) x = 7 f()</pre>	<b>Fine!</b>	<pre>def f():     y = x     print(y)     x = 22 x = 7 f()</pre>	<b>Error!</b>
--	--------------	---	---------------

- If x is not found in f(), Python looks at enclosing function for **global** x
- If x is updated in f(), it becomes a **local** name!

So **if** x is not found in f, Python is willing to look at the enclosing function for a global x. However, if x is updated in f then it becomes a local name and then it gives an error.

(Refer Slide Time: 03:15)

## Global variables

- Actually, this applies only to immutable values

```
def f():
    y = x[0]
    print(y)
    x[0] = 22

x = [7]
f()
```

So strictly speaking this applies only to immutable values. If we change this function as follows we made x not an integer, but a list for example and we asked y to pick up the 0th element in the list and then later in f we change the 0th element of x to 22.

(Refer Slide Time: 03:38)

```
madhavan@dolphinair:...016-jul/week6/python/scope$ more f3.py
def f():
    y = x[0]
    print(y)
    x[0] = 22

x = [7]
f()
madhavan@dolphinair:...016-jul/week6/python/scope$ python3.5 f3.py
7
madhavan@dolphinair:...016-jul/week6/python/scope$
```

Here we have this function in which we now changed x from an integer to a list and then we try to assign it in y. But we update that list inside the function and then if you run it then it does print the value 7 as we expect.

(Refer Slide Time: 03:55)

## Global variables

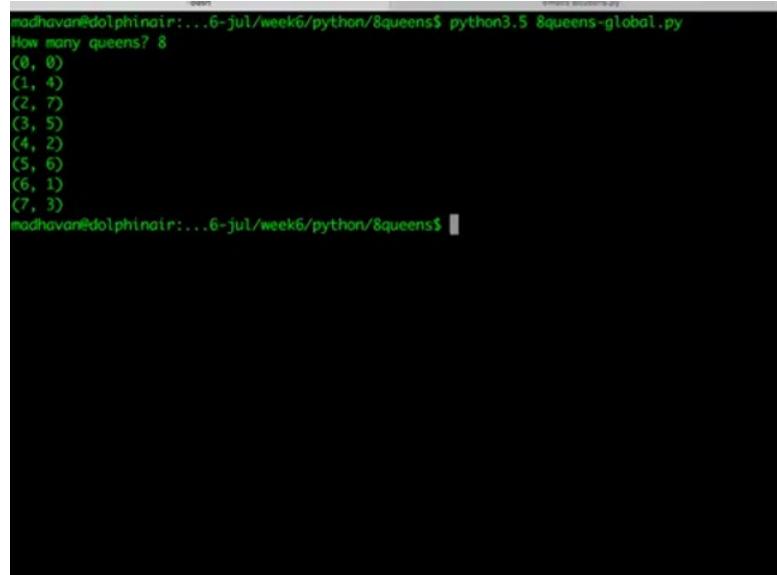
- Actually, this applies only to immutable values
- Global names that point to mutable values can be updated within a function

```
def f():
    y = x[0]
    print(y)
    x[0] = 22
x = [7]
f()
Fine!
```

So this works. If we have an immutable value, I mean mutable value sorry, then we can actually change it inside `f` and nothing will happen. So, global names that point to mutable values can be updated within a function.

In fact, this means therefore that the problem that we started out to solve namely; how to avoid passing the board around with its inside 8 queens actually requires no further explanation. Since board is a dictionary, it is a mutable value and in fact we can write 8 queens in such a way that we just ignore passing the board around, we change all the definitions so that board does not occur and works fine.

(Refer Slide Time: 04:33)



```
madhavan@dolphinair:...6-jul/week6/python/8queens$ python3.5 8queens-global.py
How many queens? 8
(0, 0)
(1, 4)
(2, 7)
(3, 5)
(4, 2)
(5, 6)
(6, 1)
(7, 3)
madhavan@dolphinair:...6-jul/week6/python/8queens$
```

Here we have rewritten the previous code just removing board from all the functions. So initialize earlier took board and n now it just takes n print board does not taken argument at all and all of them are just referring to this global value board which you can see everywhere. So, we have this global value board here which is being referred to inside the function and it does not matter that is not being passed because this is the mutable value, so it is going to look for the value which is defined outside namely this empty dictionary. And then all these functions like place queen or undo queen or add queen just take their relevant parameters and implicitly refer to the global value of board.

So, if you run this now this global version, we get exactly the same output. So, as we said for our purpose which is to fix that 8 queens problem without having to pass the board around, the fact that python implicitly treats mutable global names as updatable within a function is all that we need.

(Refer Slide Time: 05:36)

## Global immutable values

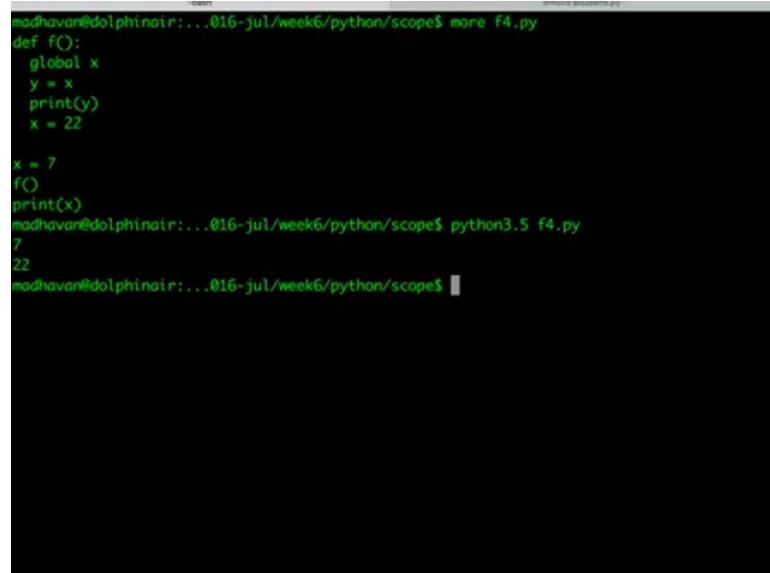
- What if we want a global integer
- Count the number of times a function is called
- Declare a name to be `global`

```
def f():
    global x
    y = x
    print(y)
    x = 22
x = 7
f()
print(x)
```

But, what if we actually want a global integer? Why would you want a global integer? Well, suppose for instance we want to count the number of times a function is called. So every time a function is called we would like to update that integer inside this function. But that integer cannot be a local name to the function because that local value will be destroyed in the function, we want it to persist, so it must be a value which exists outside the function. But being an integer it is an immutable value and therefore we try to update it inside the function it will treat it as a local value. So, how do we get around this?

Python has a declaration called Global, which says a name is to be referred to globally and not taken as a local value. If we change our earlier definition of f, so that we add this particular tag `global x` then it says that this x and this x both refer to the same x outside. This is the way of telling python, do not confuse this x equal to 22 with creation of a new local name x. All x's referred to in f are actually the same as the x outside and to be treated as global values. So, this is one way in which we can make an immutable value accessible globally within a Python program.

(Refer Slide Time: 06:51)



```
madhavan@dolphinair:...016-jul/week6/python/scope$ more f4.py
def f():
    global x
    y = x
    print(y)
    x = 22

x = 7
f()
print(x)
madhavan@dolphinair:...016-jul/week6/python/scope$ python3.5 f4.py
7
22
madhavan@dolphinair:...016-jul/week6/python/scope$
```

So here is that global code. We have global x, and just make sure that the x is equal to 22 inside is actually affecting that x outside. We have a print x now after the call to f. So the bottom of the main program we have print x, now x was 7 before f was called but x got set to 22 inside f. So, we would expect the second print statement to give us 22. This statement should first print 7. It should print a 7 from the print y and then print 22 from this print x. So, if you run this indeed this is what we see right we have two lines, the first 7 comes from print y and the second level 22 comes from print x outside.

(Refer Slide Time: 07:33)

## Nest function definitions

- If we look up x, y inside g() or h() it will first look in f(), then outside
- Can also declare names global inside g(), h()
- Intermediate scope declaration: nonlocal
- See Python documentation

```
def f():
    def g(a):
        return(a+1)

    def h(b):
        return(2*b)

    global x
    y = g(x) + h(x)
    print(y)
    x = 22

    x = 7
    f()
```

While we are on the topic of local scope, Python allows us to define functions within functions. So, here for instance the function f has defined functions g and h, g of a returns a plus 1, h of b returns two times b. Now we can update y for instance by calling g of x plus h of x rather than just setting it the value x.

Now, the point to note in this is that these functions g and h are only visible to f. They are defined within this scope of f. So, they are inside f, and hence they are not visible outside. So, from outside if I go if I ask g of x at this point this will be an error, because it will say there is no such g defined. This is useful because now you can define local functions which we may want to perform one specialized task which are relevant to f, but it should not be a function which is exposed to everybody else and this is a possibility.

Of course, the same rules apply so if we look up x inside g or h. So, if we look up an x here it will first try to look up f, if it is not there in f it will go outside and so on. So, either we will declare it global in which case we can update it within g or h or it will use the same rule as before if we do not update an immutable value it will look outside and if it is a mutable value it will allow us to update it from inside.

Now there are some further refinements. Python has an intermediate scope called non local which says within g and h refer to the value inside f, but not to the value outside f. This is a technicality and it will not be very relevant if we need it we will come back to it, but for the moment if you want to find out more about non local declarations please see the Python documentation. But global is the important one, global allows us to transfer an immutable value from outside in to a function and make it updatable within a function.

(Refer Slide Time: 09:33)

## Summary

- Python names are looked up inside-out from within functions
- Updating a name with immutable value creates a local copy of that name
  - Can update global names with mutable values
  - Use `global` definition to update immutable values
  - Can nest helper function — hidden to the outside

To summarize, what we `have` seen is `that` Python names `are` looked up inside out from within functions, if we update an immutable value it creates a local `copy` so we need to have a `global` definition to update immutable values.

On the other hand, if you have mutable values like `lists` and dictionaries there is no problem. Within a function we can implicitly refer to the global one and `update` it. And this we saw in our 8 queens solution, we can make the board into a global value and just keep updating it within each function rather than passing it around explicitly as an argument.

Finally, what we `have` seen is that we can nest functions. We can create so called helper functions within functions that are hidden to the outside that can be `used` inside `the` function to logically break up its activities in to smaller units.

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

**Week - 06**  
**Lecture - 03**  
**Generating Permutations**

(Refer Slide Time: 00:03)

## Backtracking

- \* Systematically search for a solution
- \* Build the solution one step at a time
- \* If we hit a dead-end
  - \* Undo the last step
  - \* Try the next option

We will be looking at Backtracking. In backtracking we systematically search for a solution one step at a time and when we hit a dead end we undo the last step and try the next option.

(Refer Slide Time: 00:14)

## Generating permutations

- \* Often useful when we need to try out all possibilities
  - \* Each potential columnwise placement of N queens is a permutation of  $\{0, 1, \dots, N-1\}$
  - \* Given a permutation, generate the next one
  - \* For instance, what is the next sequence formed from  $\{a, b, \dots, m\}$ , in dictionary order after

d c h b a e g l k o n m j i

Now, in the process of doing the backtracking we need to generate all the possibilities. For instance, remember when we try to printout all the queens we ran through every possible position for every queen on every row, and if it was free then we tried it out and if the solution extended to a final case then we print it out. Now if we look at the solutions that we get for the 8 queens, each queen on row is in a different column from every other queen. The column numbers if we read then row by row, the column numbers form a permutation of 0 to N minus 1. So, each number 0 to N minus 1 occurs exactly once as a column number for the n queens.

So, one way of solving a problem like 8 queens or similar problems is actually it generate all permutations and keep trying them one at a time. This give rise to the following question; if we have a permutation of 0 to N minus 1 how do we generate the next permutation. This is like thinking of it as a next number, but this could be in an arbitrary number of symbols.

Suppose, we have the letters a to m. So, these are the first thirteen letters of the alphabet and we treat the dictionary order of words as the ordering of numbers, we think of them as digits if you want to think of it is base **thirteen**. Here for instance, is a number in a base thirty or now alternatively a rearrangement of a to m in some order. Now what we want to is, what is the next rearrangement after this you immediately next one in dictionary order.

(Refer Slide Time: 01:54)

## Generating permutations

- \* Smallest permutation — all elements in ascending order

a b c d e f g h i j k l m

- \* Largest permutation — all elements in descending order

m l k j i h g f e d c b a

- \* Next permutation — find shortest suffix that can be incremented

- \* Or longest suffix that cannot be incremented

In order to solve this problem the first observation we can make is that, if we have a sequence of such letters or digits the smallest permutation is the order in which the elements are arranged in ascending order. So we start with a which is smallest one then b and c and so on and there is no smaller permutation than this one. Similarly, the largest permutation is one in which all the elements are in descending order, so we start with the largest element m and we work backwards down to a.

If we want to find the next permutation we need to find as short suffix as possible that can be incremented, it is probably easiest to do it in terms of numbers but let us do it with letters. The shortest suffix that can be incremented consists of something followed by the longest suffix cannot be incremented. So this will become a little clear when we work through an example.

(Refer Slide Time: 02:56)

## Next permutation

- \* Longest suffix that cannot be incremented
- \* Already in descending order

d c h b a e g l k o n m j i

- \* The suffix starting one position earlier can be incremented
  - \* Replace k by next largest letter to its right, m
  - \* Rearrange k o n j i in ascending order

d c h b a e g l m i j k n o

next

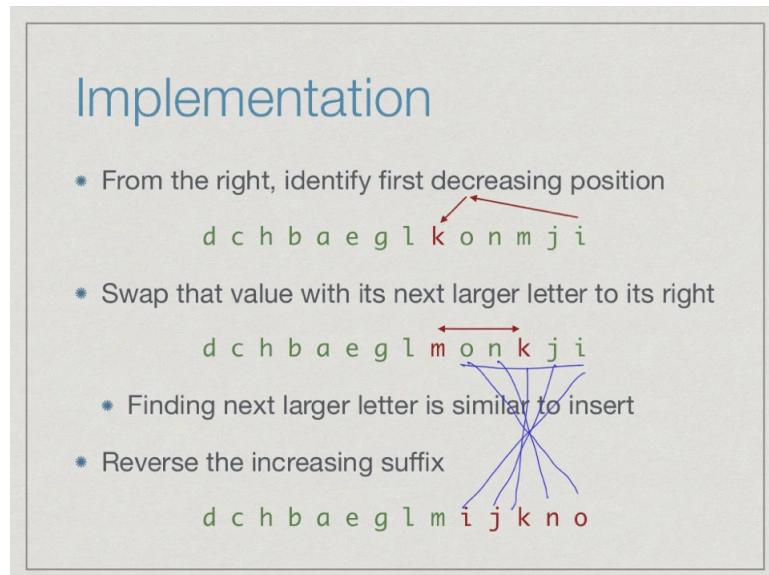
We want to find the longest suffix that cannot be incremented. So, a suffix that cannot be incremented is one which is as large as it could possibly be which means that it is already in descending order. If you look at example that we had before for which we want to define the next permutation, we find this suffix o n m j i these five letters are in descending orders so I cannot make any larger permutation using this.

So, if I fix the letter from d to k then this the largest **permutation** I can generate with d to k fixed. If I want to change it and need to increment something and I mean to increment it, I cannot increment it within this red box so I must extend this to find the shortest suffix namely; suffix started with k where something can be incremented. Now how do we increment this? Well, what we need to do is that now is like say that we have with k we cannot do any better so we have to replace k by something bigger and the something bigger has to be the smallest thing that we can replace it by, so we will replace k at the next largest letter to its right namely m.

Among these letters m n and o are bigger than k if I replace it by j or i, I will get a smaller permutation which I do not want, so I may replace it m n or o, but among these this m is the smallest I must now start a sequence where the suffix of length six begins with the letter m. And among suffix **that** begins with letter m I need the smallest one, that mean I rearrange the remaining letters k o n j i in ascending order to give me the smallest permutation to begin with m and has the letters k o n j i after it.

This gives me this permutation. So, I have now moved this m here and I have now taken these letters and rearrange them in an ascending order to get i j k n o. Therefore, this means that for this permutation the next permutation is this one.

(Refer Slide Time: 04:55)



So, algorithmically we can do it as follows, what we need to do is first identify the suffix that can be incremented. We begin by looking for suffix that cannot be incremented namely we go backwards so long as it is in descending order. So we keep looking for values as they increase. So, i is smaller than j, j is smaller than m, m is smaller than n, n is smaller than o, but o is bigger than k so that means than up to here we have a suffix that cannot be incremented and this is the first position where we can make an increment.

Having done this we now need to replace k by the letter to its right which is next bigger. Now this is a bit like insert we go one by one, we say that k smaller than n so we continue, and we say than k is bigger than j so we stop here. So this tells us that the letter m is the one we want. We can identify this in one scan, because this remember it is in descending order, it is in sorted order so we can go through and find the first position where we crossed on something bigger than k to something smaller than k and that is the position of the letter that we need to change. So, it is exactly like inserting something into a sorted list.

Now having done this, we have exchange this m and k now we need to put this in ascending order, but remember it was in descending order and what we did to the

descending order we replace m by k but what are the property of k? k was smaller than m but bigger than j so, o n k j i remains in descending order. If we want convert it to ascending order we do not need to sort it we just need to reverse it we just needed backwards, so this is just the reversal of this.

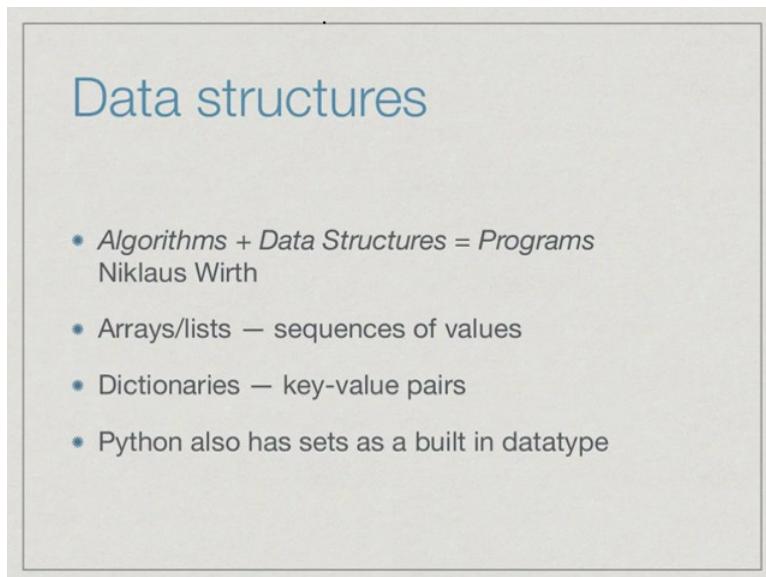
This is a concrete way in all to which find the next permutation, walk backwards from the end and see when the order stops increasing. So, wherever we first decrease this that is the suffix that you want to increment, of course if we go all the way and go back to the first letter and we are not found such a position then we have already reached the last permutation in the overall scheme of things.

Once we find such a position we find which letter to swap it with by doing equivalent of the search that we do for insertion sort. So, we do an insert kind of thing find the position in this case m to swap with k after swapping it we take the suffix after the new letter we put namely m and we reverse it to get the smallest permutation starting with that letter m.

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

**Week - 06**  
**Lecture - 04**  
**Sets, Stacks, Queues**

(Refer Slide Time: 00:01)



**Data structures**

- *Algorithms + Data Structures = Programs*  
Niklaus Wirth
- Arrays/lists — sequences of values
- Dictionaries — key-value pairs
- Python also has sets as a built in datatype

In the 1970s Niklaus Wirth, the inventor of the programming language Pascal wrote a very influential book called Algorithms plus Data Structures **equals** Programs. So, the title **emphasises** the importance of both algorithms and data structures as components of effective programs.

So far we have seen algorithms in some detail. So, now let us take a closer look at some specialized data structures. The data structures that we have seen that are built into python began with arrays and lists which are just sequences of values. We also saw dictionaries which are key value pairs and which are very useful for maintaining various types of information. **Another** built in data type that is available in python is the set.

(Refer Slide Time: 00:48)

## Sets in Python

- List with braces, duplicates automatically removed

```
colours = {'red', 'black', 'red', 'green'}  
>>> print(colours)  
{'black', 'red', 'green'}
```

- Create an empty set

```
colours = set()  
• Note, not colours = {} — empty dictionary!
```

Set is like a list except that you do not have duplicates. In python, one way of writing a set is to write a list with braces like this. So, here we have associated with the name colours a list of values red, black, red and green. Notice that in setting it up, we have repeated red, but because this is a set, the duplicate red would be automatically removed. So, if we print the name colours, we just get the list black, red and green. Now, since the empty brace notation is already used, for empty dictionary if we want to create an empty set, we have to call the set function as follows.

(Refer Slide Time: 01:33)

## Sets in Python

- Set membership

```
>>> 'black' in colours  
True
```

- Convert a list into a set

```
>>> numbers = set([0,1,3,2,1,4])  
>>> print(numbers)  
{0, 1, 2, 3, 4}  
  
>>> letters = set('banana')  
>>> print(letters)  
{'a', 'n', 'b'}
```

So, we say colours equal to set with no arguments. Like lists and other data structures, we can test membership using in. So, if in the previous lists set colours which had red, black and green, we ask whether black is in colours by using the word in, then, the return value is true. In general we can convert any list to a set using the set function.

We saw that if we give no arguments to set you get an empty set, but if we give a list such as this 1, 3, 2, 1, 4 with duplicates and assign it to the name numbers, then because its a set the duplicate ones will be removed and we will get a list of, we will get a set of numbers 0, 1, 2, 3, 4. Notice again that the order in which the set is printed need not be the order in which you provided it. This is very much like a dictionary sets; are optimized for internal storage to make sure there are no duplicates etcetera.

So, we should not assume anything about the order of elements in set. An interesting feature is that a string itself is essentially a list of characters. So, if we give a string to a set, then it produce the set function, then it produces a set which consists of individual letters from this set. So, if we give this string banana to the set function, then we get the three individual letters a, n and b without duplicates in the set.

(Refer Slide Time: 02:58)

## Set operations

```
odd = set([1,3,5,7,9,11])
prime = set([2,3,5,7,11])
```

- Union  
`odd | prime → {1, 2, 3, 5, 7, 9, 11}`
- Intersection  
`odd & prime → {3, 11, 5, 7}`
- Set difference  
`odd - prime → {1, 9}`
- Exclusive or  
`odd ^ prime → {1, 2, 9}`

So, as you would expect sets support basic operations like **their** counterpart in mathematics, so suppose we set up the odd numbers to be the set of all odd numbers between 1 and 11 and the prime numbers to be the set of all prime numbers from 1 and 11 between 2 and 11 using these set function as we saw before. If we write this vertical bar, then we can get the union of the two sets.

So, odd union prime will be those elements which are either in odd or in prime. So, we get one from the top two from the bottom 3, 5, 7, 9, 11. We get all the elements in both the sets, but without any duplicates. If we ask for the intersection of two sets, we use ampersand to denote this. We get those which occur in **both** sets, those sets, those numbers which are both odd and prime and in this case 3, 5, **7 and 11**.

Notice again that the order in which these numbers are printed may be arbitrary. Set difference asks for those elements that are in odd, but not in prime. In other words, odd numbers that are not prime, in this particular collection 1 and 9 are examples of odd numbers that are not prime.

And finally, unlike union which collects elements which are in both sets, we can do an exclusive or which takes elements which are exactly in one of the two sets. If we use this

carrot symbol, then we will get 1 from the first set, 9 from the first set and 2 from the second set because 3, 5, 7, and 11 occur in both sets. So, we will not talk much more about sets, but you can use them in various contexts in order to keep track of a collection of values without duplicates using these built in operations.

(Refer Slide Time: 04:40)

## Stacks

- Stack is a last-in, first-out list
  - `push(s, x)` — add `x` to stack `s`
  - `pop(s)` — return most recently added element
- Maintain stack as list, push and pop from the right
  - `push(s, x)` is `s.append(x)`
  - `s.pop()` — Python built-in, returns last element

Let us look at different ways in which we can manipulate sequences. A list as we saw is a sequence in which we can freely insert and delete values all over the place. Now, if we impose some discipline on this, we get specialized data structures one of which is a stack. A stack is a last in first out list. So, we can only remove from a stack the element that we last added to it.

Usually this is denoted by giving two operations. When we push an element `on` to a stack, we add it to the end of the stack and when we pop a stack, we implicitly get the last value that was added. Now, this is easy to implement using built in python list. We can assume that stacks `grow` to the right. So, we push to the right and we pop from the right. So, `push s x` would just be `append x to s`.

So, you can use the built-in append function that is available `for lists to say s dot append x` when we want to push and it turns out `that` python's lists actually have a built in

function called pop which removes the last element and returns it to us. So, we just have to say s dot pop, where s is a list and we get exactly the behavior that we expect of our stack.

(Refer Slide Time: 05:57)

## Stacks

- Stacks are natural to keep track of recursive function calls
- In 8 queens, use a stack to keep track of queens added
  - Push the latest queen onto the stack
  - To backtrack, pop the last queen added

A stack is typically used to keep track of recursive function calls where we want to keep going through a sequence of functions and then, returning to the last function that was called before this. In particular when we do back tracking, we have a stack like behavior because as we add queens and remove them, what we need to do effectively is to push the latest queen onto the stack, so that when we backtrack, we can pop it and undo the last move.

(Refer Slide Time: 06:29)

## Queues

- First-in, first-out sequences
  - `addq(q, x)` — adds `x` to rear of queue `q`
  - `removeq(q)` — removes element at head of `q`
- Using Python lists, left is rear, right is front
  - `addq(q, x)` is `q.insert(0, x)`
    - `l.insert(j, x)`, insert `x` before position `j`
  - `removeq(q)` is `q.pop()`

Another disciplined way of using a list is a queue. Unlike a stack which is last in first out, a queue is a first in first out sequence. In other words, we add at one end and we remove at another end. This is exactly like a queue that you see in real life, where you join the queue at the back and when your turn comes, you are at the head of the queue and then you get served. So, `add q` will add `x` to the rear of the queue and `remove q` will remove the element which is at the head of the `q`.

Once again we can use python lists and it turns out that it is convenient to assume that a list it that represents a queue has its head at the right end rather than the rear at the left and the head at the right. This is because we can use `pop` as before, but now when we want to insert into a queue, we can use the `insert` function that is provided with this. We have not seen this explicitly, but if you have gone through the documentation, you will find it.

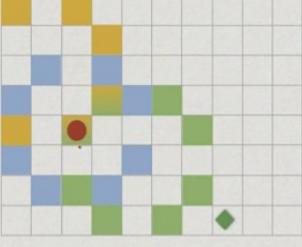
If I have a list `l` and if I insert with two arguments `j` and `x`, what it means is to put the value `j` before position `j`, put the value `x` before position `j` in particular if I insert at position 0, this has the effect of putting something before every element in the list. So, `add q q comma x` is just the same as `q dot insert 0 comma x`.

In other words, push an x to the beginning. If I have a **queue** at this form which has some values v 1, v 2 and **so on**, then this insert function will just put an x at the beginning and as we said before, the reason we have chosen to use this notation is that we can then use the pop to just remove the last element of the list. Queues and stacks can both be like easily implemented using built-in lists.

(Refer Slide Time: 08:22)

## Systematic exploration

- Rectangular m x n grid
- Chess knight starts at (sx,sy)
  - Usual knight moves
- Can it reach a target square (tx,ty)? ♦



So, one typical use of the queue is to systematically explore through **search** space. Imagine that we have a rectangular m cross n grid and we have a knight. Knight **as** a chess piece starting at a position s x comma s y. In this case, the knight is denoted by this red symbol. So, this is our knight. Now, the knight move, **if** you are familiar with chess is to move two squares up and one square left. This is a knight move.

Similarly, this is a knight move; similarly this is a knight move and so on. So, knight move consists of moving two squares in one direction, then one square across. So, these are all the positions that are reachable from this initial position, **where** the knight move there are eight of them. So, our question is that we have this red starting square and we have a green diamond indicating a target square.

Can I hop using a sequence of knight moves from the red square to the green diamond?

So, one way to do this is to just keep growing the list of squares one can reach. So, in the first step we examine these 8 squares that we can reach as we said using one move from the starting position and we mark them as squares that are available to us to reach in one step. Now, we can pick one of them for instance one of the top left and explore what we can reach from there. So, if we start at this square for instance and now we explore its neighbors, some of its neighbors are outside the grid. So, we throw them away. We keep only those neighbors inside the grid and one of them notice brings us back to the place where we started from.

Now, we could pick another square for example, we could pick this square over here and if we explore that it will again in turn produce 8 neighbors and some of these neighbors overlap the yellow neighbors. I indicate it by joint shading of yellow and green and in particular because both of them were originally **reached** from the starting point.

Of course, the starting point reaches from both of them. The starting point is both colored yellow and green. So, as you can see in the process of marking, these squares, sometimes we mark the square twice and we have to have a systematic way of making sure that we do this correctly and do not get into a loop.

(Refer Slide Time: 10:34)

## Systematic exploration

- $X_1$  — all squares reachable in one move from  $(sx, sy)$
- $X_2$  — all squares reachable from  $X_1$  in one move
- ...
- Don't explore an already marked square
- When do we stop?
  - If we reach target square
  - What if target is not reachable?

So, what we are trying to do is the following. So, in the first step we are trying to mark all squares reachable in one move from the starting point  $s_x$  comma  $s_y$ . Then, we try to mark all squares reachable from  $x_1$  in one move, call this  $x_2$ , and then we will explore all squares reachable from  $x_2$  in one move, call this  $x_3$  and soon.

Now, one of the problems is that we saw that since we could reach  $x_2$  from  $x_1$  in one move, then the squares that can reach from  $x_2$  will include squares in  $x_1$ . So, how do we ensure that we do not keep exploring an already marked square and go around and round in circles and related to this question is how do we know when to stop.

(Refer Slide Time: 11:40)

## Systematic exploration

- Maintain a queue  $Q$  of cells to be explored
- Initially  $Q$  contains only start node  $(s_x, s_y)$ 
  - Remove  $(ax, ay)$  from head of queue
  - Mark all squares reachable in one step from  $(ax, ay)$
  - Add all newly marked squares to the queue
- When the queue is empty, we have finished

Of course since we know that we are looking for the target square, if ever we marked the target square, we can stop. On the other hand, it is possible **that** the target square is not reachable. In this case, we may keep going on exploring without ever realizing that we are fruitlessly going ahead and we are never going to reach the target square. So, how do we know when to stop? So, a queue is very useful for this. What we do is we maintain at any point a **queue** of cells which remain to be explored. Initially the **queue** contains only the start node which is  $s_x$  comma  $s_y$ .

At each point we remove the head of the queue and we explore its neighbors, but when

we explore its neighbors, we mark these neighbors. Some of them may already be marked. So, we look at a x, a y, the element we remove from the head of the queue and we look at all the squares reachable at one step.

So, reachable means I can take one knight move and go there and the result of this knight move does not take me off the board. So, I mark all these squares which are reachable from a x and a y, some of which were already marked, some of which are marked just now. So, what I do is, I take the ones which I have newly marked and add them to the queue saying that these are being newly marked.

Now I need to also explore these squares for what I can reach from there. So, this guarantees that a square which has been reached once will never be reintroduced into the queue. Finally, we keep going until the queue is empty. When the queue is empty, there have been no new squares added which are unmarked before they were added. So, there is nothing more to explore and we have gone to every square we can possibly visit.

(Refer Slide Time: 13:04)

## Systematic exploration

```
def explore((sx,sy),(tx,ty)):
    marked = [[0 for i in range(n)]
              for j in range(m)]
    marked[sx][sy] = 1
    queue = [(sx,sy)]
    while queue != []:
        (ax,ay) = queue.pop()
        for (nx,ny) in neighbours((ax,ay)):
            if !marked[nx][ny]:
                marked[nx][ny] = 1
                queue.insert(0,(nx,ny))
    return(marked[tx][ty])
```

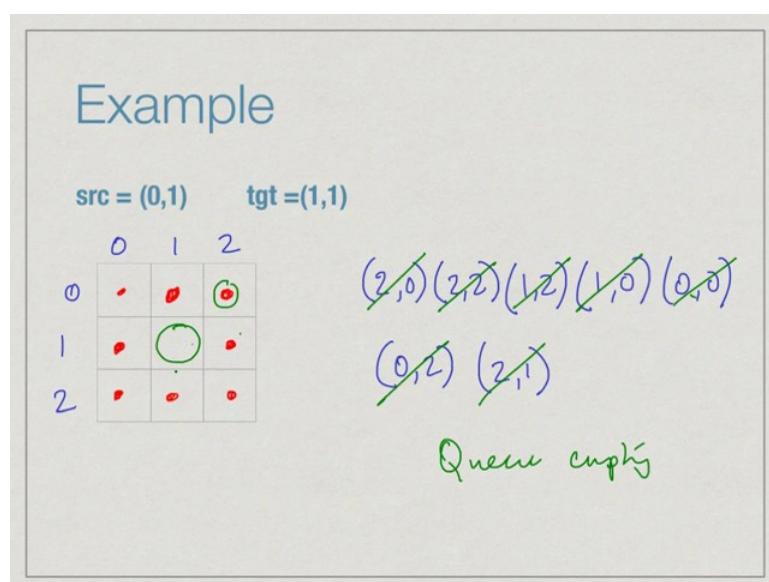
Here is some python pseudo code for this. We are going to explore from s x, s y to t x, t y. We assume that we have given to us the values m and n indicating the number of rows and columns in our grid. So, what we do is initially we set the marked array to be 0.

Remember this list comprehension notation. It says 0 for  $i$  in range  $n$  gives us a list of  $n$  zeros and we do this  $m$  times for  $j$  in range  $m$ . So, I we will get a list consisting of  $m$  blocks and each block having  $n$  zeros. This says that initially nothing is marked.

Now, we set up the thing by saying that we mark the starting node and we insert the starting node from the queue. Now, so long the queue is not empty, we pop one element from the queue. In this case  $s x, s y$  will come out. Now, there is a function which we have not written, but which will examine all the neighbors that I can reach from a  $x, y$  and give me a list of such pairs of nodes I can reach.

For each neighbor  $nx ny$  if it is not marked, then I will mark it and I will insert it into the queue, right. So, I pull out an element from the queue to explore, look at all its neighbors those which are not marked, I mark and put them back in the queue and finally, in this case I am not even going to check whether I have marked  $t x$  or  $t y$  in the middle. I know that if I have a finite set of squares at some point, this process has to stop. At the end I will return whether I will return the value of marked at the target node  $t x, t y$ . So, if I have reached it, this will return 1 which is true. If it is not reached, it will return 0 which is false.

(Refer Slide Time: 14:53)



Let us look at an example of how this works. So, here we have a three by three grid. Remember that the cells are 1 0 1 2 and 0 1 2. **by our** naming convention we want to start from the top center square. This is our source and here in the center is our target. So, let us erase all these marks and set up this thing as we expect. So, we say that initially the queue that we want to have as a source node and we mark the source node in the grid. The marking is indicated by a red mark. So, this is how we start.

So, our first step is to remove this from the queue and explore its neighbors. Now, its neighbors are 2, 0, and 2. This means we will henceforth remove these brackets because it is more annoying. So, you just grow it like this. So, we say that my queue consists of 2 comma 0 and 2 comma 2. This is my **queue of vertices way to be explored**. At each step I will now remove the first element of the queue and explore its neighbors. When I explore the neighbors of 2 0, I will find one of them is of course is where I start it from. I only look at unmarked neighbors. So, an unmarked neighbor is 1 2. I will add that back at the end of the queue.

Now, proceeding I will take the next element of the queue which is 2 2 and look at its neighbors. So, 2 2 can go back again to the original thing and it also has a new thing here which is 1 0. So, continuing like this I remove 1 2 which is this one and then, look at its neighbors. So, one of its neighbors is 2 0, but one of them is 0 0. So, I get a new neighbor 0 0 here and then, I continue by taking 1 0 of the queue. So, 1 0 is this one. So, it has one new neighbor unexplored which is that one. So, my queue now has 0 0 followed by 0 2. Then, when I explore 0 0, I get this neighbor at the bottom which is 2 1.

Now, when I remove 0 2 which is this one, I find that both these neighbors I explored. So, I add nothing. I continue with 2 1. Again, I find both its neighbors explored and do nothing. Now at this point, the queue is empty and since the queue is empty, I stop and I find that my square of interest namely 1 1 was not marked. Therefore, in this case the target is not reachable from the source node.

(Refer Slide Time: 17:35)

## Example

src = (0,1)      tgt =(1,1)



- This is an example of breadth first search

This is actually an example of breadth first search which you will study if you look at graphs, but it just illustrates that a queue is a nice way to systematically keep track of how you explore through a search space.

(Refer Slide Time: 17:47)

## Summary

- Data structures are ways of organising information that allow efficient processing in certain contexts
- Python has a built-in implementation of sets
- Stacks are useful to keep track of recursive computations
- Queues are useful for breadth-first exploration

To summarize data structures are ways of organizing information that allow efficient

processing in certain contexts. So, we saw that python has a built-in implementation of lists of sets rather we also saw that we can take sequences and use them in two structured ways. So, stack is a last-in first-out list and we can use this to keep track of recursive computations and queues are first-in first-out list that are useful for breadth first exploration.

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

**Week - 06**  
**Lecture - 05**  
**Priority queues and heaps**

(Refer Slide Time: 00:02)

### Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities.
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it.
- New jobs may join the list at any time.
- **How should the scheduler maintain the list of pending jobs and their priorities?**

Let us look at a data structure problem involving job schedulers. Job scheduler maintains a list of pending jobs with priorities.

Now, the job scheduler has to choose the next job to execute at any point. So, whenever the processor is free it picks the job, not the job which arrived earliest, but the one with maximum priority in the list and then schedules it. New jobs keep joining the list, each with its own priority and according to their priority they get promoted ahead of other jobs which may have joined earlier. So, our question is how should the scheduler maintain the list of pending jobs and their priorities? So that it can always pull out very quickly the one with the highest priority to schedule next.

(Refer Slide Time: 00:47)

## Priority queue

- Need to maintain a list of jobs with priorities to optimise the following operations
  - **delete\_max()**
    - Identify and remove job with highest priority
    - Need not be unique
  - **insert( )**
    - Add a new job to the list

This is like a queue, but a queue in which items have priority based on some other characteristic not on when they arrived. So, we saw a normal queue is a first-in-first-out object, the ones that arrive first leave first. In a priority queue, they leave according to their priority. There are two operations associated with the priority queue, one is delete max. In delete queue we just said we take the element at the head of the queue. In delete max we have to look through the queue and identify and remove the job with the highest priority.

Note of course, that the priorities of two different jobs may be the same in which case we can pick any one and the other operation is which we normally called add to the queue we will call insert and insert just adds a new job to the list and each job when it is added comes with its own priority.

(Refer Slide Time: 01:39)

## Linear structures

- Unsorted list
  - `insert()` takes  $O(1)$  time
  - `delete_max()` takes  $O(n)$  time
- Sorted list
  - `delete_max()` takes  $O(1)$  time
  - `insert()` takes  $O(n)$  time
- Processing a sequence of  $n$  jobs requires  $O(n^2)$  time

Based on linear structures that we already studied, we can think of maintaining these jobs just as a list. Now, if it is an unsorted list when we add something to the queue we can just add **it** to the list append it in any position. This takes constant **time**; however, to do a delete max we have to scan through the list and search for the maximum element and as we have seen in an unsorted list, it will take us order  $n$  time to find the maximum element in list because we have to scan all the items.

The other option is to keep the list sorted. This helps to delete max, for instance, if we keep it sorted in descending order the first element of the list is always the largest element. However, the price we pay is for inserting because to maintain the sorted order when we insert the element we have to put it in the right position and as we saw in insertion sort, insert will take linear **time**. So, as a trade-off we either take linear time for delete max or linear time for insert. If we think of  $n$  jobs entering and leaving the queue one way or another we end up spending  $n$  squared time processing these  $n$  jobs.

(Refer Slide Time: 02:55)

## Binary tree

- Two dimensional structure
- At each node
  - Value
  - Link to parent, left child, right child

```
graph TD; 5((5)) --- 2((2)); 5 --- 8((8)); 2 --- 1((1)); 2 --- 4((4)); 8 --- 9((9)); classDef = "parent"; 2; classDef = "left child"; 1; classDef = "right child"; 4; classDef = "parent"; 8; classDef = "leaf"; 9;
```

This is the fundamental limitation of keeping the data in a one dimensional structure.

Let us look at two dimensional structures; the most basic two dimensional structure that we can think of is a binary tree. A binary tree consists of nodes and each node has a value stored in it and it has possibly a left and a right child. We start at the root which is the top of the tree and then each node will have 1 or 2 children. A node which has no children is called a leaf and then with respect to a child we call the parent node, the node above it and we refer to the children as the left child and the right child.

(Refer Slide Time: 03:40)

## Priority queues as trees

- Maintain a special kind of binary tree called a **heap**
  - **Balanced:** N node tree has height  $\log N$
  - Both `insert()` and `delete_max()` take  $O(\log N)$
  - Processing N jobs takes time  $O(N \log N)$
  - Truly flexible, need not fix upper bound for N in advance

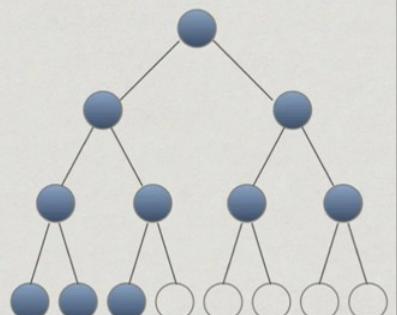
So, our goal **is** to maintain a priority queue as a special kind of binary tree which we will call a heap. This tree will be balanced. A balanced tree is one in which roughly speaking at each point the left and right sides are almost the same size. Because of this it turns out that in a balanced tree, if we have  $n$  nodes then the height of the tree will be logarithmic because remember that the height doubles with each level and as a result of which we can fit  $n$  nodes in  $\log n$  levels provided it is balanced and because it is of height  $\log n$ , we will achieve both insert and delete max in order  $\log n$  time.

This means that if we have to add and remove  $n$  elements from the queue, overall we will go from  $n$  squared to  $n \log n$  and another nice feature about a heap is that we do not have to fix  $n$  in advance this will work as the heap grows and shrinks so we do not need to know what  $n$  is.

(Refer Slide Time: 04:44)

## Heaps

- Binary tree filled level by level, left to right
- At each node, value stored is bigger than both children
- (Max) Heap Property  
Binary tree filled level by level, left to right

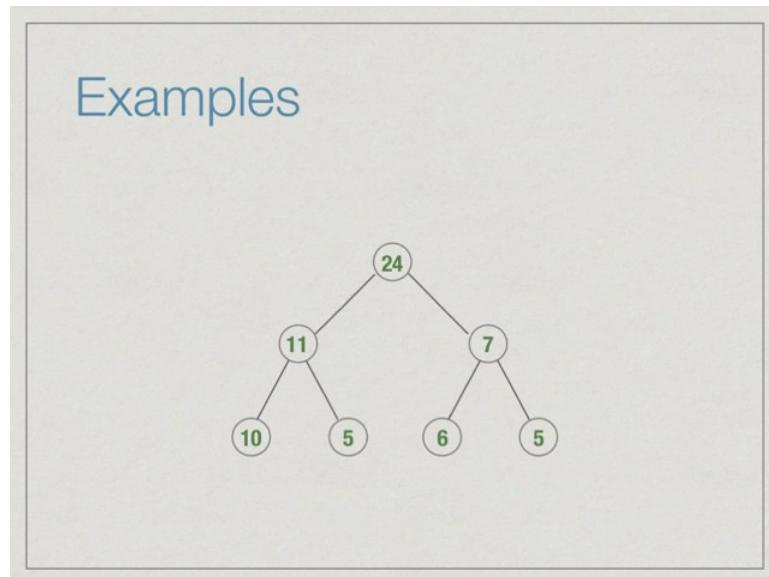


What is a heap? Heap is a binary tree with two properties, the first property is structural: which are the values which are stored in the tree, remember that leaves, nodes in a binary tree may have 0 children, 1 children or 2 children. So, we could have in general a very uneven structure. Heaps have a very regular structure when we have a heap we have a binary tree in which we fill each level from top to bottom, left to right.

In other words, at any point a heap consists of a number of filled levels and then in the bottom level we have nodes filled from left to right and then possibly some unfilled nodes. The other property with the heap, the first property is **structural**, it just tells us how the values look in the heap. In this node, for example, in this picture the blue nodes are those which have values and the empty nodes are indicated with open circles at the bottom.

The second property about the heap is the values themselves. So, the heap property in this case what we call the max heap property because we are interested in maximum values says that every value is bigger than the values of its 2 children. So, at every node if you look at the value and we look at the value in the left child and the right child then the parent will have a larger value than the both the children.

(Refer Slide Time: 06:06)



Let us see some examples. Here is a four node heap, because it has four nodes we fill the root in the first level and finally, in the second level we have only one node which is a left most and notice that the values are correctly ordered, 24 is bigger than 11 and 7, 11 is bigger than its only child 10, 7 has no children. So, there are no constraints.

Here is another example where the bottom level is actually full, here we have 7 nodes and once again at every point we see that 11 is bigger than 10 and 5, 7 is bigger than 6 and 5. So, we have the value property - the max heap property along with the structural property.

(Refer Slide Time: 06:42)

## Non-examples

- No “holes” allowed

```
graph TD; 24((24)) --- 11((11)); 24 --- 7((7)); 11 --- 10((10)); 11 --- 5((5)); 7 --- 5((5));
```

Here is an example of something which is structurally not a heap because it is a heap we should have it filled from top to bottom, left to right. So, we should have a node here to the left of 7 before we add a right child therefore, this is not a heap.

(Refer Slide Time: 06:59)

## Non-examples

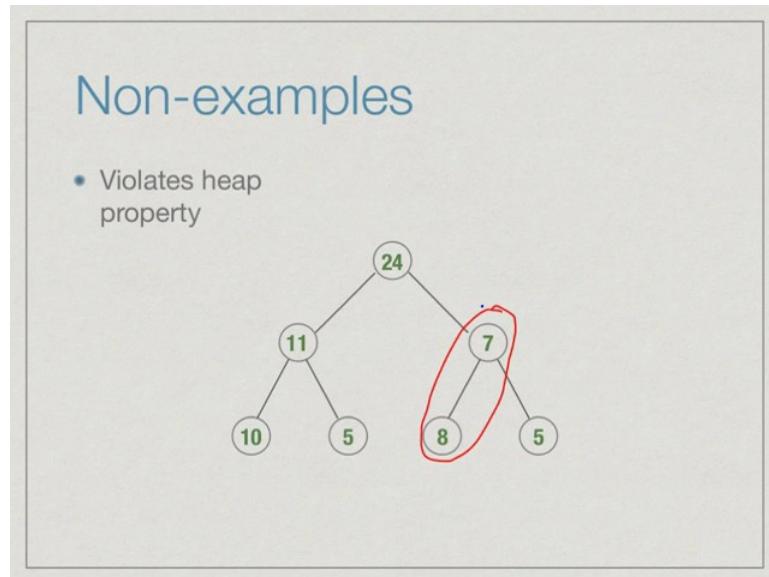
- Can't leave a level incomplete

```
graph TD; 24((24)) --- 11((11)); 24 --- 7((7)); 11 --- 10((10)); 11 --- 5((5)); 7 --- 6((6)); 7 --- 8((8));
```

Similarly, here the node 8 could not have been added here before we filled in the right

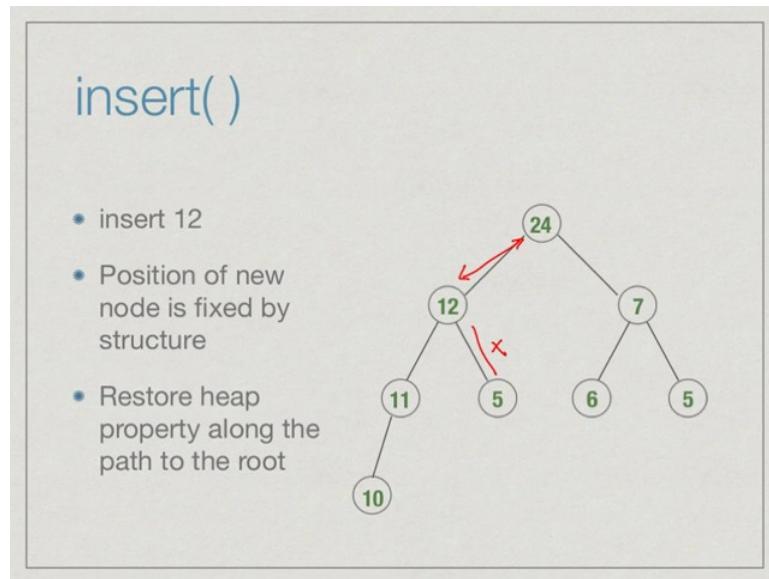
child of 7. So, once again this has not been filled correctly left to right, top to bottom and therefore, this is not a heap.

(Refer Slide Time: 07:14)



This particular tree satisfies the structural property of a heap in the sense that it is filled from top to bottom, but we have here a violation of the heap property because 7 has a child which has a larger value than it namely 8.

(Refer Slide Time: 07:32)

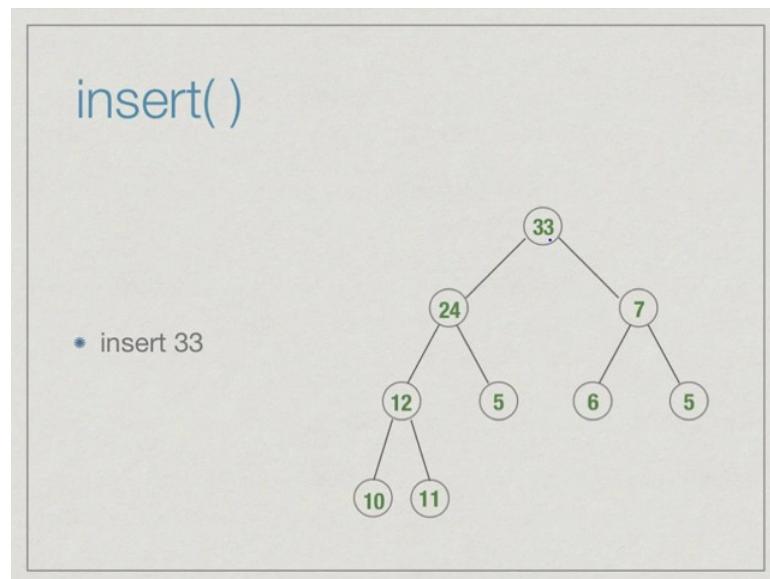


Our first job is to insert a value into a heap while maintaining the heap property. So, the first thing to note is that we have no choice about where to put the new node, remember that heap nodes are constructed top to bottom, left to right. If we want to insert 12 it must come below the 10 to the left because we have to start a new level, since the previous level is full. The problem is that this may not satisfy the heap property; in this case 12 is bigger than its parent 10.

Although this is now structurally correct, it does not have the right value distribution. So, we have to restore the heap property in some way. This is what we do we first create the node, we put the new value into that node and then we start looking at violations with respect to its parent. We notice that 12 and 10 are not correctly ordered. So, we exchange them, right now this is a new node. We have to check whether it is correctly ordered with respect to its current parent. So, we look and we find that it is not. So, again we exchange these.

Now, notice that because 11 was already bigger than 5, 12 will remain bigger than 5. There is no need to check anything down from where we got, we only have to look up. Now, we have to check whether there is still a problem above. In this case, there is no problem 12 is smaller than 24. So, we stop.

(Refer Slide Time: 08:59)



Let us add another node. Supposing, we add 33 now to the heap that we just created. So, 33 again creates a new node at this point. Now, 33 being bigger than 11 we have to walk up and swap it then again we compare 33 and its parent 12 and we notice that 33 is bigger than 12. So, we swap it again then we look at the root, in this case 24 and we find that 33 is bigger than 24. So, we swap it again and now 33 has no parents and it is definitely bigger than it is 2 children. So, we can stop.

(Refer Slide Time: 09:35)

## Complexity of insert( )

- Need to walk up from the leaf to the root
  - Height of the tree
- Number of nodes at level 0,1,...,i is  $2^0, 2^1, \dots, 2^i$
- K levels filled :  $2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$  nodes
- N nodes : number of levels at most  $\log N + 1$
- insert( ) takes time  $O(\log N)$

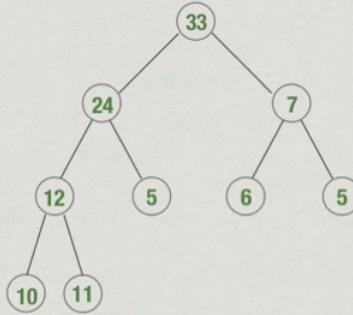
How much time does insert take? In each time we insert a node, we have to check with its parent, swap, check with its parent, swap and so on, but the good thing is we only walk up a path we never walk down a path. So, the number of steps you walk up will be bounded by the height of the tree.

Now, we argued before or we mentioned before that a balanced tree will have height  $\log n$ . So, we can actually measure it correctly by saying that the number of nodes at level i is  $2$  to the  $i$ . Initially, we have 1 node  $2$  to the  $0$ , then at the first level we have 2 nodes  $2$  to the  $1$  and second level we have 4 nodes  $2$  to the  $2$  and so on. If we do it this way then we find that when  $k$  levels are filled, we will have  $2$  to the  $k$  minus 1 nodes and therefore, turning this around we will find that if we have  $n$  nodes then the number of levels must be  $\log n$ . Therefore, insert walks up a path, the path is equal to the height of the tree, and the height of the tree is order of  $\log n$ . So, insert takes time order  $\log n$ .

(Refer Slide Time: 10:42)

## delete\_max( )

- Maximum value is always at the root
  - From heap property, by induction
- How do we remove this value efficiently?

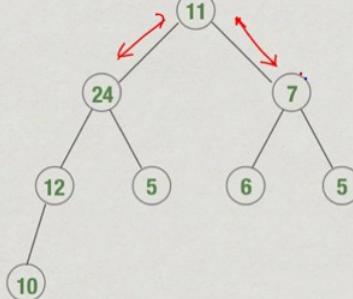


The other operation we need to implement in a heap is delete max. Now, one thing about a heap is that the maximum value is always at the root this is because of the heap property you can inductively see that because each node is bigger than its children the maximum value in the entire tree must be at the root. So, we know where the root is; now the question is how do we remove it efficiently?

(Refer Slide Time: 11:10)

## delete\_max( )

- Removing maximum value creates a “hole” at the root
- Reducing one value requires deleting last node
- Move “homeless” value to root

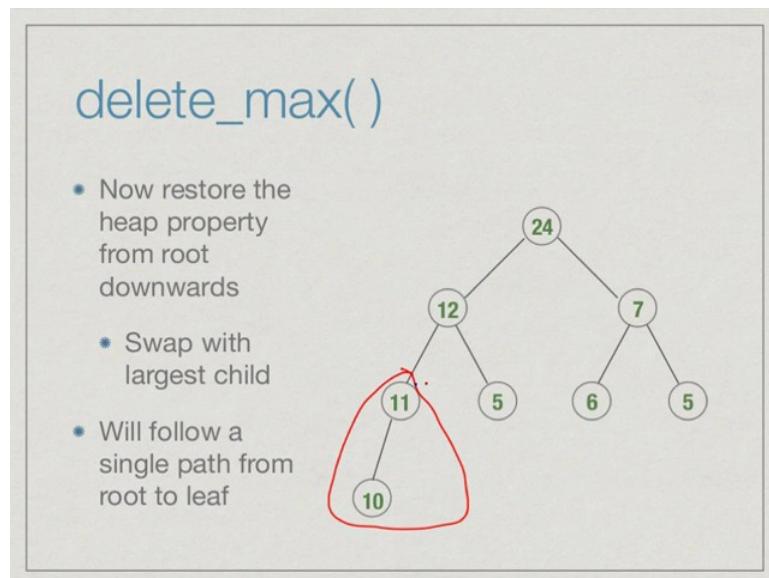


If we remove this node, first of all we cannot remove the node because it is a root. If you remove this value then we have to put some value there. On the other hand, the number of values in the node in the heap has now shrunk. So, this node at the bottom right must be deleted because the structural property of the heap says that we must fill the tree left to right, top to bottom. We are going top to bottom and we have run out of a value.

The last node that we added was the one at the right most end of the bottom row and that must go. So, we have a value which is missing at the top and we have a value at the bottom namely 11 whose node is going to be deleted. So, the strategy now is to move this value to 11 and then fix things, right. So, we first remove the 33 from the root, we remove the node containing 11 and we move the 11 to the position of the root.

Now, the problem with this is we have moved an arbitrary value not the maximum value to the top. Obviously, there is going to be some problem with respect to its children. So, here it turns out that 11 is bigger than 7 which is correct, but unfortunately it is smaller than 24.

(Refer Slide Time: 12:19)

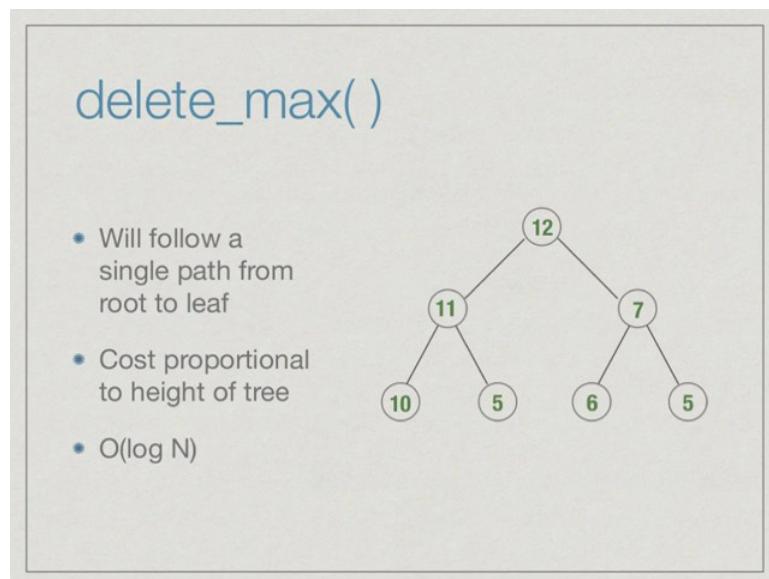


To restore the property what we do is, we look at both directions right and we exchange it with the largest child. Suppose, this had been 17 here then we could swapped 11 with

17 here or 11 with 24, both violations are there, but if you move this smaller child up then 17 will not be bigger than 24, so we move the largest one.

In this case we move 24 up right and now we have 11 again and now we have to again check whether it is correct with respect to its 2 children, again it is not. So, we move the largest one up namely 12 and then we see now whether it is correct with respect to it's children. At this point 11 is bigger than 10. So, we stop.

(Refer Slide Time: 13:03)



Just as insert followed a single path from the new node at the leaf up to the root, delete max will follow a single path from the root down to a leaf.

Once again the cost of delete max will be proportional to the height of the tree which as we said earlier is  $\log n$ . Let us do another delete; we delete, in this case 24, now we remove the node for 10, 10 goes to the root. We compare 10 with it's 2 children, 12 and 7 and find that it is not satisfying heap property. So, we move the larger of the two up namely 12. Now, we look at its children, new children here 11 and 5 and again we see it is not satisfying the property. So, the larger one moves up and once it reaches the leaf there are no properties to be satisfied anymore. So, we stop.

(Refer Slide Time: 13:56)

## Implementing using arrays

- Number the nodes left to right, level by level
- Represent as an array  $H[0..N-1]$
- Children of  $H[i]$  are at  $H[2i+1], H[2i+2]$
- Parent of  $H[j]$  is at  $H[\lfloor(j-1)/2\rfloor]$  for  $j > 0$

```
graph TD; 0((0)) --> 1((1)); 0((0)) --> 2((2)); 1((1)) --> 3((3)); 1((1)) --> 4((4)); 2((2)) --> 5((5)); 2((2)) --> 6((6)); 3((3)) --> 7((7)); 3((3)) --> 8((8)); 4((4)) --> 9((9)); 4((4)) --> 10((10)); 5((5)) --> 11((11)); 5((5)) --> 12((12)); 6((6)) --> 13((13)); 6((6)) --> 14((14));
```

One very attractive feature of heaps is that we can implement this tree directly in a list or in an array. So, we have **an**  $n$  node heap, we can **represent** it as a list or an array with position 0 to  $n$  minus 1. The position 0 represents a root then in order 1 and 2 represent the children, then 3, 4, 5, 6, 7 nodes are the next level and so on. So, just as we said we filled up this heap left to right, top to bottom right. In the same way, we number the nodes also top to bottom, left to right. So, we start with 0 at the root, then 1 on the left, 2 on the right then 3, 4, 5, 6, 7, 8, 9, 10 and so on.

From this you can see that, if I have a position labeled  $i$  then the two children are read  $2i + 1$  and  $2i + 2$ . So, the children of 1 are 2 into 1 plus 1, which is 3 and 2 into 1 plus 2, which is 4. Similarly, children of 5 are 2 into 5 plus 1, which is 11 and 2 into 5 plus 2 which is 12. So, just by doing index calculations for a position in the heap we can figure out where **its** children are and by reversing this calculation we can also find the index of the parent, the parent of  $j$  is that  $j - 1$  by 2. Now,  $j - 1$  by 2 may not be an integer. So, we take the floor.

If we take 11, for example,  $11 - 1$  is 10, 10 by 2 is 5. If we take 14, for example,  $14 - 1$  is 13, 13 by 2 is 6.5 we take the floor and we get 6.

(Refer Slide Time: 15:32)

## Building a heap, heapify( )

- Given a list of values  $[x_1, x_2, \dots, x_N]$ , build a heap
- Naive strategy
  - Start with an empty heap
  - Insert each  $x_i$
  - Overall  $O(N \log N)$

This allows us to manipulate parent and children nodes by just doing index arithmetic we go from  $i$  to  $2i + 1$ ,  $2i + 2$  to go to the children and we go from  $j$  to  $j - 1$  by 2 floor to go to the parent. How do we build a heap. A naive way to build a heap is just to take a list of values and insert them one by one using the heap operation into the heap.

So, we start with an empty heap, we insert  $x_1$ , create a new heap containing  $x_1$ , we insert  $x_2$ , creating a heap of  $x_1, x_2$  and so on. Each operation takes  $\log n$  time of course,  $n$  will be growing, but it does not matter if we take the final  $n$  as an upper bound we do  $n$  inserts each just  $\log n$  and we can build this heap in order  $n \log n$  time.

(Refer Slide Time: 16:23)

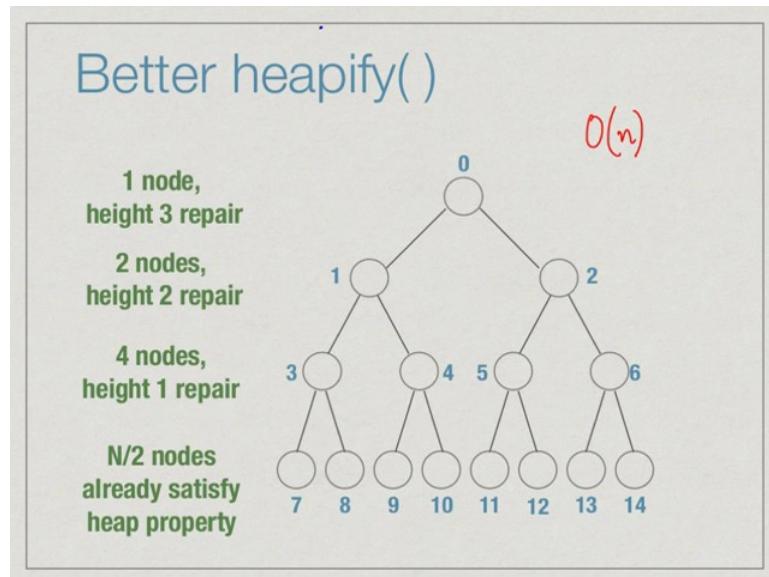
## Better heapify( )

- Set up the array as  $[x_1, x_2, \dots, x_N]$ 
  - Leaf nodes trivially satisfy heap property
  - Second half of array is already a valid heap
- Assume leaf nodes are at level k
  - For each node at level k-1, k-2, ..., 0, fix heap property
  - As we go up, the number of steps per node goes up by 1, but the number of nodes per level is halved
  - Cost turns out to be O(N) overall

There is a better way to do this heap building if we have the array as  $x_1$  to  $x_n$  then the last half of the nodes correspond to the leaves of the tree. Now, a leaf node has no properties to satisfy because it has no children. We do not need to do anything we can just leave the leaves as they are.

We go one level above and then we can fix all heap errors at one level above right and then again we move one level above and so on. So, we do the kind of top to bottom heap fixing that we did with the delete max, while we are building the heap. So, as we are going up the number of steps that we need to propagate this error goes higher and higher because we need to start at a higher point on the other hand the number of nodes for which this happens is smaller.

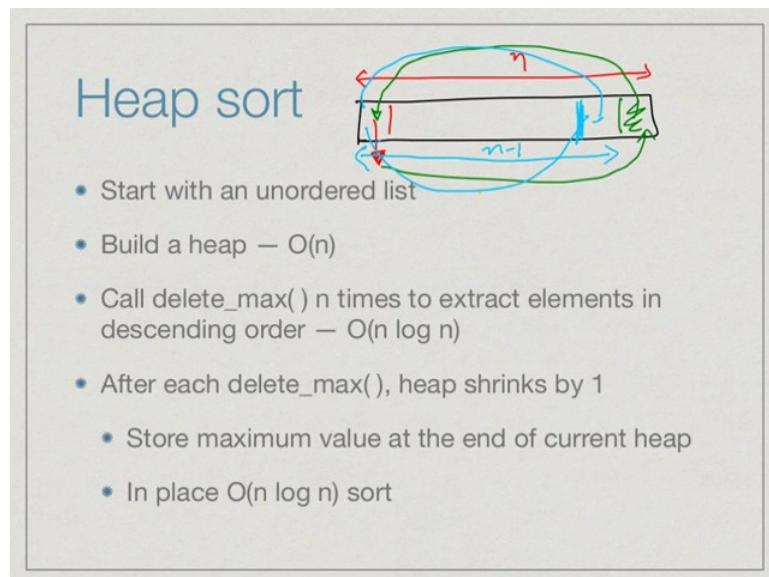
(Refer Slide Time: 17:19)



Let us look at this here. What we are saying is that if we start with the original list of say elements 0 to 14, then the numbers 7 to 14 already satisfy the heap property. Whatever values there are, we do not need to worry, then we go up and we may have to swap this with its children, we may have to swap this with its children and so on. For 4 nodes we have to do one level of shifting perhaps to repair the heap property then we go up now 1 and 2 are the original values, they may be wrong. So, again we may have to shift it down one value and then another value.

Now, we need two levels of shifting, but we have only two nodes for this. The number of nodes for which this is required is shrinking, it's halving actually and the number of steps for which we have to do it is increasing by 1. We will not do a careful calculation here, but it turns out that as a result of this, in this particular way of doing the heapify by starting from the bottom of the heap and working upwards rather than inserting one at a time into an empty heap actually takes us only linear time order n.

(Refer Slide Time: 18:30)



A final use of heap is to actually sort, we are taking out one element at a time starting with maximum one. It is natural that if we start with a list, build a heap and then do  $n$  times delete max we will get the list of values in descending order. We build a heap in order  $n$  time, call delete max  $n$  times and extract the elements in descending order. So, we get an order  $n \log n$  algorithm for heap.

Now, the question is where do we keep these values. Well, remember that a heap is an array. Initially, we have a heap which has  $n$  elements. So, we build this heap now we said that the delete max will remove the element at the top because that is the root, but it will also create a vacancy here, this is the value that will go to the top, this is the last leaf which will go the top when we fix the delete max.

Since there is a vacancy here we can actually move this to this position, the maximum value will now go to the end of the heap, but the next time we process the heap there will be only  $n$  minus one values. So, we will not look at that value we will just use from 0 to  $n$  minus 2. Again this value will come here this value will go there and now we will have two elements fixed and so on. So, one by one the maximum value, second maximum value and so on will get into the same list or array in which we are storing the heap and eventually the heap will come out in ascending order.

This is actually an  $n \log n$  sort. It has same as asymptotic complexity as merge sort we saw before and unlike merge sort which forced us to create a new array everytime we merge two lists, this is actually creating a list in place.

(Refer Slide Time: 20:20)

## Summary

- Heaps are a tree implementation of priority queues
  - insert() and delete\_max() are both  $O(\log N)$
  - heapify() builds a heap in  $O(N)$
  - Tree can be manipulated easily using an array
  - Can invert the heap condition
    - Each node is smaller than its children
    - Min-heap, for insert(), delete\_min()

To summarize heaps are a tree based implementation of priority queues in which both insert and delete max can be done in  $\log n$  time. We can do a bottom up heapify to build a heap in  $n$  time and these are trees, but they can be manipulated very easily using an array. Now, in this case we were looking at max heaps; we can also do a dual construction where we change the heap condition to say that each element must be smaller than its children, in which case we have what is called a min-heap and then instead of delete max, the operation we perform is delete min. Everything is exactly symmetric to the other case.

When we move something up we have to move it down according to the min condition and when we insert something at the bottom we have to move it up right. So, the insert and delete min work exactly like insert and delete max, except that the comparisons are reversed because we now have a min condition rather than the max condition locally and finally, we saw that with a heap we can do sorting in  $n \log n$  time in place.

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

**Week - 07**  
**Lecture - 01**  
**Abstract Datatypes, Classes and Objects**

(Refer Slide Time: 00:02)

## Data structures

- Behaviour defined through **interface**
  - Allowed set of operations
- Stack: `push()` and `pop()`
- Queue: `addq()` and `removeq()`
- Heap: `insert()` and `delete_max()`
  - Heap implemented as a list `h`, does not mean `h.append(7)` is legal

We have seen how to implement data structures such as, stacks, queues and heaps using the built in list type of Python. It turns out that one can go beyond the built in types in python and create our own data types. So, we will look at this in more detail in **this** weeks' lectures.

Let us revisit what we lean by a data structure. A data structure is basically an organization of information whose behavior is defined through an interface. So an interface is nothing but the allowed set of operations, for instances for a stack **the** allowed set of operations are push and pop. And of course, we can also query whether a stack is empty or not.

Likewise, for a queue the only way we can modify a queue is to add something to the tail of the queue using the function `add q` and remove the element at the head of the queue

using the function remove q. And for a max heap for instance, we have the functions insert to add **an element** and delete max which removes the largest element from the heap.

Now, just because we implement a heap as a list it does not mean that the functions that are defined for lists are actually legal for the heap. So if we have a heap h, which is implemented as a python list though the list will allow an append function. The append function on its own does not insert a value and maintain the heap property. So, in general the call such as h dot append 7 would not be legal.

(Refer Slide Time: 01:35)

## Abstract datatype

- Define behaviour in terms of operations
  - `(s.push(v)).pop() == v`
  - `((q.addq(u)).addq(v)).removeq() == u`
- No reference to implementation details
- Implementation can be optimized without affecting functionality

So, we want to define new abstract data types in terms of the operations allowed. We do not want to look at the implementation and ask whether it is a list or not, because we do not want the implementation to determine what is allowed, we only want the actual operations that we define as the abstract interface to be permitted.

For instance if we have a stack s and we push value v then the property of a stack guarantees that if we immediately apply pop the value we get back is our last value push and therefore we should get back v. In other words, if we execute this sequence we first to s dot push and then we do a pop then the value that we pushed must be the value that

we get back.

This is a way of abstractly defining the property of a stack and how push and pop interact without actually telling us anything about how the internal values are represented. In the same way if we have an empty queue and we add to it two elements u and v and then we remove the head of the queue, then we expect that we started with an empty queue and then we put in from this end u and then we put in a v, then the element that comes out should be the first element namely u. In other words assuming that this is empty then if we add u and add v and then remove the head we should get back first element that we put in namely u.

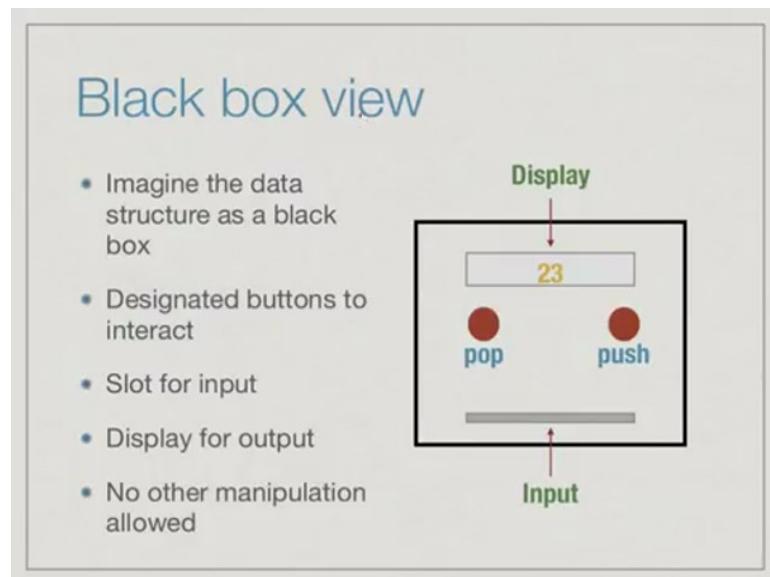
The important thing is that we would like to define the behavior of a data type without reference to the implementation. Now this can be very tedious because you have to make sure that it captures all the properties between functions, but this can be done and this is technically how an abstract data type is defined. Now for large purposes we will normally define it more informally and we will make reference to the implementation, but we definitely do not want the implementation to determine how these functions work.

In other words, we should be able to change one implementation to another one such that the functions behave the same way and the outside user has no idea which implementation is used. Now this is often the case when we need to optimize implementation, we might come up with an inefficient implementation and then optimize it. For instance we saw that for a priority queue we could actually implement it as a sorted list and then we could implement insert as an insert operation in a sorted list which you take order  $n$  time, but delete max would just remove the head of the list.

This is not optimal because over a sequence of  $n$  inserts and deletes this takes time order  $n$  square. So if we replace the internal implementation from a sorted list to a heap we get better behavior, but in terms of the actual values that the user sees as a sequence of inserts and delete max the user does not see any difference between the sorted list implementation and the heap implementation. Perhaps, there is a perception that the one is faster than the other, but the actual correctness of the implementation should not be affected by how you choose to represent the data. So, this is the essence of defining an

Abstract datatype.

(Refer Slide Time: 04:40)



So, good way to think of an abstract datatype is as a black box which allows limited interaction. Imagine something like an ATM machine. So, we have the data structure as a black box and we have certain buttons which are the public interface, these are the functions that we are allowed to use. In this picture imagine this is a stack and the buttons were allowed to push are pop and push let they are allowed to remove the top elements from the stack; they are allowed to put an element into the stack.

Now this requires us to also add and view things from the stack, so we also have a slot for input which is shown as a kind of a thing at the bottom here we have the slot for input. And we have the way to receive information about the state of the stack. So we can imagine that we have some kind of a display.

This is typically how we would like to think of a data structure, we do not want to know what is inside the black box we just want to specify that if we do a sequence of button pushes and we start supplying input through the input box what do we expect to see in the display. Other than this, no other manipulation should be allowed. We are not allowed to exploit what is inside the box in order to quickly get access say to the middle of a stack.

or the middle of a queue. So we do not want such operations, we only want those operations which the externally visible interface or the buttons in this case of the black box picture allow us to use.

(Refer Slide Time: 06:21)

## Built in datatypes

```
l = []
```

- List operations `l.append()`, `l.extend()` permitted
  - ... but not dictionary operations like `l.keys()`
- Likewise, after `d = {}`, `d.values()` is OK
  - ... but not `d.append()`
- Can we do this for stacks, queues, heaps, ...?

In a sense this is already implemented when we use the built in data types of python, if we announce that the name `l` is of type list by setting `l` to the empty list then immediately python will allow us to use operations like append and extend on this list, but because it is of list type and not dictionary type we would not be able to execute an operations such as `keys` which is defined for dictionaries are not list.

Likewise if we define `d` to be an empty dictionary then we can use a function such as `d` dot `values` to get the list of values currently stored, but we cannot manipulate `d` as a list. So, we cannot say `d` dot `append` it will give us an error. Python uses the type information that it has about the value give assign to a name to determine what functions are legal which is exactly what we are trying to do with these abstract data types. We are trying to say that the data type on its own should allow only certain limited types of access whose behavior is specified without telling us anything about the internal implementation.

Remember for instance we saw that in a dictionary even if we add a sequence or values in the particular order we ask for the values after sometime they may not written in the same order, because internally there is some optimization in order to make it fast to look up a value for it. We have no idea actually how dictionaries implemented inside, but what we do know is that if we provide a key and that key is a valid key we will get the associated with that key, we do not ask how this is done and we do not know whether from one version of python to the next the way in which this is implemented changes.

Our question is, that instead of using the built in list for stacks, queues and heaps and other data structures can we also defined a data type in which certain operations are permitted according to the type that we start with.

(Refer Slide Time: 08:17)

## Object Oriented programming

- Data type definition with
  - Public interface
    - Operations allowed on the data
  - Private implementation
    - Match the specification of the interface

This is one of the main things which are associated with a style of programming called Object Oriented program. In object oriented program, we can provide data type definitions for new data types in which we do precisely what we have been talking about we describe the public interface, that is the operations that are allowed on the data and separately we provide an implementation which should be private, we will discuss later that in python we do not actually have a full notion of privacy because of the nature of the language.

But ideally the implementation should not be visible outside only the interface should allow the user to interact with the implemented data. Of course, the implementation must be such that the functions that are visible to the user behave correctly.

So here for instance if we had a heap the public interface would say insert and delete max, the private implementation may be a sorted list or it may be a heap and then we would then have to ensure that if we are using a sorted list we implement delete max and insert in the correct way and if we switch from that to a heap the priority queue operations remain the same.

(Refer Slide Time: 09:33)

## Classes and objects

```
class Heap: Constructor
    def __init__(self,l):
        # Create heap
        # from list l
        # Create object,
        # calls __init__()
        l = [14,32,15]
        h = Heap(l)

    def insert(self,x):
        # insert x into heap
        # Apply operation
        h.insert(17)
        insert(h,17)
        h.insert(28)

    def delete_max(self,x):
        # return max element
        v = h.delete_max()
```

In the terminology of object oriented programming there are two important concepts; Classes and Objects. A class is a template very much like a function definition is a template, when we say def and define a function the function does not execute it just gives us a blue print saying that this is what would happen if this function were called with a particular argument and that argument to be substituted for the formal parameter in the function and the code in the function will be executed to the corresponding value.

In the same way a class sets up a blue print or a template for a data type. It tells us two things it tells us; what is the internal implementation? How is data stored? And it gives

us the functions that are used to implement the actual public interface. So, how you manipulate the internal data in order to effect the operations that the public interface allows. Now once we have this template we can construct many instances of it. So, you have the blue print for a stack you can construct many independent stacks, each independent stack has its own data that stacks do not interfere with each other.

Each of them has a copy of the function that we have defined associated with it. Rather than the kind of the main difference from classical programming is, in classical programming you would have for instance a function like say push define and it will have two parameters typically a stack and a value. So, you have one function and then you provide it the stack that you want to manipulate.

On the other hand, now we have several stacks s1, s2, s3, etcetera which are created as instance as class, and logically each of them has its own push function. So there is a push associated with s1, that the push associated is s2, the push associated with s3 and so on. Each of them is a copy of the same function derived from this template, but this implicitly attach to the single object. So, this is just a slight difference in perspective instead of having a function to which you pass the object that you want to manipulate you have the object and you tell it what function to apply to itself.

So, let us look at a kind of example this would not be a detailed example it will just give you a flavor of what we are talking about. Here is a skeleton of a definition of a class heap. So now, we instead of using the built in list we want to define our own data type heap. So there are some function definitions. These def statements and these correspond to definition in the functions and what we will see is that inside these definitions we will have values which are stored in each copy of the heap. So, just to get a little bit of an idea about how this is would work.

When we create an object of type heap we call it like a function. So, we say h is equal to heap 1, so this implicitly says give me an instance of the class heap with the initially value I passed to it now this calls this function `init` which is why it is called `init`. So, `init` is what is called a constructor. A constructor is a function that is called when the object is created and sets it up initially in this particular case our constructor is presumed to take

an arbitrary list of values say 14, 32, 15 and heapify it. So, somewhere inside the code of `init` there will be a heapification operation which we are not actually shown in this slide.

This is how you create objects. You first define a class we will look at a complete example soon, we define a class and then you call the class sort of like a function and the name that is attached to this function call or this class becomes a new object. As we said we have functions like insert and delete max define for heaps, but it is like we have the separate copy of this function for each individual heaps.

In this case we are created a heap `h`, so we want to tell `h` `insert in` yourself the value 70. So, we write that as `insert` with respect to `h`. So, `h dot insert 17`, as suppose to insert `h 17` which would be the normal functional style of writings. We would normally pass it the heap and the value, here instead we say given the heap `h` apply to the heap `h` the function `insert` with the argument 70.

The next line says apply to the heap `h` in function `insert` to the value 28 and then for instance we can now ask `h` to return the maximum value by it is an `h dot delete max` and store the return value in the main `v`.

(Refer Slide Time: 14:28)

## Summary

- An abstract data type is a black box description
  - Public interface — update/query the data type
  - Private implementation — change does not affect functionality
- Classes and objects can be used for this
- More details in the next lecture

So, what we would like to emphasize is that an abstract data type should be seen as a black box. Like a black box has a public interface the buttons that you can push to update and query the data type to add things, delete things, and find out what whether the data type is empty and so on. Inside we have a private implementation. This actually stores data in some particular way to make the public functions work correctly. But the important thing is, changing the private implementation should not affect how the functions behave in terms of input and output.

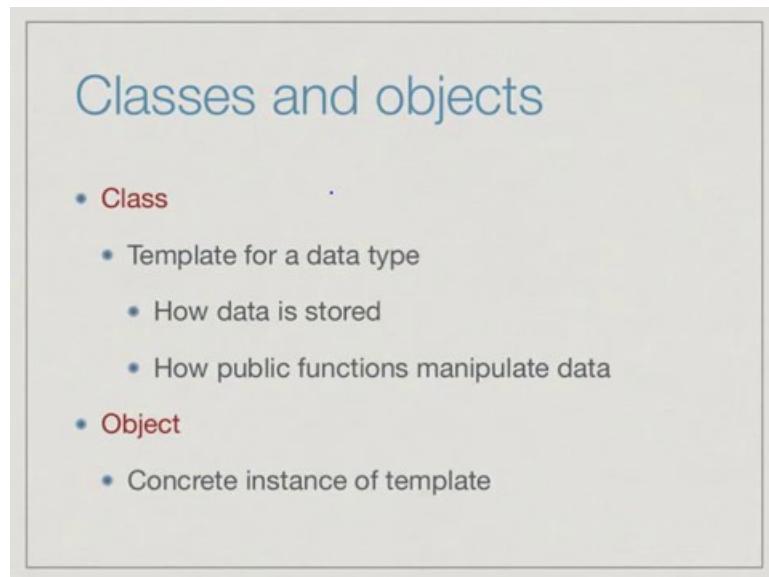
They may behave differently in terms of efficiency, you might see that one version is faster than another or one version slower than another, but this is not the same as saying that the functions change. So, we do not want the values to change, if we have a priority queue and we insert a set of values and then delete max no matter how the priority queue is actually implemented internally the delete max should give us the same value at the end.

So, we saw that python supports object oriented programming, we shall look at it in more detail in the next couple of lectures in these weeks course, but the main concept associated with this objected oriented programming are classes and objects. Classes are templates for data types and objects are instances of these classes they have a concrete data types which we use in our program.

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

**Week - 07**  
**Lecture - 02**  
**Classes and Objects in Python**

(Refer Slide Time: 00:02)



**Classes and objects**

- **Class**
  - Template for a data type
    - How data is stored
    - How public functions manipulate data
  - **Object**
    - Concrete instance of template

In the lecture, we saw that in object oriented programming we define a data type through a template called a class, which defines the internal data implementation, and the functions that we use to manipulate the data type. And then we create instances of this data type as objects.

(Refer Slide Time: 00:22)

## Classes and objects

```
class Heap:  
    def __init__(self,l):  
        # Create heap  
        # from list l  
        self.l = l  
  
    def insert(self,x):  
        # insert x into heap  
        self.l.append(x)  
  
    def delete_max(self):  
        # return max element  
        return self.l.pop()  
  
# Create object,  
# calls __init__()  
l = [14,32,15]  
h = Heap(l)  
  
# Apply operation  
h.insert(17)  
h.insert(28)  
v = h.delete_max()
```

We saw a skeleton implementation of a heap. This had a special function called `init`, which was used to create the heap, and then we `had` functions `insert` and `delete`. Now one thing which we did not explain is this argument `self` that runs through this. This is a convention which we have in `python` that every function defined inside a class should have as its first parameter the name `self`, now it need not be called `self`, but it `is` less confusing `to` always call it, `self`.

Let us just assume that this parameter is always there and it is called `self`. Now, what is `self`, `self` is a name that is used inside the class to refer to the object that we are currently looking `at`. For instance, if we are dealing with this heap `h`, when we say `h` dot `insert` then this `insert` is using the value `17`. So, `17` is the value `x` which is being passed to `insert`, and `h` is the value on which the `17` should be added and that is a name for `self`. So, `self` in other words, tells the function which object it is operating on itself. It is a name to itself because you are telling a function an object `h` insert `17` into your ‘`self`’.

In that sense, `my` values are denoted by `self` and inside a heap, it can `may` be `refer` to other heaps. We will see a little later that we can take one value and refer to another value. There will be `my` values and there will be other values. So, `my` values are always implicitly called `self`, because that is `the` one that is typically `manipulated` by a function.

To make this a little clearer, let us look at a slightly simpler example than heaps to get all the notations and the terminology correct for us.

(Refer Slide Time: 02:14)

```
class Point:  
    def __init__(self, a, b):  
        self.x = a  
        self.y = b  
  
    def translate(self, deltax, deltay):  
        # shift (x,y) to (x+deltax, y+deltay)  
        self.x += deltax # same as self.x =  
                         #           self.x + deltax  
        self.y += deltay
```

Our first example is just a representation of a point  $x$   $y$ . So, we are just thinking about a normal coordinate system, where we have the x-axis, the y-axis. Therefore, a given point is given some coordinate like a comma b. This is a point with x-coordinate a and y-coordinate b, this is a familiar concept that all of you must have seen in mathematics somehow. So, we want to associate with such an object two quantities - the x-coordinate and the y-coordinate and this is set up by the init function by passing the values a and b that you want point to have.

And now we have within the point, we have these two attributes x and y, means every point looks like this. It has an x value and a y value, and this x value is something and the y value is something. And if we change this x and y value, then the point shifts around from one place to another.

Now, in order to designate that the x and y belong to this point and no other, we prefix it by self. So, self dot x refers to the x value within this point myself, self dot y is y value within myself. If you have a generic point p then we have p dot x, p dot y these will refer

to the values x and y the names x and y associated with the point p.

Inside the class definition, self refers to the value of the attribute or the name within this particular object. Now this particular object changes as we move from one object to another, but for every object self is itself. So, for p 1 if I tell something about p 1 well in the context of p 1 self is p 1. If I have different point p 2 in the context of p 2, self is p 2. This is an important thing. Just remember that every function inside a class definition should always have the first argument as self and then the actual argument. So, init in this case takes two arguments, but we always insert a third argument.

(Refer Slide Time: 04:27)

### Points on a plane

```
class Point:
    def __init__(self, a, b):
        self.x = a
        self.y = b

    def translate(self, deltax, deltay):
        # shift (x,y) to (x+deltax, y+deltay)
        self.x += deltax # same as self.x =
                          #           self.x + deltax
        self.y += deltay
```

Handwritten notes on the slide:

- $\cancel{z} = z - b$
- $z = z - b$

Let us look at how this works. For instance, if we say p is equal to point 3, 2; then 3 will be passed as a, and 2 will be passed as b. And this will set up a point that we have drawn here, which internally is represented as self dot x is 3 and self dot y is 2. Now here is a slightly different function, it takes a point and shifts it. So, you want to say shift this to 3 plus delta x and 4 plus delta y. This function we called translate. So, it takes the amount of shift as the argument, delta x and delta y. And as usual we are always providing self as the default first argument.

It just keep this in mind every python function, every python class, if you want to use a

function in the object oriented style, the first argument must necessarily be self and then the real arguments. So, what do we want to do when we want to translate a point, we want to take self dot x and move it to self dot x plus the value delta x. So, you want self dot x plus delta x. Now, this is a very common paradigm in python and other programming languages where you want to take a name say z, and then you want to shift it by some amount, say z plus 6 or z is equal to z minus 6.

Whenever we have this kind of a thing where the same name is being updated, there is a short form where we can combine the operation with the assignment. So, self dot x plus equal to delta x is just a short cut in python for self dot x equal to self dot x plus delta x; it means that implicitly the name on the left is the first argument to the operation mentioned along with the assignment operation. This is a very convenient shortcut which allows us to save some typing. Instead of writing self dot x equal to self dot x plus delta x we just say self dot x plus equal to delta x. This shifts the x coordinate of the current point by the argument provided to translate. Similarly, self dot y plus equal to delta y will shift the argument by the amount provided by delta y.

(Refer Slide Time: 06:37)

## Points on a plane

```

class Point:
    def __init__(self,a,b):
        self.x = a
        self.y = b

    def translate(self,deltax,deltay):
        # shift (x,y) to (x+deltax,y+deltay)
        self.x += deltax # same as selfx =
                          # self.x + deltax
        self.y += deltay

```

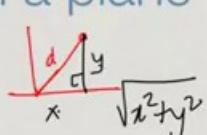
The diagram illustrates the translation of a point. It shows a coordinate system with a horizontal x-axis and a vertical y-axis. A point labeled  $\bullet(3, 2)$  is plotted. An arrow points from this point to a second point labeled  $\bullet(5, 3)$ . Above the plot, the code for the `translate` method is shown. Red annotations highlight the line `self.x += deltax` and the line `self.y += deltay`, indicating that the original values of `self.x` and `self.y` are being modified in place.

For instance, now if we say p dot translate 2 1 then we get a new point which 3 plus 2 5 for the x coordinate and 2 plus 1 3, so this 3 plus 2 gives us 5, and 2 plus 1 gives us 3.

This shifts the point from 3, 2 to 5, 3. This is how we define these internal. The internal implementation is defined inside the init function; this is the function that is called when the point is set up and this associates these internal names x and y with the objects. This is where the implementation really gets set up, and then the functions that we define externally like translate manipulate this internal representation in an appropriate way, so that it changes consistently with what you expect the functions to be.

(Refer Slide Time: 07:24)

## Points on a plane



```

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self):
        # Distance from (0,0)
        # from math import *
        return(
            sqrt(
                (self.x*self.x) + (self.y*self.y)
            )
        )
    p = Point(3,4)
    n = p.distance()

```

Let us look at different functions. So supposing we want to compute the distance of a point from the origin. So, we want to know what is this distance. This distance by Pythagoras' theorem is nothing but the square root of x square plus y square. So, remember this is like a hypotenuse of a right angled triangle. So, you take a x square plus y square root and you get d.

If you want the distance of a point, we do not give it any arguments, but we always have this default argument self. So, we want to know what is the distance from 0, 0 to the current point. So, we would say something like in our earlier case p is equal to point say 3 comma 4 and then we will say p dot o distance to get its distance. Maybe we would assign this to a name, let us not call it. So, we have might assign this to a name like n right.

So, when we do this, it will look at the current value of self dot x, the current value of self dot y, square them, add them and take the square root. Now one thing to remember is that actually square root is not a function available by default in python. So, you actually have to import the math library. At the top of your class definition, you should have remembered to write from math import star. Assuming that we are done that then square root is defined. This is a typical function which returns some information about the point; the earlier function just translated the point, did not tell us anything; this is the function that returns information about the point.

(Refer Slide Time: 09:01)

## Polar coordinates

- Recall polar coordinates
- Instead of  $(x,y)$ , use  $(r,\theta)$ 
  - $x = r \cos \theta$
  - $y = r \sin \theta$
  - $r = \sqrt{x^2 + y^2}$  — same as distance
  - $\theta = \tan^{-1}(y/x)$

$$\frac{y}{r} = \tan \theta$$

Now if o distance is something that we need to do often, then may be it's useful to just keep the point in a different representation. So, you may remember or you may have seen somewhere in high school mathematics that if you take the point x, y then an alternative way of representing where this point is to actually use polar coordinates. So, you can keep this distance and you can keep this angle. So, if I have r and theta it's the same information as keeping x and y.

The connection between the two is that x is  $r \cos \theta$  where cos is the trigonometric cosine function; y is equal to  $r \sin \theta$ . And on the other hand, if I have x and y then I can recover r as we just did for o distance it's the square root of x square plus y square,

and theta so  $y$  by  $x$  is actually if you if you divided  $y$  by  $x$  you get tan of theta that is because it's sin divided by cos and the  $r$  cancels. So,  $y$  by  $x$  is tan theta, so theta is the tan inverse of  $y$  by  $x$ . Now speaking of changing implementation, we could change our implementation, so that we do not keep the internal representation in terms of  $x$  and  $y$ , we actually keep it in terms of  $r$  and theta, but the functions remain the same.

(Refer Slide Time: 10:22)

## Points on a plane

```

class Point:
    def __init__(self,a,b):
        self.r = sqrt(a*a + b*b)
        if a == 0:
            self.theta = 0
        else:
            self.theta = atan(b/a)

    def odistance(self):
        return(self.r)

    def translate(self,deltax,deltay):
        # Convert (r,theta) to (x,y) and back!

```

For instance, we could take the earlier definition and change it. So, we again pass it  $x$  and  $y$ . So, from the user's prospective, the user believes that the point is defined in terms of the  $x$  and  $y$  coordinate, but instead of using  $a$  and  $b$  as the argument directly to set up the point, we first set up the  $r$  - the radius by taking square root of  $a$  square plus  $b$  square.

And then depending, so we want to divide  $b$  by  $a$ , but if  $a$  is 0, then we have a special case  $b$  by  $a$  will give us an error. So, if  $a$  is 0, we set the angle to be 0; otherwise, this is the python function in the math library for tan inverse, arc tan, we set theta to be the arc tan  $b$  minus  $a$   $b$  divided  $a$ . So, we internally manipulate the  $x$   $y$  version to  $r$  theta using the same formula that we had shown before which is that  $r$  is square root of  $x$  square plus  $y$  square and theta is tan inverse of  $y$  by  $x$ . Only thing we have to take care is when  $x$  is equal to 0, we have to manually set theta to 0.

Now internally we are now keeping self dot r and self dot theta. We are not keeping self dot x and self dot y. This is useful because if you want the o distance - the origin distance we just have to return the r value we do not do any computation. So, in other words if we are going to use o distance very often then it is better to use the calculation square root a square plus b square once at the beginning when we setup the point, and just return the r value without any calculation whenever you want the distance from the origin.

This might be a requirement depending on how you are using it and one implementation may be better than the other, but from the user's perspective the same function is there there is self there is o distance. So, if I take a point and I ask for o distance I get the distance from the origin whether or not the point is represented using x y or r theta.

Now, of course, using o distance is r theta is good for o distance not very good for translate. If I want to translate the point by delta x delta y, I have to convert the point back from r theta to x, y; using x equal to r cos theta and y equal to r sin theta then do x plus delta x, y to plus delta y and convert it back to r theta right. So, you pay a price in one function or the other; with the x y representation translate is better; with the r theta representation o distance is better. And this is a very typical case of the kind of compromise that you have to deal with and you have to decide which of these operations is slightly to be more common and more useful for you to implement directly.

If you think translate happens more often it's probably better to use x and y; if you think origin from the distance is more important, so probably better to use r and theta. So, often there is no one good answer. It is not like saying that a heap implementation is always better than a sorted list implementation for a priority queue. There may be tradeoffs which depend on the type of use they are going to put a data structure to as to which internal implementation works worst best, but what you have to always keep in mind is that the implementation should not change the way the functions behave. To the external user, function must behave exactly the same way.

(Refer Slide Time: 13:31)

## Points on a plane

```
class Point:  
    def __init__(self,a,b):  
        self.r = sqrt(a*a + b*b)  
        if a == 0:  
            self.theta = 0  
        else:  
            self.theta = atan(b/a)  
  
    def odistance(self):  
        return(self.r)  
  
    def translate(self,deltax,deltay):  
        # Convert (r,theta) to (x,y) and back!
```

- Private implementation has changed
- Functionality of public interface remains same

In this particular example just to illustrate what we have seen, again. We have changed the private implementation, namely we have moved from x, y to r theta, but the functionality of the public interface the functions o distance translate etcetera remain exactly the same.

(Refer Slide Time: 13:48)

## Default arguments

```
class Point:                      # Point at (3,4)  
    def __init__(self,a=0,b=0):    p1 = Point(3,4)  
        self.x = a                # Point at (0,0)  
        self.y = b                p2 = Point()  
    . . .
```

Now we have seen earlier that in python functions, we can provide default arguments which make sometimes the argument optional. So, for instance, if you want to say that if we do not specify the x and y coordinates over a point then by default the point will be created at the origin. Then we can use a equal to 0, and b equal to 0 as default arguments, so that if the user does not provide values for a and b, then x will be set to 0 and y will be set to 0.

For instance, if we want to point at a specific place 3 comma 4, we would invoke this function this class, we create an object by passing the argument 3 comma 4, but if we do not pass any argument like p 2 then we get a point at the origin.

(Refer Slide Time: 14:35)

Special functions

- `__init__()`
  - Constructor, called when object is created
- `__str__()`
  - Return string representation of object
  - `str(o) == o.__str__()`
  - Implicitly invoked by `print()`

```
def __str__(self): # For Point()
    return(''+str(self.x)+','+str(self.y)+')')
```

self.x  
self.y  
"(x,y)"

The function init clearly looks like a special function because of these underscore underscore on either side which we normally do not see when we or normally do not thing of using to write a python function. As we said before python interprets init as a constructor, so when we call a object like p equal to 0.54, then this implicitly calls init and init is used to set up self dot x self dot y. The internal representation of the point is set up in the correct way by init. So, init is a special function. Now python has other special functions.

For instance, one of the common things that we might want to do is to print out the value of an object, what does an object contain. And for this the most convenient way is to convert the object to a string. The function str normally converts an object to a string, but how do we describe how str should behave for an object, well there is special function that we can write called underscore underscore str. So, underscore underscore str is implicitly invoke when we write str of o. So, str of an object o is nothing but o dot underscore underscore str.

And for instance, print - the function print implicitly takes any name you pass to print and converts it to a string represent, when I say print x and x is an integer implicitly str of x is what is displayed. So, str is invoked and str in turn internally invokes this special function underscore underscore str. Let us see how this would work for instance for our point thing. So, if we want to represent the points so internally we are self dot x and self dot y, we want to print this out in this form value x and the value y.

So, what we do? We set up str, so remember that self is always a parameter. So, what it does is, it first creates a string with the open bracket and the close bracket either end and a comma in the middle; and in between the open bracket and the comma it puts the string representation of self dot x and in between the comma and the close bracket it produces the string representation of self dot y. This creates a string from the value, it internally invokes str on the values themselves self dot x and self dot y and then constructs these extra things the open close bracket and the comma to make it look like a point as we would expect.

(Refer Slide Time: 16:55)

## Special functions

- `__add__()`
  - Invoked implicitly by +
  - $p1 + p2 == p1.__add__(p2)$

Another interesting function that python has as a special function is add. So, when we write plus the function add is invoked. In other words  $p1 + p2$ , if these are two points would invoke  $p1.\_\_add\_\_(p2)$ . So, what we would expect that if I had  $p1$  and if I had  $p2$  then I would get something which gives me a point where I combine these two.

So, I get the x coordinate as  $x1 + p1$  plus  $p2$  and y coordinate  $p1$  plus  $p2$ . It's up to us. I mean it does not mean it has to be this way, but if I say  $p1 + p2$  the function that is invoked is add. And it is up to us define what add means. Let us assume that we want to construct a new point whose x coordinate and y coordinate is the sum of the two points given to us.

(Refer Slide Time: 17:42)

## Special functions

- `__add__()`
    - Invoked implicitly by +
    - $p1 + p2 == p1.__add__(p2)$
- ```
def __add__(self,p): # For Point()
    return Point(self.x+p.x, self.y+p.y)

p1 = Point(1,2)
p2 = Point(2,5)
p3 = p1 + p2 # p3 is now (3,7)
```

Here is way we **would** do it; we would create a new point whose x coordinate is self dot x plus p dot x. Now, notice that, self is the function associated with p 1 in this case, add; so self refers to p 1. When I say p 1 plus p 2 and the other argument p 2 is the point p. So, I can look at the values of p and say p dot x p dot y, I can look at my value **and say** self dot x self dot y, and now I can combine these by creating a new point self dot x plus p dot x and self dot x plus p dot y.

For instance, if we have two points at 1, 2 and 2, 5 then this will return a point at 3, 7 and I must store it in new point, so p 3 now becomes a new point whose x coordinate is 3 and y coordinate is 7.

(Refer Slide Time: 18:30)

## Special functions

- `__mult__()`
  - Called implicitly by \*
- `__lt__()`, `__gt__()`, `__le__()`, . . .
  - Called implicitly by <, >, <=
- Many others, see Python documentation

In the same way, we could have a special way of defining multiplication, and in python the underscore underscore mult function is the one that is implicitly called by multiply. Similarly, we can define comparisons; we might say what is to be done when we compare whether p 1 is less than p 2, do we check both coordinates are less, we check the distance from the origin is less; we have complete freedom how to define this.

We just write p 1 less then p 2 for readability enough in our program, and internally it will call a function less than lt, similarly greater than will call the function gt, and there is something for less than equal to greater then equal to and so on. These are all very convenient functions that python allows us to call implicitly, and allows us to use conventional operators and conventional functions in our code. So, we do not really have to think of these objects in a different way when we are coding our programs.

There are several other such functions; it is impossible to list all of them and it is not useful to list all of them at this introductory stage when you are learning python, but you can always look up the python documentation, and see all the special functions that are defined for objects and classes.

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

**Week - 07**  
**Lecture - 02**  
**User Defined Lists**

(Refer Slide Time: 00:03)

## Designing our own list

$l = \text{None}$

- A list is a sequence of nodes
- Each node stores a value, points to next node
- How do we represent the empty list?

The diagram illustrates the representation of a list as a sequence of nodes. It shows two cases: a non-empty list  $l$  pointing to four nodes  $v_1, v_2, v_3, v_4$ , where  $v_4$  points to `None`; and an empty list  $l$  pointing to a single node  $v_1$ , which also points to `None`. The empty list case is circled in red.

Now that we have seen the basics **about** how to define classes and objects, let us define an interesting data structure.

Suppose we want to implement our own version of **the** list, **a** list is basically a sequence of nodes and each node in the sequence stores a value and points **to** the next node. So, in order to go through the list we have to start at the beginning and walk following these pointers till we reach the last pointer which points to nothing. We have a list of the form  $v_1, v_2, v_3, v_4$  in python notation. Then this is how we would imagine it is actually **represented**. **There** are 4 nodes. The list  $l$  itself which we have set up points to the first node in this list,  $v_1$  points to  $v_2$ ,  $v_2$  points to  $v_3$  and  $v_3$  points to be  $v_4$ . The final node points to nothing and that indicates we have reached the end of the list.

In this representation what would the empty list look like well it is natural to assume that

the empty list will consist of a single node which has both the value and the next node pointers set to none, whereas for instance the singleton would be a single node in which we have the value v 1 and the next set to none. So, this is the convention that we shall follow for our representation of a list. So, notice that unless we have an empty list with a single node none, none no other node in a list can have value none, right. This is something that we will implicitly assume and use that checking for the value none will tell us it is an empty list and we will never find none in the middle of a list.

We distinguish between a singleton and an empty list purely based on the value. Both of them consist of a single node. Now the reason that we have to do this is because actually python does not allow us to create an empty list if we say something like l is equal to none and we want this to denote the empty list the problem is that none does not have a type as far as python's value system is concerned. So, once we have none, we cannot apply the object functions we are going to create for this list type. So, we need to create something which is empty of the correct type. So, we need to create at least 1 node and that is why we need to use this kind of representation in order to denote an empty list.

(Refer Slide Time: 02:26)

```
Class Node
# Create empty list
l1 = Node()

# Create singleton
l2 = Node(5)

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

Here is the basic class that we are going to use, it is a class node. So, inside each node we have 2 attributes value and next as we said and remember that self is always used with

every function to denote the object under consideration. We will use this default scheme that if we do not say anything we create an empty list. The init value, this should be init val.

The initial value is by default none unless I provide you an initial value in which case you create a list with that value and because of our assumption about empty list all we need to do to check whether a list is empty is to check whether the value at the initial node is none or not. We just take the list we are pointing to and look at the very first value which will be self dot value and ask whether it is none. If it is none, it is empty. If it is not none, it is not empty.

Here is a typical thing. We say l1 is equal to node; this creates an empty list because it is not provided any value. So, the default initial value is going to be none. If I say l2 is equal to node 5 this will create a node with the value 5. It will create the singleton list that we would normally write in python like this. If I ask whether l1 is empty, the answer will be true. If I ask whether l2 is empty, the answer will be false because self dot value is not none.

(Refer Slide Time: 03:50)

## Append a value v

- If list is empty, replace None by v
- If at last element of list (next is None)
  - Create a node with value v
  - Set next to point to new node
- Otherwise, recursively append to rest of the list

Now, once we have a list what we would like to do is manipulate it. The first thing that

we might want to do is add a value at the end of the list. If the list is already empty, then we have a single node which has value none and we want to make it a singleton node, a singleton list with value v. So, we want to go from this to this, remember that in a singleton node we just have instead of none we have the value v over here so that is all we need to do. We need to just replace the none by v, if we are at the last element of the list and we know that we are at the last element of the list because the next value is none then what we need to do is create a new value.

We walk to the end of the list and then we reach none. Now, we create a new element here with the value v and we make this element point to this, we create a new element with the node v and set the next field of the last node to point to the new node and if this is not the last value then well we can just recursively say to the rest of the list treat this as a new list starting at the next element, take the next element and recursively append v to that.

(Refer Slide Time: 05:05)

### Append a value v

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

This gives us a very simple recursive definition of append. So, we take append and we want to append v to this list. If it is empty, then we just set the value to v. So, this just converts the single node with value none to the single node with value of v, otherwise if we are at the last node that is self dot next is none then we create a new node with the

value v and we set our next pointer to point at the new node, remember when we create a new node the new node automatically is created by our init function with next none.

We would now create a new node which looks like v and none and we will set our next pointer to point to it and the final thing is that if it is not none then we have something else after us. So, we go that next element self dot next and with respect to that next element we reapply the append function with the value v, this is the recursive call.

We have been abundantly careful in making sure that this is parsable. So, we have put this bracket saying that we take the object self dot next and apply append to that actually python will do this correctly. We need not actually put the bracket, we can just write self dot next dot append v and python will correctly bracket this as self dot next dot append. So, this dot is taken from the right.

(Refer Slide Time: 06:23)

### Append a value iteratively

- If list is empty, replace None by v
- Scan the list till we reach the last element
- Append the element at the last element

Now, instead of recursively going to the end of the list we can also scan the end of the list till the end iteratively. We can write a loop which keeps traversing these pointers until we reach a node whose next is none. If the list is empty as before we replace the value none by v, otherwise we scan the list till we reach the last element and then once we reach the last element as in the earlier case we create a new node and make the last

element point to it.

(Refer Slide Time: 06:53)

## Append value iteratively

```
def append(self, v):
    if self.isEmpty():
        self.value = v
        return()
    temp = self
    while temp.next != None:
        temp = temp.next
    newnode = Node(v)
    temp.next = newnode
    return()
```

The diagram shows a linked list with three nodes. A green arrow labeled 'temp' points to the second node. Red arrows indicate the movement of pointers: one from the second node's 'next' field to the third node, and another from the first node's 'next' field to the second node.

This gives us **an** append which is iterative. So, we call it append i just to indicate that **it is** iterative. The first part is the same if the current list is empty then we just set the value to be v and we return, otherwise we now want to walk down the list. We set up a temporary pointer to point to the current node that we are at and so long as the next is none we keep shifting temp to the next value. So, we just write a loop which says while temp dot next is not none just keep going from temp to temp dot next.

**So, just** keep shifting **temp**. Finally when **we** come out of this loop at this point we know that **temp** dot **next** is **none**. This is the condition under which we **exit** the loop. We have reached, the node **temp** is now pointing to the last node in the current list. **At** this point we do exactly what we did in the recursive case we create a new node with a value v and we make this last node point to this new node. So, we reset next of **temp** from none to the new node.

(Refer Slide Time: 07:57)

## Insert a value v

- Want to insert v at the head of the list
- Create a new node with v
  - But we cannot change where l points to!
- Instead, swap the contents of v with the current first node

```
graph LR; l --> v1[v1]; v1 --> v2[v2]; v2 --> v3[v3]; v3 --> v4[v4]; v4 --> None[None]; v1 -- "x v1" --> v1_new[v1]; v1_new --> v2; v1 --> None;
```

What if we do not want to append, but we want to **insert**. Now it looks normally that **insert** should be easier than **append**, but actually **insert** is a bit tricky. So, by **insert** we mean that we want to put a value at the beginning. We want to put a node here which has v and make this pointer.

This is what we want to do **now**. The problem with this really is that after we create a new node we cannot make this point here and this point here there is no problem in making the new node point to v 1, but if we reassign the value of l or inside a object if we reassign the value of self then this creates a completely different object. We saw this when we were looking at how parameters are passed and immutable value are **passed**.

We said that if we pass a mutable value to a function so long as we do not reassign that thing any mutation inside the function will be reflected outside the function, but if we reassign **to the list** or dictionary inside the function we get a new copy and then after that any change we make is **off**. So same way if we reassign l or self to point to a new node then we will lose the connection between the parameter we **passed to** the function and the parameter we get back. So, we must be careful not to make l point to this thing. We cannot change where l **points** to. So, how do we get around this **problem**? We have created a new node, we want to make l point to it, but we are not allowed to do **so**,

because if we do so, then python will disconnect the new l from the old l. So, there is a very simple trick. What we do is we do not change the identity of the node, we change what it contains. So, we know now that v 1 is the old first node and v is a new first node, but we cannot make l point to the new first node, so we exchange the values. So, what we do is we replace v 1 by v and v by v 1.

Now, the values are swapped and we also have to do a similar thing for what is pointing where. So, l is now pointing to v as the first node, but now we have bypassed v 1 which is a mistake. We must now make the first node point to the new node and the new node point to the old second node. So, by doing this kind of plumbing what we have ensured is that the new list looks like we have inserted v before the v 1, but actually we have inserted a new node in between v and v 2 and we have just changed the links to make it appear as though the new node is second and not first.

(Refer Slide Time: 10:22)

### Insert a value v

```
def insert(self,v):
    if self.isempty():
        self.value = v
        return()
    newnode = Node(v)
    # Exchange values in self and newnode
    (self.value, newnode.value) =
        (newnode.value, self.value)
    (self.next, newnode.next) = (newnode, self.next)
    return()
```

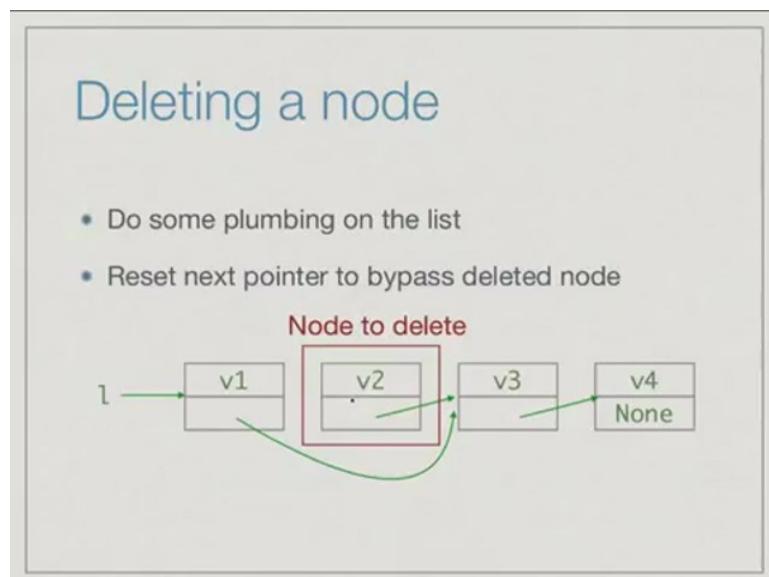
[] → [v]

Here is the code for insert. As usual, if you have an empty list insert is easy. We just have to change none to v. So, insert and append both behave the same way with an empty list. We go from the empty list to the list v. It does not matter whether you are inserting or appending. Otherwise, we create this new node and then we do this swapping of values between the current node that self is pointing to, that is the head of the list and the new

node.

We exchange the values; we set self dot value to new node dot value and simultaneously new node dot value to self dot value using this python simultaneous assignment. And similarly we take self dot next which was pointing to the next node and make it point to the new node and the new node instead should point to what we were pointing to earlier. So, new node dot next is self dot next. This is how we insert and insert as we saw is a little bit more complicated than append because of having to handle the initial way in which l points to the list or self points to the list.

(Refer Slide Time: 11:19)



What if we want to delete a node? How do we specify to delete a node? Well we specify it by a value, but let us just suppose you want to delete say the second node in this list.

Now, how would we delete it? Well again just as we did insert we would do some re plumbing or re connection. So, we take the node that we want to delete and we just make the link that points to v 2 bypass it. So, we take the link from v 1 and make it directly point to v 3. So, in essence, all that delete requires us to do is to reassign the pointer from before the deleted node with the pointer after the deleted node. It actually does not physically remove that object from memory, but it just makes it inaccessible from the

link end.

We provide a value  $v$  and we want to remove the first occurrence of  $v$ . We scan the list for the first  $v$ . Now notice that in this plumbing procedure we need to be at  $v_1$  in order to make it point to  $v_3$ . If we wanted **to delete** the next **node** then we are in good shape because we can take the next dot next and assign it to the current **next**. So, we should look 1 step ahead. If you are already **at**  $v_2$  then we have gone **past**  $v_1$  and we cannot go back to  $v_1$ , easily the way we have set up our list because it only goes forward; we cannot go back to  $v_1$  and change it.

(Refer Slide Time: 12:38)

## Delete a value $v$

- Remove first occurrence of  $v$
- Scan list for first  $v$
- If `self.next.value == v`, bypass `self.next`
  - `self.next = self.next.next`
- What if first value in the list is  $v$ ?

What we will actually do is we will scan by looking at the next value. If the `self` dot `next` dot value is  $v$  that is **if the** next node is to be deleted then we bypass it by saying the current **node's next is not the** next node that we had, but the next node's `next`. So, `self` dot `next` is **reassigned** to `self` dot `next` dot `next` - **bypass** the next node. As before like **with** insert the only thing we have to be careful about **is** if we have to delete actually the first value in the list.

(Refer Slide Time: 13:12)

## Deleting first value in list

- `l.delete(v1)`
- Cannot delete the node that `l` points to
  - Reassigning name in function creates a new object
- Instead, copy `v2` from next node and delete second node!

If you want to delete the first value in the list exactly like we had with insert the natural thing would be to, now say that `l` should point to the second value in the list, but we cannot point `l` there because if we reassign the node that `l` points to then it will create a `new` object and it will break the connection between the parameter `we passed and` the thing we get back. We use `the` same trick. What we do is we copy `the value v 2` from the next node and `then... So we` just copy this value from here to here and then we delete `v 2`. So, we wanted to delete the first node, we are not allowed to delete the first node because we cannot change `what` `l` points to. So, instead we take the value in the second node which was `v 2`, copy it here and then pretend we `deleted` `v 2` by making `the` first node point to the third.

(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()
```

[x] → []

Bypass

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

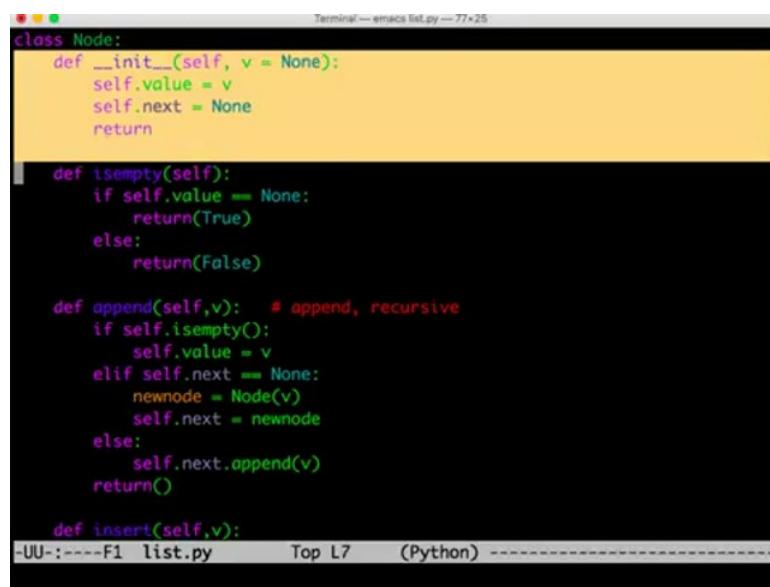
```
def __str__(self):
    selflist = []
    if self.value == None:
        return(str(selflist))
    temp = self
    selflist.append(temp.value)
    while temp.next != None:
        temp = temp.next
        selflist.append(temp.value)
    return(str(selflist))
```

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)



```
Terminal --- emacs list.py --- 77x25
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)

```
Terminal --- emacs list.py --- 77*25
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)

```
Terminal --- emacs list.py --- 77*25
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()

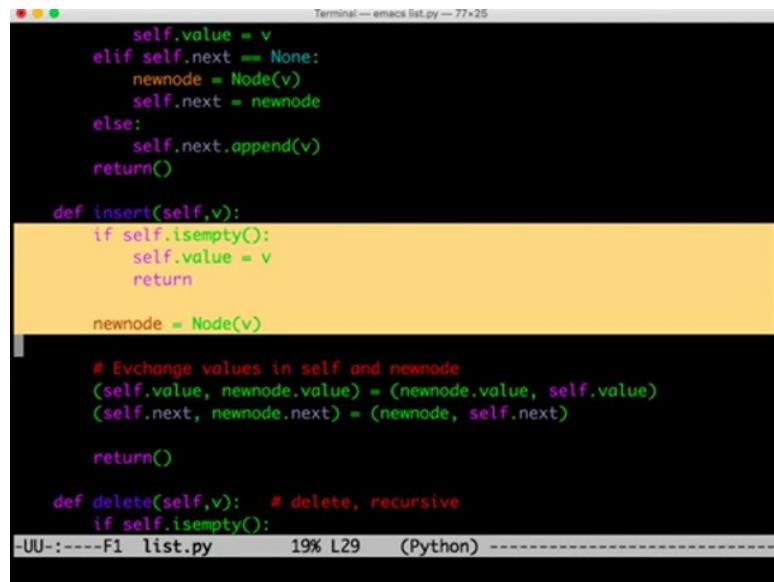
def insert(self,v):
    if self.isempty():
        self.value = v
    return

-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)



The screenshot shows a terminal window titled "Terminal — emacs list.py — 77×25" displaying Python code. The code defines a class with methods for insertion and deletion. The "insert" method handles both empty and non-empty lists by creating a new node or appending to the current tail. The "delete" method handles an empty list by returning. The code uses a Node class and swap operations to maintain pointers. The terminal status bar at the bottom shows the file name "list.py", line count "19%", and column count "L29".

```
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)

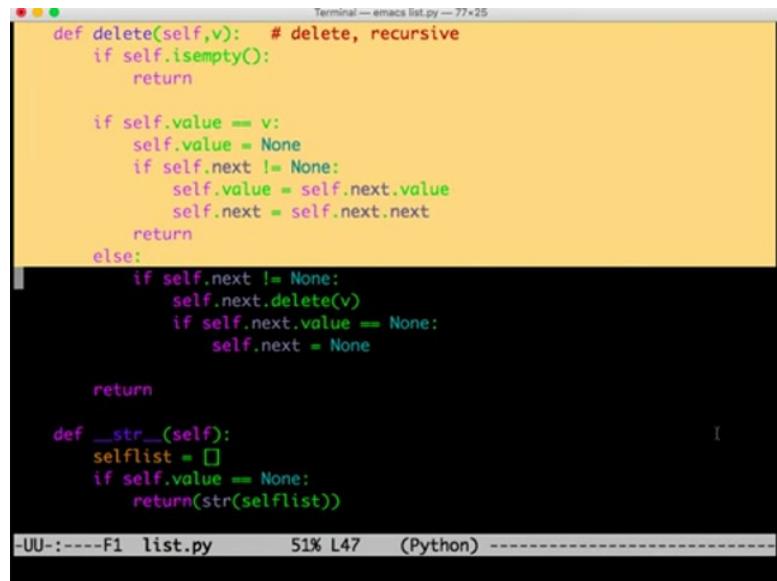
    # Exchange values in self and newnode
    (self.value, newnode.value) = (newnode.value, self.value)
    (self.next, newnode.next) = (newnode, self.next)

    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)



```
Terminal — emacs list.py — 77x25
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) -----
I
```

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 in range(1,11):
...     File "<stdin>", line 1
...     for i in range(1,11):
...         ^
SyntaxError: invalid syntax
>>> for i in range(1,11):
...     ...     l.append(i)
...

```

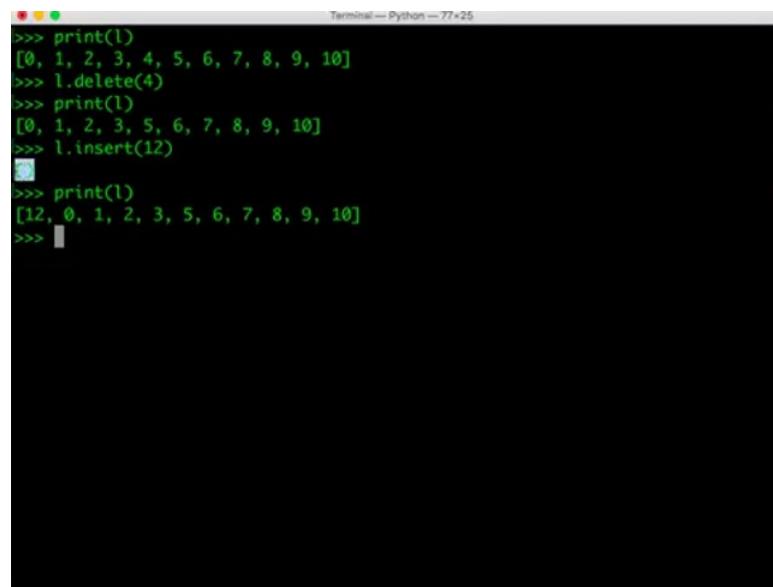
If we now run this by importing, then we could say, for instance, that `l` is a list with value 0 and if we say `print l` then we will get this representation 0, we could for instance put this in a loop and say for `i` in range 1 say 11, `l` dot `append i`.

(Refer Slide Time: 22:34)

```
>>> from list import *
>>> l = Node(0)
>>> print(l)
[0]
>>> for i in range(1,11):
...     File "<stdin>", line 1
...     for i in range(1,11):
...         ^
SyntaxError: invalid syntax
>>> for i in range(1,11):
...     ...     l.append(i)
...
[0]
[0]
[0]
[0]
[0]
[0]
[0]
[0]
[0]
[0]
[0]
>>> 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 1 dot delete 4 for instance and we print 1 then 4 is formed and so on. If we say 1 dot insert 12 and print 1, 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.

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

**Week - 07**  
**Lecture - 04**  
**Search Trees**

(Refer Slide Time: 00:02)

## Dynamic sorted data

- Sorting is useful for efficient searching
- What if the data is changing dynamically?
  - Items are periodically inserted and deleted
  - Insert/delete in sorted list take time  $O(n)$
  - Like priority queues, move to a tree structure

As a final example of a user defined data structure we will look at binary search keys. We are looking at a situation where we need to maintain dynamic data in a sorted manner, remember that one of the byproducts of sorting is that we can use binary search to efficiently search for a value, but binary search can be used if we can sort data once and for all and keep it in sorted order if the data is changing dynamically then in order to exploit binary search will have to keep resorting the data which is not efficient.

Supposing, we have a sequence of items and items are periodically being inserted and deleted now as we saw with heaps, for instance, if we try to maintain a sorted list and then keep track of inserts then each insert or delete, in this case would take order and time and that would also be expensive. However, it turns out that we can move to a tree like structure or a tree structure like in a priority queue it move to a heap and then do insert and delete also efficiently alongside searching.

(Refer Slide Time: 01:11)

## Binary search tree

- For each node with value  $v$ 
  - Values in left subtree  $< v$
  - Values in right subtree  $> v$
- No duplicate values

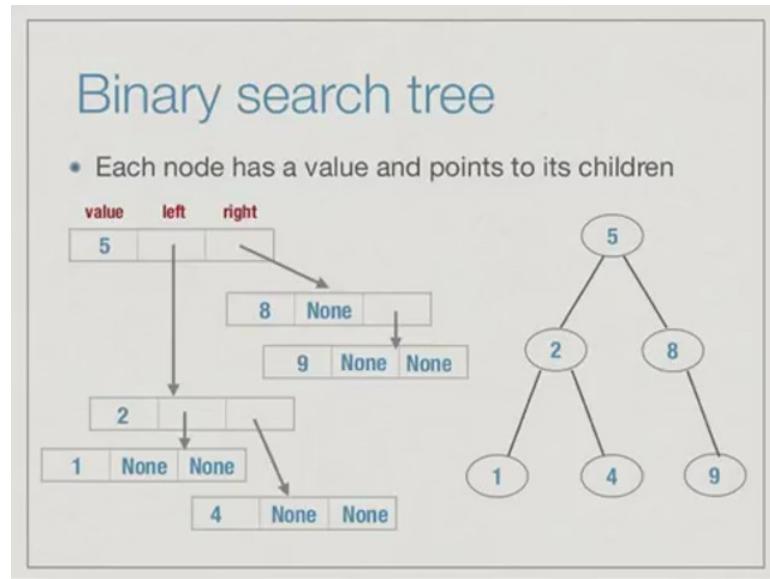
```
graph TD; 5((5)) --- 2((2)); 5 --- 8((8)); 2 --- 1((1)); 2 --- 4((4)); 8 --- 9((9));
```

The data structure we have in mind is called a binary search tree. So, in a binary search tree we keep exactly one copy of every value. It is like a set we do not keep duplicates and the values are organized as follows for every node all values which are smaller than the current nodes value are to the left and all values that are bigger than the current node value are to the right.

Here is an example of a binary search tree, you can check for instance that to the left of the root 5, we have all values 1, 2 and 4 which are smaller than 5 and to the right of 5 we have the values 8 and 9 which are bigger, now this is a recursive property. If you go down, for instance, if you look at the node label two then below it has values 1 and 4 since 1 is smaller than 2, 1 is to the left of 2 and since 4 is bigger than 2, 4 is to the right of 2.

Similarly, if we look at the node 8, it has only one value below it namely 9 and therefore, it has no left child, but 9 is in the right subtree of 8.

(Refer Slide Time: 02:14)



We can maintain a binary search tree using nodes exactly like we did for a user defined lists except in a list we had a value and a next pointer in a binary search tree we have two values below each node potentially a left child and a right child. So, each node now consist of three items the value being stored the left child and the right child.

If we look at the same example that we had 4 on the right then the root node 5 will have a pointer to the nodes with 2 and 8, the node 2 will have a pointer to the nodes 1 and 4, these are now what are called leaf nodes. So, they have no children. Their left and right pointers will be **None** indicating there is nothing in that direction. Similarly, 8 has got **None** as his left pointer because it has no left child and the right pointer points to 9 and the node with nine again has two **None** pointers because it is a leaf node.

(Refer Slide Time: 03:10)

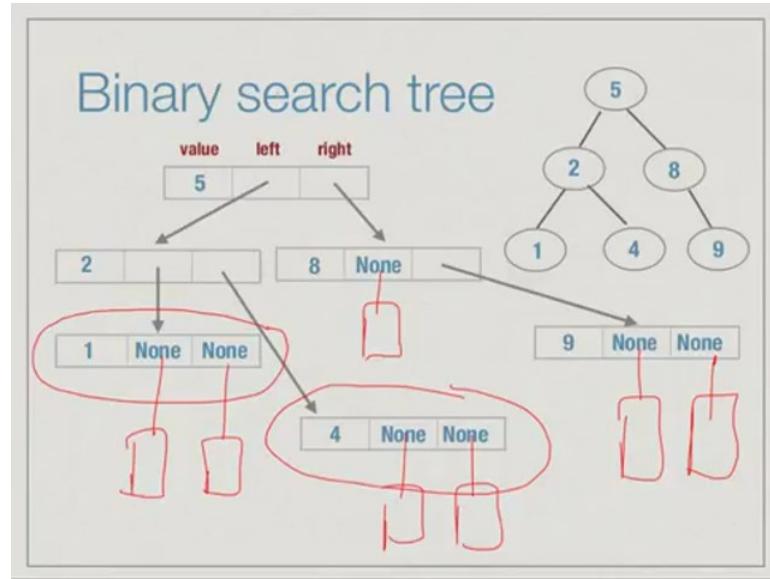
## A better representation

- Add a frontier with empty node: all fields None
- Empty tree is a single empty node
- Leaf node has value that is not None, left and right children point to empty nodes
- Makes it easier to write recursive functions to traverse the tree

Now, it will turn out that we will want to expand this representation in order to exploit it better for recursive presentations. So, what we will do is that we will not just terminate the tree with the value and the two pointers none you will actually add a layer of empty nodes with all fields **None** with this the empty tree will be a single empty node and a leaf node that is not none will have a value and both its children will be empty nodes.

It would not directly have **None** as it is left and right pointers it will actually help children which are empty. So, this makes it easier to write recursive functions and if we do not do this then it is a bit harder to directly implement recursive functions.

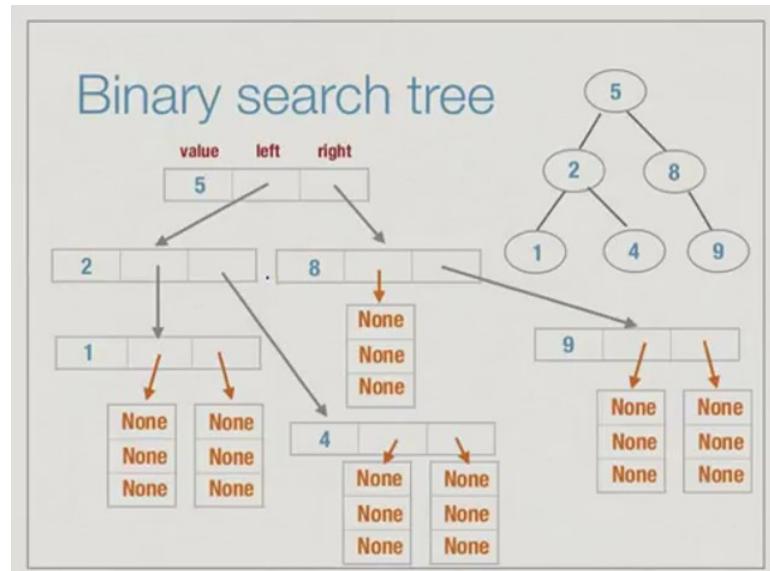
(Refer Slide Time: 03:56)



Just to understand how our tree structure changes this is the structure that we had before the same example. Here, notice that in the leaf nodes the leaf nodes have a value and both the child pointers left and right are directly none. So, we want to change this, what we want to do is to we want to insert below this an empty node at everywhere, where we see **None**, we want to insert and extract the node.

This of course, grows up the tree a little bit, but then this **cost** is not that much as we will as you can calculate. So, if we do this then we get a new tree which looks like this right

(Refer Slide Time: 04:28)



Below every leaf node wherever we normally had a `None` pointer indicating that the path has ended we will explicitly add one extra node which has all three fields none and it will turn out that this is very useful to clean up our programming later on. So, this is a representation that we will use for a binary search tree each node has three pointers and at the leaf's we have an extra layer of empty nodes with all three values none, none, none.

(Refer Slide Time: 04:58)

## The class Tree

```

class Tree:
    def __init__(self, initval=None):
        self.value = initval
        if self.value:
            self.left = Tree()
            self.right = Tree()
        else:
            self.left = None
            self.right = None
        return()
    def isempty(self):
        return(self.value == None)

```

Here is the basic class tree. So, as before we have an init function which takes an initial values which is by default none. The init function works as follows right, we first setup the value to be `initval` which could be none if there is a value, then we create an tree with one node in which case we have to create these empty nodes. Now, notice that if we go back and we go do not give a values. So, maybe it is better to look at this case right if we do not give a value then we end up with a tree we just says none, none, none. So, this is our empty.

The initial value is none we get this tree if the init value is not none then we get a tree in which we put the value v and then we make the left in the right pointers both point to this none, none. So, this is a tree that will contain exactly one value. So, depending on that the init values none or not none we end up either a tree with three nodes with two dummy nodes below or a single empty node denoting the empty tree. So, given this as before we have the function isempty which basically checks if the value is none, if I start

looking at a tree and the very first node says none then that tree is empty otherwise it is not empty.

(Refer Slide Time: 06:15)

## Inorder traversal

```
def inorder(self):
    if self.isEmpty():
        return []
    else:
        return(
            self.left.inorder() +
            [self.value] +
            self.right.inorder()
        )
def __str__(self):
    return(str(self.inorder()))
• Lists values in sorted order
```

```
graph TD; 5((5)) --> 2((2)); 5 --> 8((8)); 2 --> 1((1)); 2 --> 4((4)); 8 --> 9((9));
```

1 2 4 5 8 9

Let us first look at a way to systematically explore the values in a tree. These are called traversals and one traversal which is very useful for a binary search tree is an inorder traversal. So, what an inorder traversal does is that it first explores the left side. So, it will first explore this recursively again using an inorder traversal then it will display this and then it will explore the right.

If you see the code you can see that if the tree is not empty you first do an inorder traversal of the left self tree then you pick up the value at the current node and then you do an inorder traversal of the right self tree and this produces the list of values. So, if we execute this step by step, 5 if we reach it says first do an inorder traversal of the left.

We come down to two this is again not at a trivial tree. So, again we have to do a inorder traversal. So, we go it is left and now when we have one and inorder traversal of one consists of it is left child which is empty one and then it is right child is empty. So, this produces one now I come back and I list out the node two and now I do an inorder traversal of it is right. So, I have got 1, 2, 4. So, this completes inorder traversal of the left subtree of 5. Now, I list out 5 itself and then I do an inorder traversal of 8 and 9 since 8 has no left child the next values are comes out as a 8 itself and then 9.

So, what you can see is that since we print out the values on the left child before the current value when the value is the right child after the current value with respect to the current value all these values are sorted because that is how the search key is organized and since is recursively done at every level down the final output of a inorder traversal of a search tree is always a sorted list. This is one way to ensure that the tree that you have constructed is sorted you do an inorder sub traversal that the key of constructed is a search tree you do an inorder traversal and ensure that the output that you get from this traversal is a sorted list.

(Refer Slide Time: 08:16)

The slide has a light gray background with a white rectangular content area. At the top left of the content area, the text "Find a value v" is written in a blue font. Below this, there is a bulleted list of four items, each preceded by a small black circle:

- Scan the current node
- Go left if v is smaller than this node
- Go right if v is larger than this node
- Natural generalization of binary search

As we mentioned that the beginning of this lecture, one of the main reasons to construct a search tree is to be able to do something like binary search and this is with dynamic data. So, we will also have insert and delete as operations, but the main fundamental operation is find.

Like in binary search we start at the root. So, you imagine that this is the middle of an list of an array, for example, we look at this element if we have found it it is fine if we have not found it then we need to look in the appropriate sub tree since the search tree is organized with the left values smaller in the right value is bigger we go left if the value we were searching for is smaller than the current node and we go right if it is larger than the current node. So, this is very much a generalization of binary search in the tree.

(Refer Slide Time: 09:04)

## Find a value v

```
def find(self, v):
    if self.isEmpty():
        return(False)
    if self.value == v:
        return(True)
    if v < self.value:
        return(self.left.find(v))
    else:
        return(self.right.find(v))
```

Here is the code, it is very straight forward we want to find value of v remember, this is this python syntax. We always have the self as the first parameter to our function. So, if the current node is empty we cannot find it.

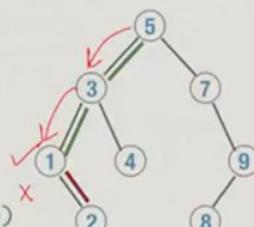
So, if it is v return false if we do find v then we return true and if we do not find v then if it is. This is not, it should be self if it is smaller than this current value then we go left and search for it otherwise we go right and search for it. So, this is exactly a binary search and it is a very simple recursive thing which exactly follows a structure of a search tree.

(Refer Slide Time: 09:43)

## Minimum

- Left most node in the tree

```
def minval(self):
    # Assume t is not empty
    if self.left == None:
        return(self.value)
    else:
        return(self.left.minval())
```



It will be useful later on to be able to find the smallest and largest values in a search tree. So, notice that as we keep going left we go smaller and smaller. So, where is the minimum value in a tree it is along the smaller left most path. So, if I have to go from the left most path and if I cannot go any further then I find it. So, we will always apply this function only when tree is non empty. Let us assume that we are looking for the minimum value in a non empty tree. Well, we find the left most path.

If I cannot go further left then I found it. In this case, if I reach one since I cannot go further one is the minimum value otherwise if I can go left then I will go one more step. So, if we start this, for instance, say 5 it will say that 5 have left a subtree. So, the minimum value below 5 is the minimum value below it is left child. So, you go to three now we say that the minimum value below three is the minimum value below it is left child. So, you go to one then we say that the minimum value is the minimum value at 1 because there is no left child and therefore, we get the answer 1 and it is indeed 2 that anything below that is only on the right that is 2.

(Refer Slide Time: 10:50)

## Maximum

- Right most node in the tree

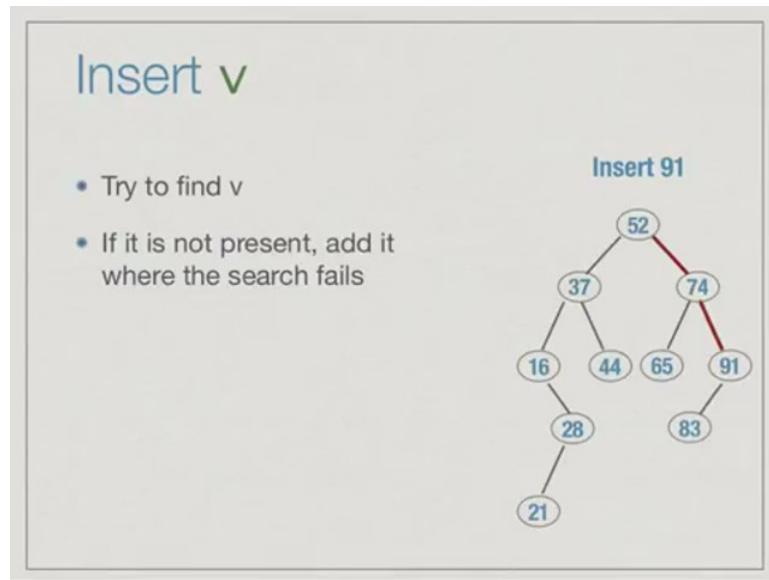
```
def maxval(self):
    # Assume t is not empty
    if self.right == None:
        return(self.value)
    else:
        return(self.right.maxval())
```

Dually, one can find the maximum value by following the right most path right, if the right is none then we return the current value otherwise we recursively look for the maximum value to our right.

In this case we start at 5, we go down to 7, then we go down to 9 and since there is no further right path from 9, 9 must be the maximum value in this tree. We will come back

later on and see why we need this minimum and maximum value, but any way it is useful thing to be able to find and it is very easy to do, again using the structure of the binary search.

(Refer Slide Time: 11:25)



So, one of the things we said is that we need to be able to dynamically add and remove elements from the tree. The first function that we look for is insert, how do we insert an element in the tree well it is quite easy we look for it if we do not find it then the place where our search concludes is exactly where it should have been. So, we insert it at that point right.

For example, supposing we try to insert 21 in this tree, when we look at 52 and when go left when we go left then we come to 16 again and we go right then we come to 21, 28 and we find that we have exhausted this path and there is no possible 21 in this tree, but had we found 21 it should be to the left of 28. So, we insert it there. So, we look for where it should find it and if we do not find it we insert it with the appropriate place. Similarly, you can start and look for 65. So, 65 is bigger than 52. So, we go over right then it is smaller than 74. So, we go left, but there is nothing to the left 74. So, we put it to the left of 74.

Now, insert will not put in a value that is already there because we have no duplicates. So, if now we try to for instance insert ninety one then we go right from 52 we go right

from 74 and now we find that 91 is already present in the tree. So, insert ninety one has no effect on this tree.

(Refer Slide Time: 12:50)

### Insert v

```
def insert(self,v):
    if self.isempty():  # Add v as a new leaf
        self.value = v
        self.left = Tree()
        self.right = Tree()
    if self.value == v: # Value found, do nothing
        return
    if v < self.value:
        self.left.insert(v)
        return
    if v > self.value:
        self.right.insert(v)
        return
```

This is a very simple modification of the find algorithm. So, we keep searching and if we reach the leaf node then we come to an empty node right. If you find that we have reached an empty node then we do the equivalent of creating a new node here we set this value to be v and we create a new frontier below by adding two empty nodes in the left and right rather than just having none.

On the other hand, if we find the value in the tree we do nothing and if we do not find the value then we just use the search tree property and try to insert either on the left or on the right **as appropriate.**

(Refer Slide Time: 13:32)

## Delete v

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

**Delete 65**

```
graph TD; 52((52)) --- 37((37)); 52 --- 74((74)); 37 --- 16((16)); 37 --- 44((44)); 16 --- 28((28)); 16 --- 21((21)); 74 --- 91((91)); 74 --- 83((83))
```

How about delete. So, delete is a little bit more complicated than insert. So, basically whenever we find v in the tree and that can only be one v remember this is not like deleting from the list that we had before where we were removing the first copy of v in a search tree we have only one copy of every value if at all. If we find v we must delete it.

So, we search for v as usual now if the node that we are searching for is a leaf then we are done we just delete it and nothing happens, if it has only one child then we can promote **child**. If it has two children we have a hole we have a leaf we have a node which we have to remove value, but we have values on both sides below it and now we will use this maximum function **maxval** or **minval** in order to do the work. Let us just see how this works through some examples right. So, supposing we first delete 65 then we first search for 65, we find it since it is a leaf then we are in this case the first case we just I have to remove this leaf and we are done.

(Refer Slide Time: 14:40)

## Delete v

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

**Delete 74**

```
graph TD; 52((52)) --- 37((37)); 52 --- 91((91)); 37 --- 16((16)); 37 --- 44((44)); 44 --- 28((28)); 28 --- 21((21));
```

Now, we try to delete 74. So, we find 74 and we find that it has only one child. So, if it has only one child then we move this out then we can just effectively short circuit this link and move this whole thing up and make 52 point to 91 directly. So, we are in this second case this is what it means to promote the child. So, the deleted node has only one child we can bypass the child and directly connect the parent of the deleted node to the one child of the deleted node. So, this will result in 91 moving up there.

(Refer Slide Time: 15:14)

## Delete v

- If v is present, delete it
- If deleted node is a leaf, done
- If deleted node has only one child, "promote" that child
- If deleted node has two children, fill in the hole with `self.left.maxval()` (or `self.right.minval()`)
- Delete `self.left.maxval()`— must be leaf or have only one child

**Delete 37**

```
graph TD; 52((52)) --- 28((28)); 52 --- 91((91)); 28 --- 16((16)); 28 --- 44((44)); 44 --- 21((21));
```

Now, finally, we have the difficult case which is you want to delete a node which is in the middle of the tree, in case this case 37. So, we come to 37 and we want to remove this. Now, if we remove this there will be a vacancy now, what do we fill the vacancy again and how do we adjust the shape of the tree. So, we look to the left and find the maximum value remember that everything to the left is smaller than 37 and everything to the right is bigger than 37.

So, among the left nodes we take the biggest one and we move it there then everything to the left will remain smaller than that node you could also do it the other way and take the smallest values from the right, but you would not do that you will stick to taking the maximum value from the left. We go to the left and find the maximum value is 28. So, basically we have taken this 28 and moving it up there

Now, we should not have two copies of 28. So, we need to remove this 28. So, how do we do that well within this subtree, we delete 28 now this looks like a problem because in order to delete a node we are again deleting a node, remember that the way that the maximum value was defined it is along with the right most path. So, the right most path will either end in leaf or it will end in the node like this which has only one **child** and we know that when we have a leaf or only one child we are in the first two cases which we can handle without doing this maximum value that. So, we can just walk out remove the 28 and promote the 21. So, this is exactly how delete works.

(Refer Slide Time: 16:37)

### Delete v

```

def delete(self,v):
    if self.isEmpty():
        return
    if v < self.value:
        self.left.delete(v)
        return
    if v > self.value:
        self.right.delete(v)
        return
    if v == self.value:
        if self.isleaf():
            self.makeempty()
        elif self.left.isEmpty():
            self.copyright()
        else:
            self.value = self.left.maxval()
            self.left.delete(self.left.maxval())
        return
    # Convert leaf to
    # empty node
def makeempty(self):
    self.value = None
    self.left = None
    self.right = None
    return

# Copy right child values
# to current node
def copyright(self):
    self.value =
    self.right.value
    self.left =
    self.right.left
    self.right =
    self.right.right
    return

```



We can now look at the function. If there is no value  $v$  then we just return the easy cases are when we do not find we are the current thing. So, if it is less than the current value then we go to the left and delete if it is bigger than the current value we go to the right and delete. So, the hard work comes then we actually find  $v$  at the current value. If this is a leaf now we have not shown, how write this function if this is a leaf this means that it has left and right child both as empty nodes if this is a leaf then we will make it empty we will see how we do this in a minute I will just show you the code for this.

So, if this is a leaf we delete it right this is the first case is simple case this is the leaf we just delete it and we make this node empty if on the other hand it has only **one** child, if. So, actually in this case if the left is empty then we just promote the right and if it is left is not empty then we will copy the maximum value from the left and delete the maximum value on the left.

We need to just see these two functions here make empty and copy right. So, make empty just says convert this into an empty node an empty node is one which all three fields are none. So, we will just say self dot value is none self dot left is none self dot right is none. If it had an empty node hanging off it those empty nodes are now disconnected from return.

This is this make empty function and now the copy right function just takes everything from the right and moves it up. It takes the right value and makes it the current value the left value right dot left and makes with the left. So, we just take basically this node and copy these values one by one here. So, we copy right dot value to the current value right dot left to the current left right dot right to the current right.

(Refer Slide Time: 18:36)

## Complexity

- All operations on search trees walk down a single path
- Worst-case: height of the tree
- Balanced trees: height is  $O(\log n)$  for  $n$  nodes
- Tree can be balanced using rotations — look up AVL trees

So, how much time do all these take well if you examined the thing carefully you would realize that in every case we are just walking down one path searching for the value and along that path either we find it or we go down to the end and then we stop. So, the complexity of every operation is actually written by the height of the tree if we have a balanced tree a balanced tree is one where you can define that each time we come to a node the left in the right child roughly have the same size.

If we have a balanced tree then it is not difficult to see then we have height logarithmic in  $\log n$  this is like a heap a heap is an example of a balanced tree now search tree will not be has nicely subset of heap because we will have some holes, but we will have a logarithmic height in general we will not explain how to balance a tree in this particular course we can look it up you can look for topic called AVL trees which is one variety of balanced trees which are balanced by rotating sub trees. So, it is possible while doing insert and delete to maintain balance at every node and ensure that all these operations are logarithmic.

Let us just look at the code directly and execute it and convenience ourselves that all the things that we wrote here actually work as intended.

(Refer Slide Time: 19:55)

```
#class Tree:
    # Empty node has self.value, self.left, self.right = None
    # Leaf has self.value != None, and self.left, self.right point to empty node

    # Constructor: create an empty node or a leaf node, depending on initval
    def __init__(self, initval=None):
        self.value = initval
        if self.value:
            self.left = Tree()
            self.right = Tree()
        else:
            self.left = None
            self.right = None
        return

    # Only empty node has value None
    def isempty(self):
        return (self.value == None)

    # Leaf nodes have both children empty
    def isleaf(self):
        return (self.left.isempty() and self.right.isempty())

    # Convert a leaf node to an empty node
=UU-----F1 searchtree.py Top L1 (Python) -----
```

Here we have a python code for the class tree that we showed in the lectures. So, we are just added a comment about how the empty node is organized and the leaves are organized. So, there is the constructor which sets up either an empty node or a leaf node with two empty children then we have isempty and isleaf we check whether the current value is none or both the left and right children are empty, respectively.

(Refer Slide Time: 20:26)

```
# Convert a leaf node to an empty node
def makeempty(self):
    self.value = None
    self.left = None
    self.right = None
    return

# Copy right child values to current node
def copyright(self):
    self.value = self.right.value
    self.left = self.right.left
    self.right = self.right.right
    return

# Check if value v occurs in tree
def find(self, v):
    if self.isempty():
        return(False)

    if self.value == v:
        return(True)

    if v < self.value:
        return(self.left.find(v))
=UU-----F1 searchtree.py 26% L24 (Python) -----
```

Then we have this function `makeempty` which converts the leaf to an empty node `copyright`, copies a right child values to the current node and then we have the basic recursive functions.

(Refer Slide Time: 20:39)

```

def copyright(self):
    self.value = self.right.value
    self.left = self.right.left
    self.right = self.right.right
    return

# Check if value v occurs in tree
def find(self,v):
    if self.isempty():
        return(False)

    if self.value == v:
        return(True)

    if v < self.value:
        return(self.left.find(v))

    if v > self.value:
        return(self.right.find(v))

# Insert value v in tree
def insert(self,v):
    if self.isempty():
        self.value = v
        self.left = Tree()
        self.right = Tree()

    if self.value == v:
        return

    if v < self.value:
        self.left.insert(v)
        return

    if v > self.value:
        self.right.insert(v)
        return

-UU-:----F1 searchtree.py 33% L45 (Python) -----

```

So, we start with `find`. So, `find` is the one which is `equivalent to` binary search then `insert` is like `find` and there it where it does not find it tries to insert.

(Refer Slide Time: 20:48)

```

        return(self.right.find(v))

# Insert value v in tree
def insert(self,v):
    if self.isempty():
        self.value = v
        self.left = Tree()
        self.right = Tree()

    if self.value == v:
        return

    if v < self.value:
        self.left.insert(v)
        return

    if v > self.value:
        self.right.insert(v)
        return

# Find maximum value in a nonempty tree
def maxval(self):
    if self.right.isempty():
        return(self.value)
    else:
-UU-:----F1 searchtree.py 48% L73 (Python) -----

```

And finally, we have `maxval` which we made for delete and delete now when we reach the situation where we are found a value to that needs to be deleted if it is a leaf then we

remove the leaf and make it empty if it is the left child is empty then we copy the right child up otherwise we delete the maximum from the left and promote that maximum value to a current.

(Refer Slide Time: 21:12)

```
        self.makempty()
    elif self.left.isempty():
        self.copyright()
    else:
        self.value = self.left.maxval()
        self.left.delete(self.left.maxval())
    return

# Inorder traversal
def inorder(self):
    if self.isempty():
        return []
    else:
        return(self.left.inorder()+[self.value]+self.right.inorder())

# Display Tree as a string
def __str__(self):
    return(str(self.inorder()))

-----
```

Finally, we have this inorder traversal which generates a sorted list from the tree values and the str function just displays the inorder traversal.

(Refer Slide Time: 21:22)

```
madhavan@dolphinair:...on-2016-jul/week7/python7 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 searchtree import *
>>> t = Tree()
>>> for i in [ 1 , 3 , 2, 18, 7, 5, 4, 22, 14]
...     File "<stdin>", line 1
...     for i = [ 1 , 3 , 2, 18, 7, 5, 4, 22, 14]
...         ^
SyntaxError: invalid syntax
>>> for i in [ 1 , 3 , 2, 18, 7, 5, 4, 22, 14]:
...     ...     t.insert(i)
...
>>> print(t)
[1, 2, 3, 4, 5, 7, 14, 18, 22]
>>> t.insert(17)
>>> print(t)
[1, 2, 3, 4, 5, 7, 14, 17, 18, 22]
>>> t.insert(4.5)
>>> print(t)
[1, 2, 3, 4, 4.5, 5, 7, 14, 17, 18, 22]
>>> t.delete(3)
>>> print(t)
[1, 2, 4, 4.5, 5, 7, 14, 17, 18, 22]
>>> t.delete(14)
>>> |
```

Now if we go to this then we can for instance import this package set up an empty tree and then put in some random values it is important not **put** in sorted order, otherwise a

sorted tree if you just insert one at a time it will just generate one long path. So, I am just trying to put it in some random order. We insert into the tree all these values and now we are print t give me a sorted version of this. So, 1, 2, 3, 4, I can now insert more values.

So, I can for random insert 17 and verify that now 17 is there before 14 and 18 and I can keep doing this, I can insert I can even insert values in between like 4.5 because I have not specified the integers right. So, it puts a between 4 and 5 and so on and now I can delete, **for example if I delete** 3 then I find that I have 1, 2, 4. If I delete, for example, 14 then I have no longer 14 between 7 and 17 and so on.

(Refer Slide Time: 22:45)

```
>>> print(t)
[1, 2, 4, 4.5, 5, 7, 17, 18, 22]
>>> █
```

This incrementation definitely works, although we have balanced it. If we do not balance it then the danger is that if we keep inserting a sorted sequence then we keep inserting larger values it keeps adding on the right child. So, the tree actually looks like a long path right. Then it becomes like a sorted list and every insert will take order n time, but if we do have rotations built in as we do, we could be using an AVL tree then we can ensure that the tree never grows to height more than  $\log n$ .

So, all the operations insert, find and delete they always be logarithmic respect to the number of values currently being **maintained**.

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

**Week - 08**  
**Lecture - 01**  
**Memoization and Dynamic Programming**

(Refer Slide Time: 00:02)

The slide has a light blue header with the title "Inductive definitions". Below the title, there are two lists of inductive definitions:

- \* Factorial
  - \*  $f(0) = 1$
  - \*  $f(n) = n \times f(n-1)$
- \* Insertion sort
  - \*  $\text{isort}([ ]) = [ ]$  ✓
  - \*  $\text{isort}([x_1, x_2, \dots, x_n]) = \text{insert}(x_1, \text{isort}([x_2, \dots, x_n]))$

Handwritten annotations are present: a blue circle around the base case of insertion sort, and blue arrows pointing from the recursive call in the insertion sort definition to the corresponding part of the insertion sort formula.

We saw earlier that inductive definitions often provide a nice way to get a hold of the functions we want to compute. Now we are familiar with induction for numbers. For instance, we can define the factorial function in terms of the base case  $f$  of 0, and in terms of the inductive case saying  $f$  of  $n$  is  $n$  times factorial of  $n$  minus 1. What we saw is we can also define inductively functions on structures like lists, so for instance, we can take as the base case **an** empty list, and in the inductive case, we can separate the task of sorting of list into doing something with the initial element and something with the rest.

Insertion sort can be defined in terms of insert function as follows. So,  $\text{isort}$  of the base case for the empty case just gives us the empty list. And then if you want to sort a list with  $n$  elements, we pull out the first element right and then we insert it into the result of inductively sorting the rest. This is a very attractive way of describing the dependency of the function that we want to compute the value that we are trying to compute on smaller values, and it gives us a handle on how to go about computing **it**.

(Refer Slide Time: 01:12)

## ... Recursive programs

```
def factorial(n):
    if n <= 0:
        return(1)
    else:
        return(n*factorial(n-1))
```

The main benefit of an inductive definition is that it directly yields a recursive program. So, we saw this kind of a program for factorial which almost directly follows a definition saying that f of 0 is 1 and f of n is n into f n minus 1. So, we can just directly read it talk more or less and translate it. The only thing we have done is we have taken care of some error case, where if somebody feeds a negative number, we will still say 1, and not go into a loop.

(Refer Slide Time: 01:40)

## Sub problems

- \* factorial(n-1) is a **subproblem** of factorial(n)
  - \* So are factorial(n-2), factorial(n-3), ..., factorial(0)
- \* isort([x<sub>2</sub>,...,x<sub>n</sub>]) is a **subproblem** of isort([x<sub>1</sub>,x<sub>2</sub>,...,x<sub>n</sub>])
  - \* So is isort([x<sub>i</sub>,...,x<sub>j</sub>]) for any 1 ≤ i ≤ j ≤ n
- \* Solution of f(y) can be derived by combining solutions to subproblems

In general, when we have such inductive definitions, what we do is we have sub problems that we have to solve in order to get to the answer we are trying to reach. So, for instance, to compute factorial of n, one of the things we need to do is compute factorial of n minus 1. So, we call factorial of n minus 1 as sub problem of factorial n.

Now in turn factorial of n minus 1 requires us to compute factorial n minus 2, so actually if you go down the chain, the factorial n sub problems are all the factorials for values smaller than n. Similarly, for insertion sort, in order to sort the full list, we need to sort all the elements excluding the first one what is called the tail of the list, and in turn we need to sort its tail and so on.

In general, when we do insertion sort we will find that we need to sort things a segment of the list. So, we can in general talk about  $x_i$  to  $x_j$ . And in all these cases, what the inductive definition tells us is how to compute the actual value of f for our given input y by combining the solutions to **these** sub problems; for instance, in factorial we combine it by multiplying the current input with the result of solving it for the next smaller input. For insertion sort, we combine it by inserting the first value into the result of solving the smaller input that is the tail of the list.

(Refer Slide Time: 03:05)

## Evaluating subproblems

0,1,2,3,5

Fibonacci numbers

- \*  $\text{fib}(0) = 0$
- \*  $\text{fib}(1) = 1$
- \*  $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

def fib(n):  
 if n == 0 or n == 1:  
 value = n  
 else:  
 value = fib(n-1) +  
 fib(n-2)  
 return(value)

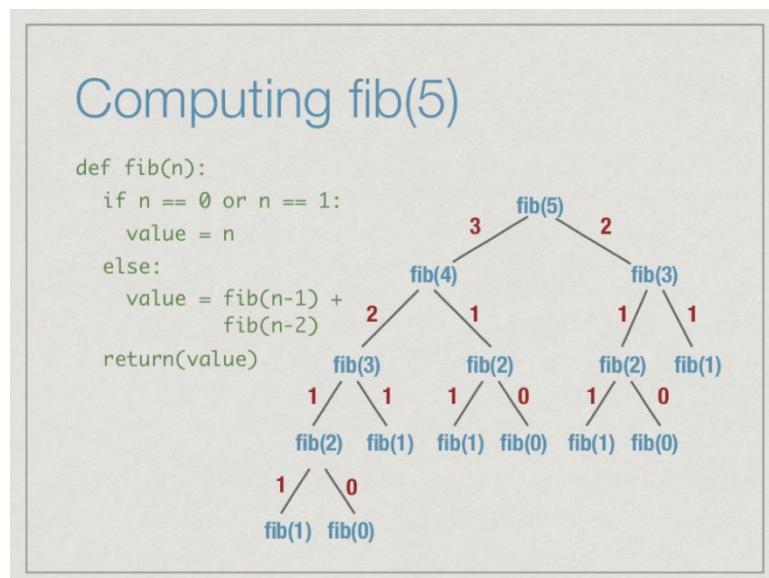
Let us look at one particular problem, which will highlight an issue that we have to deal with when we are looking at inductive specifications and naively translating them into programs. The Fibonacci numbers are a very famous sequence which were invented by

Fibonacci, and they occur in nature and they are very intuitive and most of you would have seen them. The Fibonacci numbers are 0, 1 and then you add. So, 1 plus 0 is 1, 1 plus 1 is 2, 3, 5 and so on.

So, you just keep adding the previous two numbers and you go on. The inductive definition says that 0th Fibonacci number is 0; the first Fibonacci number is 1; and after that for two onwards, the nth Fibonacci number is obtained by sub adding the previous two. The Fibonacci number 2 is Fibonacci 1 plus Fibonacci 0. As before we can directly translate this into an inductive into a recursive program. We can just write a python function fib which says if n is 0 or n is 1, you **return** the value n itself. So, if n is 0 return 0, if n is 1, we return 1.

Otherwise, you compute the value by recursive to recursively calling Fibonacci on n minus 1 and n minus 2, add these two and return this value. Here is the clear case of an inductive definition that has a natural recursive program extracted from it.

(Refer Slide Time: 04:27)



Let us try to compute a value and see what happens. So, supposing we want to compute Fibonacci of 5 using this definition. So, Fibonacci of 5, we will go into the else clause and say we need to compute Fibonacci of n minus 1 namely 4 and n minus 2 namely 3. As Fibonacci of 5, leaves us with two problems to compute, Fibonacci of 4 and Fibonacci of 3. So, we do these in some order; let us go left to right. So, we pick

Fibonacci of 4, and this in turn will require us to compute Fibonacci of 3 and Fibonacci of 2 by just applying the same definition to this value.

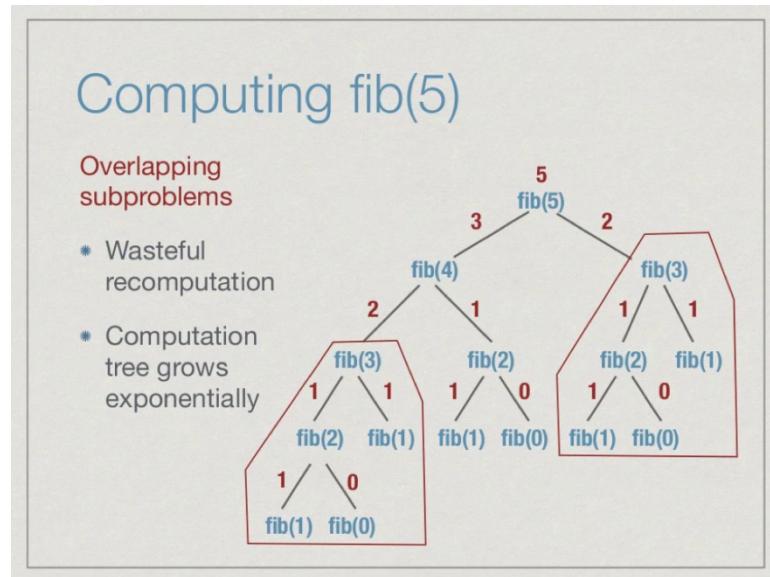
Similarly, we go to the left of the two sub problems Fibonacci of 3 requires 2 and 1; 2 requires 1 and 0. Now for 1 and 0 fortunately we can **exit** without making a recursive call; if n is equal to 0 or n is equal to 1, we just return the value. So, we get back Fibonacci 1 as 1 and Fibonacci 0 as 0. So, with this, we can complete the computation of Fibonacci 2, we get value is 1 plus 0 in other words 1. So, Fibonacci of 2 is 1.

Now, we are back to Fibonacci of 3. So, we have computed for Fibonacci of 3, the left case Fibonacci of 2. Now we have to compute the right case. And once again we find the Fibonacci of 1 **being** a base case it gives us 1, and now we can combine this and get Fibonacci of 3 is 2.

Now, we are back to Fibonacci of 4. And we have computed the left side of Fibonacci of 4. So, we **need** to compute the right side. And now what happens is we end up having to compute Fibonacci of 2 again, even though we already know the value. We naively have to execute Fibonacci of 2 call 1 and 0, again propagate the values 1 and 0 back up add them up and get 1. Now, we can compute Fibonacci of 4 is 2 plus 1 3.

And now we are finally, back to the original call where we had to compute Fibonacci of 4 and Fibonacci of 3. So, we **are** done with 4. Now, we want to do Fibonacci of 3. Notice that we have already computed Fibonacci of 3, but this will blindly require us to call this function again. So, we will again have to execute this full tree, go all the way down, go all the way up and eventually Fibonacci of 3 will of course, give us the same answer namely 2, which we already knew, but we would not take exploit or we would not take advantage of the fact that we knew **it**. And in this way, we get 3 plus 2 and therefore, Fibonacci of 5 is 5.

(Refer Slide Time: 06:45)



The point to note in this is that we are doing many things again and again. In this particular computation, the largest thing that we repeat is Fibonacci of 3. So, as a result of this re-computation of the same value again and again, though we in principle only need  $n$  minus 1 sub problems.

If we have fib of 5, we need to fib of 4, fib of 3, fib of 2 and so on. N subproblems, if we don't include fib of 0, but some of these sub problems like in this case Fibonacci of 3, we compute repeatedly in a wasteful way. As a result, we end up solving an exponential number of solve sub problems even though there are only order  $n$  actual problems to be solved.

(Refer Slide Time: 07:26)

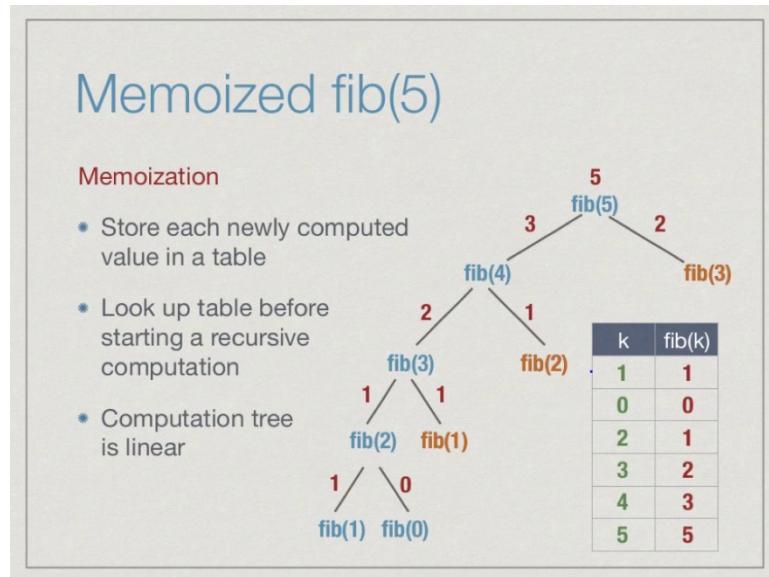
## Never re-evaluate a subproblem

- \* Build a table of values already computed
  - \* Memory table
  - \* Memoization
  - \* Remind yourself that this value has already been seen before

So, what we want to do is move away from this naive recursive implementation of an inductive definition, and try to work towards never reevaluating a sub problem. This is easy to do, if we could only remember the sub problems that we have solved before, then all we have to do is look up the value we already computed rather than recompute it. So, what we need is a kind of a table, a table where we store the values we have computed and before we go and compute a value, we first check the table. If the table has an answer, we take the table's answer and go ahead.

If the table does not have an answer then we apply the recursive definition compute it, and then we add it to the table. This table is normally called a memory table to memorize; and from this, we get this word memoization. It is actually memo and not memorization; memoization in the sense of write yourself a memo, memo is like a reminder, write yourself a reminder that this has been done before. Memoization is the process by which when we are computing a recursive function, we compute the values one at a time, and as we compute them we store them in a table and look up the table before we recompute any.

(Refer Slide Time: 08:41)



Here is how a computation of Fibonacci of 5 would go, if we keep a table. This is our table here right. So, we have a table where in some order, it does not really matter for now; in some order as and when we find Fibonacci of k for some k, we just record it. And notice that this table is empty, even though we know the base case of Fibonacci of 0 and Fibonacci of 1 are 0 and 1 respectively, we do not assume you know it, because it will come out as the first time we hit the base case it will come out of the recursive definition.

Let us see how it goes right. So, we start Fibonacci of 5 as usual, it says do 4 and 3, 4 says do 3 and 2, 3 says do 2 and 1, 2 says do 1 and 0. And now from our basic case, the base case in the function, we will get back that fib of 1 is 1. So, we store this in the table, this is the first value we have actually computed. Notice we did not assume we knew it, when we came to it in the base case; we put it into the table. Same way, fib of 0 is 0 we did not know it before; we put it in the table.

Now we come up and we realize that fib of 2 is now available to us, it is 1 plus 0 is 1. So, we store that in the table. We say for k equal to 2, fib of k is 1. Now we come back to fib of 3, and now we go down and it asks us to compute fib of 1 again. Now although this does not take us any work, because it is a base case we do not actually exploit that fact. We first look in the table and say is there an entry for k equal to 1, yes there is and so we pick it up.

We highlight in orange the fact that this value was actually not recomputed, but looked up in the table, so from 1 plus 1, we now have Fibonacci of 3 is 2. Now we go back up to Fibonacci of 4, and it asks us to compute the second half of its sub problems, namely Fibonacci of 2. Once again we find that there is an entry for 2 in our table. So, we mark it in orange, and we just take the value from the table without expanding and computing the tree again as we had done before when we did the naive computation. Now, 2 plus 1 is 3, so we have Fibonacci of 4.

So, we have to now go back and compute the other branch Fibonacci of 3, but once again 3 has an argument k is in our table. So, we have an entry here for 3. So, we can just look up Fibonacci of 3 and say oh, it is two. So, once again we mark it in orange. And so now, we have Fibonacci of 5 is 3 plus 2 and that is 5, and then now this is a new value, so we enter that.

Notice therefore, that every value we computed, we expanded the tree or even looked up the base case only once according to the function definition. Every subsequent time, we needed a value we just looked it up in the table, and we can see that the table grows exactly as many times as much as there are sub problems to be solved and we never solved a sub problem twice in the sense of computing it twice. We solved it by looking at the table.

(Refer Slide Time: 11:32)

### Memoized fibonacci

```
def fib(n):
    if fibtable[n]:
        return(fibtable[n])
    if n == 0 or n == 1:
        value = n
    else:
        value = fib(n-1) + fib(n-2)
    fibtable[n] = value ✓
    return(value)
```

This is a very easy step to incorporate into our Fibonacci table, the Fibonacci functions. So, we just add this red code. The green lines are those ones we have already had before. Now what Fibonacci says is, the first thing you do when you get a number is try and look up the table. If there is a value Fibonacci of n, which is defined then return that value; otherwise, we go through the recursive computation. This is the usual computation which will make a recursive call and eventually come up with a new value which is the value for this particular n. So, before we return this value back as a result of this function, we store it in the table.

Henceforth, if this value n is ever invoked again, we never have to look up the thing we never have to compute it; it will be in the table right. It is very simple as we said when you get an argument n for which you want to compute the function, you first check the table, if it is there in the table you do not do anything more you just return the table value. If it is not in the table, you apply your recursive definition to compute it, just like you would normally. Having computed it you first store it in the table, so that future accesses to this function will work without having to do this recursion and then you return the value you got.

(Refer Slide Time: 12:49)

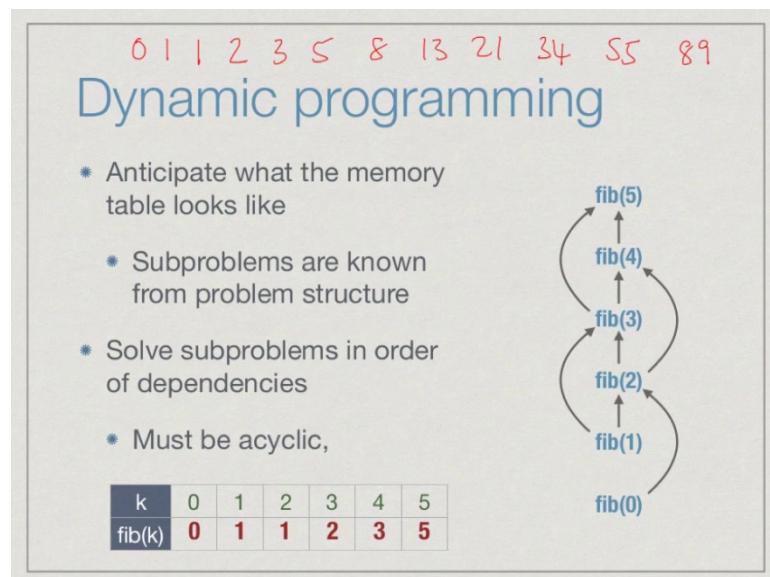
```
In general  
function f(x,y,z):  
    if ftable[x][y][z]:  
        return(ftable[x][y][z])  
    value = expression in terms of  
           subproblems  
    ftable[x][y][z] = value  
    return(value)
```

This can work for any combination of arguments. You just need a table, in python terms; you just need a dictionary for every combination of arguments if that value has been computed before that key will be there in our table. This table in general in python would

be a dictionary and not a list because the arguments could be any particular values. They could some could be string, some could be numbers, they need not be continuous that is all.

We basically for given the particular combination of arguments, we look up whether that combination of keys is there in the dictionary if so we look it up and return it. Otherwise, we compute a new value for this combination, store it and then return it. So, we have glossed over little few things for instance typically where if you want to really write this properly in python way we have to use some exceptions and all that, but this is more or less the skeleton of what we need to do.

(Refer Slide Time: 13:42)



This brings us to the main topic that we want to do this week in a couple of lectures which is called dynamic programming. So, dynamic programming is just a strategy to further optimize this memoized recursion. This memoized recursion tells us that we will store values into a table as and when they are computed. Now it is very clear that in an inductive definition, we need to get to the base case and then work ourselves backwards up from the base case, to the case we have at hand. That means, there are always some values some base values for which no further values need to be computed; these values are automatically available to us.

We have some problems which have sub problems, and some other problems which have no sub problems. If a problem has a sub problem, we cannot get the problem, we cannot

get Fibonacci of five unless we solve its sub problems, Fibonacci 4 and 3. But if we have a base case that Fibonacci of 1 or Fibonacci of 0, we do not have any sub problems, so we can solve them directly. This is a kind of dependency. So, we have to solve the sub problems in the dependency order, we cannot get something which is dependent on something else until that something else has been solved, but a little thought tells us that this dependency order must be acyclic, we cannot have a **cycle of dependency**.

So, **if** one value it is like 5 depends on 3, 3 depends on 1, and 1 again depends on 5, because there will be no way to actually resolve this right. There will always be a starting point, and we can solve the sub problems directly in the order of the dependencies instead of going through the recursive calls. We do not have to follow the inductive structure, we can directly say ok, tell me which are all the sub problems which do not need anything solve them, which are all the ones which depend only on these solve them and so on. This **gives** as an iterative way.

If we look at the sub problem for Fibonacci of 5, for example, it says that it requires all these sub problem **but** the dependency is very straightforward; 5 depends on 4 and **3**; 4 depends on 3 and 2; 3 depends on 2 and 1; 2 depends on 1 and 0; and 0 and 1 have no dependencies. So, we can start at the bottom, and then work ourselves up, we can say Fibonacci of 0, **needs** no dependencies, so let me write a value for it. Fibonacci of 1 needs no dependency, so let me write a value for it.

Now we see that for Fibonacci of 2 both the things that it needs have been computed. So, I can write fib 2 in the table directly without even going to it from 5, I am not coming down from 5, we are just directly filling up the table, just keeping track of which values depend on which values. So, assuming that we know the function, we can calculate this dependency and just compute that values as and when the dependencies are satisfied.

Now, we have 1 and 2. So, we can compute Fibonacci of 3. So, we just compute it. We have 3 and 4, so we can compute I mean 2 and three. So, we can compute Fibonacci of 4, so we compute. And finally, we have 2 and fib 3 and fib 4, so we can get fib 5. This value as you can see becomes now a linear computation, I can just walk up from 0 to 5 and fill in the values. In fact, this is what we do by hand.

When I can you ask me for the tenth Fibonacci number, I can write it out. I can say 0. 0 plus 1 is 1, and then 1 plus 1 is 2, 2 plus 1 is 3, 5, 8, 13, 21, 34, so clearly it is not a very

complicated process as it seems to be when we have this exponential recursion. I can do it on the fly more or less, because all I have to do is keep generating the values in a sequence, and this is the virtue of dynamic programming. It converts this recursion into a kind of iterative process, where you fill up the values that you would normally fill up in the MEM table by recursion to fill them up iteratively starting with the ones which have no dependencies.

(Refer Slide Time: 17:25)

## Dynamic programming fibonacci

```
def fib(n):
    fibtable[0] = 0
    fibtable[1] = 1
    for i in range(2,n+1):
        fibtable[i] = fibtable[i-1] +
                      fibtable[i-2]

    return(fibtable[n])
```

Then the dynamic programming version of Fibonacci is just this iterative blue. So, it says that you start from with value 0 and 1 to be the values themselves what we earlier said was that if n is 0 or 1 then you return n. So, here we just store it into the table directly; we say fib table of 0 is 0, fib table of 1 is 1.

And then we walk from 2 to n, so in python notation the range end is n plus 1; and at each stage, we just take the ith value to be the sum of the i minus 1, i minus 2 values which we have already computed because we are going in this particular order because we have recognized the dependency order goes from 0 to n. And finally, the answer we need is the nth entry, so we just return that.

(Refer Slide Time: 18:10)

## Summary

**Memoization**

- \* Store values of subproblems in a table
- \* Look up the table before making a recursive call

**Dynamic programming:**

- \* Solve subproblems in order of dependency
  - \* Dependencies must be acyclic
- \* Iterative evaluation

To summarize, the basic idea to make naïve recursion more efficient is to never compute something twice. And to never compute something twice, we store the values we compute in what we call a memo table this is called **memoization**. And we always look up the table before we make a recursive call.

Or this can be further optimized, so we avoid making recursive calls altogether and we just directly fill in the table in dependency order which must be acyclic otherwise this sub problem will not be solvable. And this converts the recursive evaluation into an iterative evaluation, which is often much more efficient.

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

**Week - 08**  
**Lecture - 02**  
**Grid Paths**

(Refer Slide Time: 00:02)

### Grid Paths

- Roads arranged in a rectangular grid
- Can only go up or right
- How many different routes from  $(0,0)$  to  $(m,n)$ ?



In the last lecture we looked at how to make iterative or inductive definitions more efficient than naïve recursion, and we saw memoization and dynamic programming as tools to do this.

Now, let us look at a typical problem and see how we can apply this technique. So, here is a problem of grid paths. So, we have a grid here, you can imagine there are roads which are arranged in a rectangular format. We can imagine that the intersections are numbered. So, we have  $(0, 0)$  at the bottom left corner and in this case, we have  $(5, 10)$  because going across from left to right we have 1, 2, 3, 4, 5 different intersections and 10 going up. So, we have at  $(5, 10)$  the top right corner.

If these are roads the constraint that we have is that one can only travel up or right. So, you can go up a road or you can go right, but you cannot come down. This is not allowed. These are one way roads which goes up and right, and what we want to ask is how many ways there are to go from the bottom left corner to the top right corner. So,

we want to count the number of what are called grid paths. So, a grid path is one which follows this right. So, we want to know how many such different paths are there which take us from  $(0, 0)$  to  $(5, 10)$  only going up or right.

(Refer Slide Time: 01:26)

## Grid Paths

- Roads arranged in a rectangular grid
- Can only go up or right
- How many different routes from  $(0,0)$  to  $(m,n)$ ?

(0,0) (5,10)

So, here is one path drawn in blue.

(Refer Slide Time: 01:32)

## Grid Paths

- Roads arranged in a rectangular grid
- Can only go up or right
- How many different routes from  $(0,0)$  to  $(m,n)$ ?

(0,0) (5,10)

Here is a different path drawn in red and notice that these 2 paths actually, start in different directions from the first point and they never meet except with the target. They do not overlap at all. On the other hand we could have paths which overlap. This yellow

path overlaps a part of its way with the blue path in this section and it also overlaps with the red path in 2 portions. There are many different ways in which we can choose to make this up and right moves and the question is, how many total such different paths are there?

(Refer Slide Time: 02:05)

## Combinatorial solution

- Every path from (0,0) to (5,10) has 15 segments
  - In general  $m+n$  segments from (0,0) to (m,n)
  - Of these exactly 5 are right moves, 10 are up moves
  - Fix the positions of the 5 right moves among the overall 15 positions
  - $15 \text{ choose } 5 = (15!)/(10!(5!)) = 3003$
  - Same as  $15 \text{ choose } 10$ : fix the 10 up moves

$$\frac{n!}{k!(n-k)!}$$

There is a very standard and elegant combinatorial solution. So, one way of thinking about this is just to determine, how many moves we have to make. We have to go from 0 to 5 in one direction and 0 to 10 in the other direction. So, we have to make a total number of 5 horizontal moves and 10 vertical moves, in other words every path no matter which direction we started and which move, which choice of moves we make must make 15 steps and of these 5 must be horizontal steps and 10 must be vertical steps, because they all take us from (0, 0) to (5, 10).

So, all we have to do since we know that these 5 steps are horizontal and 10 are vertical is to just demarcate which ones are horizontal and which are vertical. Now once we know which ones are horizontal we know what sequence they come in because the first horizontal step takes us from column 0 to column 1, second 1 takes us from one to 2. So, we cannot do it in any order other than that.

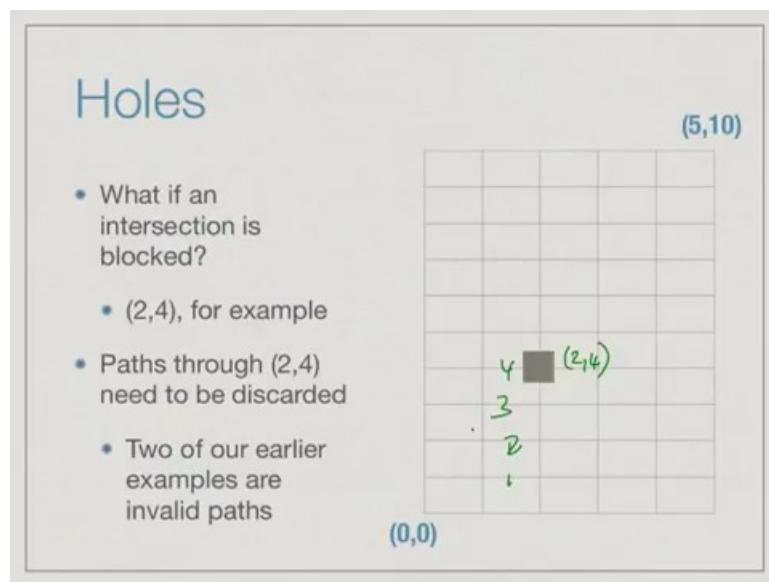
So, we have in other words we have 15 slots, where we can make moves and then we just say first we make an up move, then we make a right move then we make an up move then make another up move and so on. So, every path can be drawn out like this as 10 up

moves and 5 right moves and if we fix the 5 right moves then automatically all the remaining slots must be 10 up moves or conversely.

It is either 15 choose 5, it is the way of choosing 5 positions to make the right move out of the 15, and it turns out that the definition of 15 choose 5 is clearly the same as 15 choose 10 because we could also fix the 10 up moves and the definition is basically... if you know the definitions... then  $n$  choose  $k$  is  $n$  factorial by  $k$  factorial into  $n$  minus  $k$  factorial.

This  $k$  and  $n$  minus  $k$  basically says that 15 minus 5 is 10. So, we get a symmetric function in terms of  $k$  and  $n$  minus  $k$ . In this case we can apply this formula if you would like to call it that and directly get that the answer is 3003. There does not appear to be much to compute other than writing out large factorials and then seeing what the number comes.

(Refer Slide Time: 04:11)



But the problem becomes more interesting, if we constrain it by saying that some of these intersections are blocked for instance, supposing there is some road work going on and we cannot go through this intersection (2, 4). This is the intersection 2 comma 4 second column and the fourth row counting from below. It's actually 2 comma 3, but 1, 2, 3, 4 yeah 2 comma 4. Now, if we cannot go through this then any path which goes through this particular block intersection should no longer be counted. Out to those 3003 some paths are no longer valid paths.

(Refer Slide Time: 04:49)

## Holes

- What if an intersection is blocked?
  - (2,4), for example
- Paths through (2,4) need to be discarded
- Two of our earlier examples are invalid paths

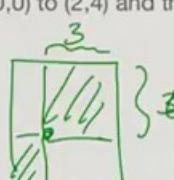


(0,0) (5,10)

For instance, in the earlier thing the blue path that we had drawn actually goes through this, the red path does not, where the yellow path overlapped with the blue path unfortunately in this bad section. It also passes through this. There are some paths which are allowed from the 3003 and some which are not. So, how do we determine how many paths survived this kind of block.

(Refer Slide Time: 05:09)

## Combinatorial solution

- Every path through (2,4) goes from (0,0) to (2,4) and then from (2,4) to (5,10)
  - Count these separately:
    - $(4+2)$  choose  $2 = 15$
    - $(6+3)$  choose  $3 = 84$
  - Multiply to get all paths through (2,4): 1260
  - Subtract from 15 choose 5 = 3003 to get valid paths that avoid (2,4): 1743

So, again we can use a combinatorial argument in order to be blocked a path must go to (2, 4) and then from (2, 4) to (5, 5). If we could only count how many paths go from (0,

0) to (2, 4) and then how many paths go from (2, 4) to (5, 10), these are all the bad paths. So, we can count these bad paths and subtract them from the good paths. How do we count the bad paths well we can just solve a smaller version of the problem. So, we have an intermediate target.

So, we solve this grid how many paths go from here to here, how many paths go from here to here. So, from (0, 0) to (2, 4) we get 4 plus 2 remember it 10 plus 5 it was a curve or get, 10; 4 plus 2 choose 2. So, we get 15 and from here to here the difference is that we have to do in both directions 3 and so, we have to go sorry we have to go up 6 and we have to go right 3, we are at (2, 4). So, we have to go from 4 to 10 and from 2 to 5.

So, we have 6 plus 3 choose 3, 84 ways of going from (2, 4) to this and each of the ways in the bottom, can be combined with a way on the top. So, we multiply this and we get 1260 paths which pass through this bad intersection, we subtract this from the original number 3003 and we get 1743 paths which remain. So, a combinatorial approach still works.

(Refer Slide Time: 06:32)

## Holes

- What if two intersections are blocked?
- Subtract paths through (2,4), (4,4)
  - Some paths are counted twice!
- Add back paths through both holes
- Inclusion-exclusion: messy

The diagram shows a 10x10 grid with a path from (0,0) to (5,10). Two intersections, (2,4) and (4,4), are marked as 'holes' with grey squares. A blue path goes from (0,0) to (2,4) and then to (5,10). A yellow path goes from (0,0) to (4,4) and then to (5,10). A red path goes from (0,0) to (4,4) and then to (5,10), bypassing the first hole. Other paths are shown in orange and green, some being counted twice due to the holes.

Now, what happens if we put 2 such intersections? So, we will you can do the same thing we can count all the parts which get blocked because of the first intersection, we can count all the paths which pass through in this case (4, 4) is the second intersection which has been blocked. So, we can count all these parts which pass through (4, 4). This we

know how to do: we just computed it for (2, 4), but the problem is that there are some paths like the yellow paths which pass through both (2, 4) and (4, 4).

So, we need a third count we need to count paths which pass through both of these and make sure we do not double count them. So, one way is that we just add these back. This is something which is called **in combinatorics** inclusion and exclusion. So, when we have these overlapping exclusions, then we have to count the overlaps and include them back. We have to keep doing this step by step. If we have 3 holes we get an even more complicated inclusion exclusion formula and it rapidly becomes very complicated even to calculate the formula that we need to get. Is there a simpler way to do this?

(Refer Slide Time: 07:37)

## Inductive formulation

- How can a path reach  $(i,j)$ 
  - Move up from  $(i,j-1)$
  - Move right from  $(i-1,j)$
  - Every path to these neighbours extends in a unique way to  $(i,j)$

The diagram shows a grid point labeled  $(i,j)$ . Two arrows point to its neighbors: one from the left labeled  $(i-1,j)$  and one from below labeled  $(i,j-1)$ .

Let us look at the inductive structure of the problem, suppose we say we want to get in one step to intersection  $(i, j)$ . How can we reach this in one step since our roads only go left to right and bottom to top, the only way we can reach  $(i, j)$  is by taking a right edge from  $i$ 's left neighbor. So, we can go from  $(i-1, j)$  to  $(i, j)$  or we can go from below from  $(i, j-1)$  to  $(i, j)$ . Notice that if a path comes from the left it must be different from a path that comes from below. So, every path that comes from the left is different from every path that comes from below. So, we can just add these up.

(Refer Slide Time: 08:20)

## Inductive formulation

- $\text{Paths}(i,j)$  : Number of paths from  $(0,0)$  to  $(i,j)$
- $\text{Paths}(i,j) = \text{Paths}(i-1,j) + \text{Paths}(i,j-1)$
- Boundary cases
  - $\text{Paths}(i,0) = \text{Paths}(i-1,0)$  # Bottom row
  - $\text{Paths}(0,j) = \text{Paths}(0,j-1)$  # Left column
  - $\text{Paths}(0,0) = 1$  # Base case

In other words if we say that  $\text{paths}(i, j)$  is the quantity we want to compute, we want to count the number of paths from  $(0, 0)$  to  $(i, j)$ . These paths must break up into 2 disjoint sets those which come from the left which recursively or inductively if you prefer to say is exactly the quantity  $\text{paths}(i-1, j)$ . How many paths are there which reach  $(i-1, j)$  every one of these paths can be extended by a right edge to reach  $(i, j)$  and, they will all be different similarly  $\text{paths}(i, j-1)$  are all those paths which come from below, because they all reach the point just below  $(i, j)$  from there each of them will be **extended** in a unique way to  $(i, j)$ .

This gives us our simple inductive formula,  $\text{paths}(i, j)$  is just **the sum of  $\text{paths}(i-1, j)$  and  $\text{paths}(i, j-1)$** . Then we need to of course, investigate the base cases: in this case the real base case is just  $\text{paths}(0, 0)$ : in how many ways can I go from  $(0, 0)$  and just stay in  $(0, 0)$ ? Well there is **only** one way, it is tempting to say 0 ways, but it is not 0 ways its one way otherwise nothing will happen. So, we have one way by just doing nothing to stay in  $(0, 0)$  and if we are now moving along the left column, if you are moving along the left column then there are no paths coming from its left because we are already on the leftmost column.

So, all the paths to  $(0, j)$  must be extensions of  $\text{paths}$  which have come from below up to  $(0, j-1)$ . Similarly if you are on the bottom row there is no way to come from below

because we are already on the lowest set of roads. So,  $\text{paths}(i, 0)$  can only come from the left, from  $\text{paths}(i-1, 0)$ .

(Refer Slide Time: 09:56)

## Dealing with holes

- $\text{Paths}(i,j) = 0$ , if there is a hole at  $(i,j)$
- $\text{Paths}(i,j) = \text{Paths}(i-1,j) + \text{Paths}(i,j-1)$ , otherwise
- Boundary cases
  - $\text{Paths}(i,0) = \text{Paths}(i-1,0)$  # Bottom row
  - $\text{Paths}(0,j) = \text{Paths}(0,j-1)$  # Left column
  - $\text{Paths}(0,0) = 1$  # Base case

This gives us a direct way to actually compute this even with holes because, the only difference now is that if, there is a hole we just declare that no paths can reach that place. So, we just add an extra clause which says  $\text{paths}(i, j)$  is 0 if there is a hole at  $(i,j)$ ; otherwise we use exactly the same inductive formulation and now what happens is, if I have a hole below me, if I have a hole below me, no paths can come from that direction because by definition  $\text{paths}(i, j)$  at that point is 0.

(Refer Slide Time: 10:29)

## Computing Paths(i,j)

- Naive recursion will recompute multiple times
  - Paths(5,10) requires Paths(4,10) and Paths(5,9)
  - Both Paths(4,10) and Paths(5,9) require Paths(4,9)
- Use memoization ...
  - ... or compute the subproblems directly in a suitable way

So, once again if we now apply this and do this using the standard translation from the inductive definition to a recursive program, we will find that we will wastefully recompute the same quantity multiple times for instance `paths(5,10)`. If we have `paths(5, 10)`, it will require me to compute this and this.

These are the 2 sub problems for `paths(5, 10)`, namely `(4, 10)` and `(5, 9)` but, in turn in order to compute `(4, 10)` I will have to compute whatever is to its left and below it and in order to compute `(5, 9)` I will also have to compute what is to its left and below it and now what we find is that this quantity namely `(4, 9)` is computed twice, once because of the left neighbor of `(5, 10)` and once because of the neighbor below `(5, 10)`.

So, as we saw before we could use memoization to make sure that we never compute `(i,j)` twice by storing a table  $i$  comma  $j$ , and every time we compute a new value for  $i$  comma  $j$  we store it in the table and every time we look up, we need to compute one we first check the table, if it is already there we look it up, otherwise we will compute it and store it, but since we know there is a table and we know what the table structure is basically it is all entries of the form  $i$  comma  $j$ . We can also see if we can fill up this table iteratively by just examining the sub problems in terms of their dependencies.

(Refer Slide Time: 12:01)

## Dynamic programming

(5,10)

- Identify dependency structure
- Paths(0,0) has no dependencies
- Start at (0,0)

In general a node the value depends on things to its left and below. If there are no dependencies, it must have nothing to its left and nothing below and there is only one such point namely (0, 0). This is the only point which is the base case which has nothing to its left and nothing below so its value is directly read. So, we start from here.

(Refer Slide Time: 12:24)

## Dynamic programming

- Start at (0,0)
- Fill row by row

|   |    |    |     |     |      |
|---|----|----|-----|-----|------|
| 1 | 11 | 51 | 181 | 526 | 1363 |
| 1 | 10 | 40 | 130 | 345 | 837  |
| 1 | 9  | 30 | 90  | 215 | 492  |
| 1 | 8  | 21 | 60  | 125 | 272  |
| 1 | 7  | 13 | 39  | 65  | 147  |
| 1 | 6  | 6  | 26  | 26  | 82   |
| 1 | 5  | 0  | 20  | 0   | 56   |
| 1 | 4  | 10 | 20  | 35  | 56   |
| 1 | 3  | 6  | 10  | 15  | 21   |
| 1 | 2  | 3  | 4   | 5   | 6    |
| 1 | 1  | 1  | 1   | 1   | 1    |

Remember that the base value at (0, 0) is one, and now once we have done this it turns out: you remember the road dependency, it said  $(i, 0)$  is  $(i-1, 0)$ . So, we can fill up this, because this has only one dependency which is known now. In this way I can fill up the

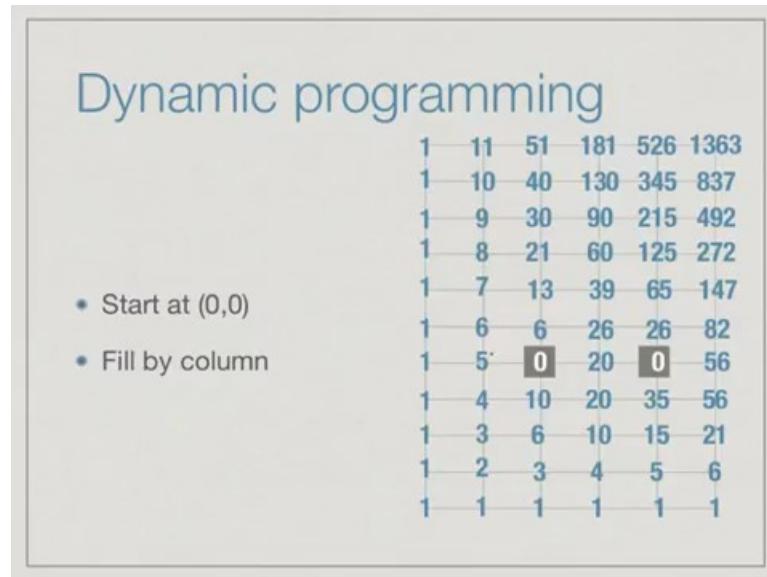
entire row and say that all along this row there is only one path namely the path that starts going right and keeps going right. Now we can go up and see that this thing is also known because, it also depends only on the value below it and once that is known then these 2 are known.

So I can add them up; remember the value at any position is just the value to its left plus the value to its bottom and now I start to get some non trivial values, and in this way I can fill up this table row by row and at each point when I come to something I will get the fact with the dependency unknown. The next row looks like this and the next row. Now we come to the row with holes. So, for the row with holes, wherever we hit a hole instead of writing the value that we would normally get by adding its left and bottom neighbour we deliberately put a 0 because; that means, that no path is actually allowed propagating through that row.

Now, when we come to the next row, the holes will automatically block the paths coming from the wrong direction. So, here for instance we have only 6 paths coming from the left because we have no paths coming from below similarly we have 26 paths coming from the left and no paths coming from below. This is how our inductive definition neatly allows us to deal with holes and from that inductive definition we recognize the dependency structure and we imagine the memo table and now we are filling up this memo table row by row so that at every point when we reach an  $(i, j)$  value its dependent values are already known.

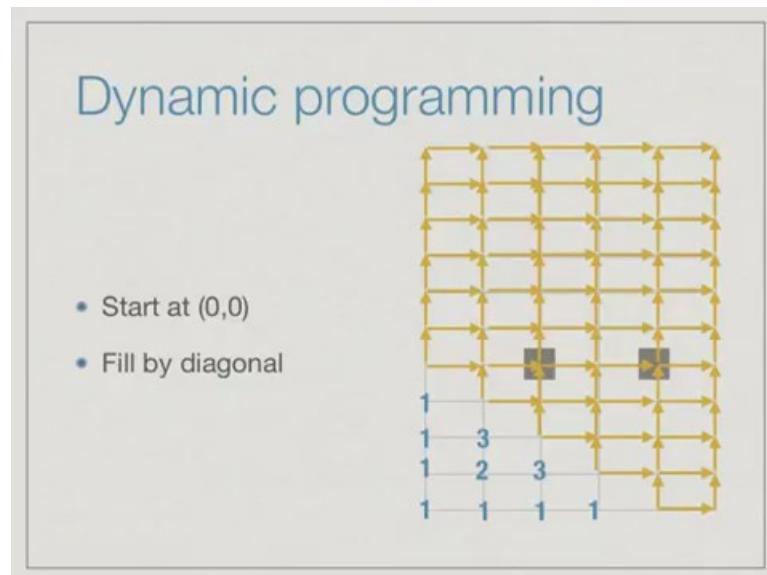
So, we can continue doing this row by row, and eventually we find look there are 1363 paths which avoid these two.

(Refer Slide Time: 14:18)



So, we could also do the same thing in a different way instead of doing the bottom row, we can do the left column and the same logic says, that we can go all the way up then we can start in the second column, go all the way up and do this column by column and not unexpectedly, we should get the same answer. There is a third way to do this.

(Refer Slide Time: 14:39)

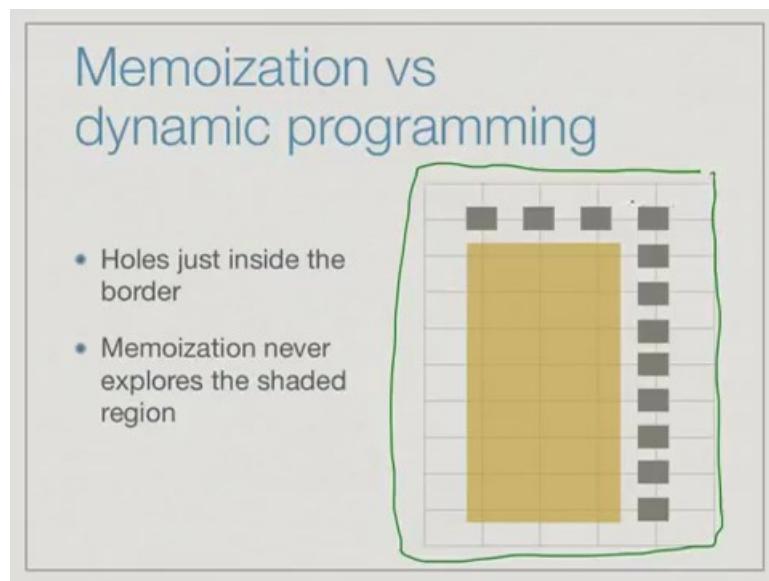


So, once we have one at (0, 0) then we can fill both the first element above it and the first element to its right. So, we can do this diagonal, now notice that any diagonal value like this one has both its entries. This has only one entry, this also. So I can now fill up this

diagonal. I can go one more diagonal, then I can go one more diagonal. So, we can also fill up this thing diagonal by diagonal.

The dependency structure may not require us to fill it in a particular way we might have very different ways to fill it up, all we want to do is systematically fill up this table in an iterative fashion not recursively we do not want to call  $f$  of  $i, j$  and then look at  $f$  of  $i$  minus 1,  $j$ . We want to directly say when we reach  $(i, j)$  we have the values we need, but the values we need could come in multiple different orders. So, we could have done it row wise, we could have done it column wise and here you see we can do it **diagonally**, but it does not **matter so long** as we actually get all the values that we need.

(Refer Slide Time: 15:36)



So, one small point, so we have said that we can use Memoization or we can use dynamic programming. **One** of the advantages of using dynamic programming is it avoids this recursive call. So, recursion we had mentioned earlier, also in some earlier lecture, **comes with a price** because whenever you make a recursive call, you have to suspend a computation, store some values, restore those values. There is a kind of administrative cost with recursion.

So, actually though it looks like only a single operation and we call  $\text{fib}$  of  $n$  minus 1 or  $\text{fib}$  of  $n$  minus 2. There is actually a cost involved with suspending this operation, going there and coming back. So, saving on recursion is one important reason to move from Memoization to dynamic programming, but what dynamic programming does is to

evaluate every value regardless of whether its going to be useful for the final answer or not.

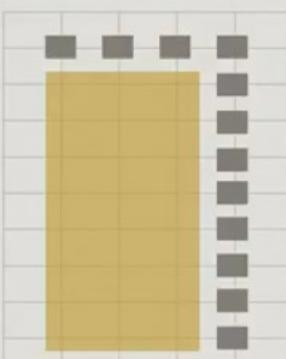
In the grid **path** thing there is one situation where you can illustrate this. **Imagine** that we have **these obstacles** placed exactly one step inside the boundary. Now, if we want to reach this its very clear that I can only come all the way along the top row or all the way up the rightmost column, there is no other way I can reach them. So, anything which is inside this these positions there is no way to go from here out. There is no point in counting all these values.

We have this region which is in the shadow of these obstacles which can never reach the final thing. So, when we do memoization when we come back and recursively explore it will never ask us to come here because it will never pass these boundaries. **On** the other hand our dynamic programming will blindly walk through everything. So, **it** will do row by row, column by column and it will eventually find the 0s, but it will fill the entire  $n$  by  $n$  grid. In this case how many will memoization **do?** It will do basically only the boundary. It will do only order  **$m+n$** .

(Refer Slide Time: 17:32)

## Memoization vs dynamic programming

- Memo table has  $O(m+n)$  entries
- Dynamic programming blindly fills all  $O(mn)$  entries
- Iteration vs recursion  
— “wasteful” dynamic programming is still better, in general



So, we have a memo table which has only a linear number of entries in terms of the rows and columns and a dynamic programming entry, which is quadratic; **if both were  $n$**  it will be  $n$  **squared**, thus is  **$2n$** . This suggests that dynamic programming in this case, is wastefully computing a **vast** number of entries. So  **$n$  squared** is much larger than  $2n$ .

remember. It will take us enormous amount of time to compute it, if we just count the cost per entry, but the flip side is that each entry that we need to add to the memo table requires one recursive call.

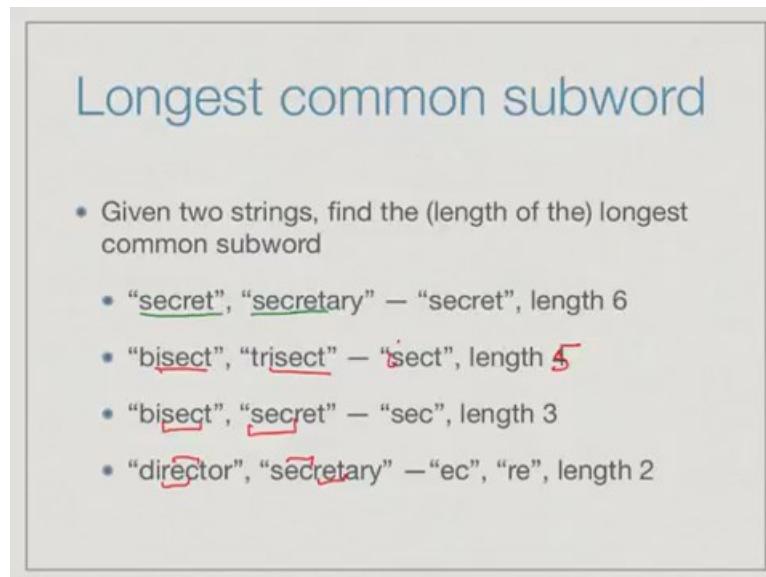
The reality is that these recursive calls will typically cost you much more, than the wastefulness of computing the entire table. In general even though you can analyze the problem and decide that memoization will result in many fewer new values being computed than dynamic programming. It is usually sound to just use dynamic programming as the default way to do the computation.

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

**Week - 08**  
**Lecture - 03**  
**Longest Common Subsequence**

We are in the realm of Inductive Definitions, Recursive Functions and Efficient Evaluation of **these** using memorization and dynamic programming.

(Refer Slide Time: 00:04)



**Longest common subword**

- Given two strings, find the (length of the) longest common subword
  - “secret”, “secretary” — “secret”, length 6
  - “bisect”, “trisect” — “sect”, length 5
  - “bisect”, “secret” — “sec”, length 3
  - “director”, “secretary” — “ec”, “re”, length 2

So, we are looking examples of problems where the main target is to identify the inductive structure and once you identify the inductive structure then the recursive structure of the program becomes apparent from which you can extract the dependencies and figure out what kind of memo-table you have and how you will can **iteratively** using dynamic programming.

This is something which comes to the practice and by looking at more examples hopefully the procedure become clearer, but the key thing to dynamic programming is to be able to understand the inductive structure. So, you need to take a problem, identify how the main problem depends on its sub parts and using this come up with the nice

inductive definition which you can translate in to a recursive program. Once you have the recursive program then the memo-table and the dynamic programming almost comes out automatically from that.

This is the problem involve in words. So, what you want to do is take a pair of words and find the longest common subword. For instance, here we have secret and secretary and secret is already also inside secretary and clearly secret is the longest word in secret itself. The longest subword that is common is the word secret and it has **length** 6. Let we move to the next think bisect and trisect then actually this should be isect that say which has length 5, similarly if we have bisect and secret then sec.

When we say subword, of course we do not mean a word in the sense; we just mean a sequence of letters. So, s e c is the longest common subword in has length 3 and if you have two very different words like director and secretary, sometimes you might have only small things, for example, here r e and e c are, for examples of subword but there are really very long words which are common to the subword is only length 2.

(Refer Slide Time: 02:11)

### More formally ...

- Two strings  $u = a_0a_1\dots a_{m-1}$ ,  $v = b_0b_1\dots b_{n-1}$
- If  $\underbrace{a_i a_{i+1} \dots a_{i+k-1}}_k = \underbrace{b_j b_{j+1} \dots b_{j+k-1}}_k$  for some  $i$  and  $j$ ,  
 $u$  and  $v$  have a common subword of length  $k$
- Aim: Find the length of the longest common subword of  $u$  and  $v$

Here is the more formal description right. So, supposing I have two words  $u$  and  $v$ . So,  $u$  is of length  $m$  and  $v$  is of length  $n$  and the number positions using python notation and

number 0 to n minus 1, 0 to n minus 1 then what I want to do is able to start at a i and go k steps. So, i to i plus k minus 1 and b j to j plus k minus 1 such that, these two segments are identical, this is a common subword and we want to find the longest such common subword, what is the k, we do not even want to subword, will find that subword will be a byproduct. You first need to just find k, what is the length of the longest common subword of u n?

(Refer Slide Time: 02:56)

## Brute force

- $u = a_0a_1\dots a_{m-1}$  and  $v = b_0b_1\dots b_{n-1}$
- Try every pair of starting positions  $i$  in  $u$ ,  $j$  in  $v$ 
  - Match  $(a_i, b_j), (a_{i+1}, b_{j+1}), \dots$  as far as possible
  - Keep track of the length of the longest match
- Assuming  $m > n$ , this is  $O(mn^2)$ 
  - $\text{mn}$  pairs of positions
  - From each starting point, scan can be  $O(n)$

There is a brute force algorithm that you could use which is you just start at  $i$  and  $j$  in two word. In each word you can start a position  $i$  in  $u$   $j$  in  $v$  and see how far you can go before you find **they are** not. So, you match  $a_i$  and  $b_j$  right. So, if  $a_i$  and  $b_j$  work then its fine. So, it should be  $b_j$  and if  $a_i$  and  $b_j$  work then you go to a  $i$  plus 1  $b_j$  plus 1 and so on and whenever we find two letters which differ then the commons adverse starting at  $a_j$  has ended and you so from  $i j$  I have a common subword of something.

Now, among all the **i** js you look for the longest one and that becomes your answer. Now, this unfortunately is effectively **now** an  $n$  cube algorithm. We think of  $m$  and  $n$  can be equal technically  $m$   **$n$  squared** because there are  $m$  times  $n$  different choices of  $i$  and  $j$  and in general I started  $i j$  and then I have to go from  $i$  to the end right and from  $j$  to the end.

So, we have to do a scan for each  $i j$  in this scan in general adds up to an order, order  $n$  factor and so we have order  $m n$  squared or order  $n$  cube if you like.

(Refer Slide Time: 04:09)

### Inductive structure

- $a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$  is a common subword of length  $k$  at  $(i,j)$  iff
  - $a_i = b_j$  and
  - $a_{i+1} \dots a_{i+k-1} = b_{j+1} \dots b_{j+k-1}$  is a common subword of length  $k-1$  at  $(i+1,j+1)$
- $LCW(i,j)$ : length of the longest common subword starting at  $a_i$  and  $b_j$ 
  - If  $a_i \neq b_j$ ,  $LCW(i,j) = 0$ , otherwise  $1 + LCW(i+1,j+1)$
  - Boundary condition: when we have reached the end of one of the words

Our goal is to find some inductive structure which makes this thing computationally more efficient. So, what is the inductive structure? Well we have already kind of seen it when can we say that there is a commons subword starting at  $i j$  of length  $k$ , the first thing is that we need this  $a i$  to be the same as  $b j$ . So, I need this condition and now if this is a commons subword of length  $k$  at  $i j$  then what remains of subword namely this segment from  $i$  plus 1 to this and  $j$  plus 1 to this must also match and they must be in turn be a  $k$  minus 1 length subword from here to there. So, we want to say that there is a  $k$  length subword starting at  $i j$  if  $a i$  is equal to  $b j$  and from  $i$  plus 1 and  $j$  plus 1 there is a  $k$  minus 1 length subword.

In other words, I can now write the following definition, I can say that the longest common the length of the longest common subword l c w starting from  $i j$ . Well, if they two or not the same if  $a i$  is not the same the same there is no common subword at all because if I start from  $i$  immediately have two different letters. So, when the length is 0 otherwise I can inductively find out what is the longest common subword to may right start  $i$  plus 1 start from  $j$  plus 1.

Find out what I can do from there and to word I can add one letter because this current letter  $a_i$  is equal to  $b_j$ . So, I get one plus that and the base case of the boundary condition is when one of the two words is empty right. If I have no letters left, if I have gone  $i, j$  I am looking at difference combinations  $i$  and  $j$ . So, if either  $i$  or  $j$  has reached the end of the word then there is no possibility of a common subword at that point. So, when we reach the end of one of the words say answer must be 0.

(Refer Slide Time: 05:58)

## Inductive structure

- Consider positions 0 to  $m$  in  $u$ , 0 to  $n$  in  $v$
- $m, n$  means we have reached the end of the word
- $\text{LCW}(m+1, j) = 0$  for all  $j$
- $\text{LCW}(i, n+1) = 0$  for all  $i$
- $\text{LCW}(i, j) = 0$ , if  $a_i \neq b_j$ ,
- 1.  $\text{LCW}(i+1, j+1)$ , if  $a_i = b_j$

This gives us the following definitions. So, remember that  $u$  is actually has length  $m$ . So, it has 0 to  $m$  minus 1. So, what we will do is, we will add a position of  $m$  to indicate that we have crossed the last letter. Similarly,  $v$  has 0 to  $n$  minus 1 has valid positions. So, we will use in this 0 to  $n$ . So, if  $i$  becomes  $m$  or  $j$  becomes  $n$  it means that that corresponding index has gone beyond the end of the word right. So, this should be  $m$  and this should be  $n$ . We have that if you reach  $m$  then  $\text{lcw}$  of  $n$  comma  $j$  0 is 0 because we have gone past the length  $u$ .

Similarly, if you reach  $n$ , but  $\text{lcw}$  of  $i$  comma  $n$  is 0 because you gone past the length of  $v$  and if you are not gone past the length, if you are somewhere inside the word in a valid position then the length is going to be 0 if the two positions are not the same. If  $a_i$  is not equal to  $b_j$ , otherwise inductively I compute the length from  $i$  plus 1 and  $j$  plus 1 and add

one to it. this is the case when  $a_i$  is equal to  $b_j$  because that segment  $i$  can extend by. So, this is just stating in an equation form the inductive definition that we purposely earlier.

(Refer Slide Time: 07:18)

## Subproblem dependency

- $\text{LCW}(i,j)$  depends on  $\text{LCW}(i+1,j+1)$
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | b |   |   |   |   |   | . |
| 1 | i |   |   |   |   |   |   |
| 2 | s |   |   |   |   |   |   |
| 3 | e |   |   |   |   |   |   |
| 4 | c |   |   |   |   |   |   |
| 5 | t |   |   |   |   |   |   |
| 6 | . |   |   |   |   |   |   |

So, here we saw example for **bisect and secret**. We have position 0 to 5 and then we have the 6 position indicating the end of the word and now remember that the way our inductive definition was phrased  $i, j$  depends only on  $i + 1, j + 1$ . So, actually the dependencies at this ways to the arrows are indicating that and in order solve this I need to solve this first. The value at 2 comma 3 depends on the value 3 comma 4.

In order to solve this I do not need to solve anything because everything once  $i$ . So, in order to solve this I only need to so anyway. We can basically, we have this simple thing which says that the corner and the actually the right column and the bottom thing do not require anything and we know that because those are all 0s.

(Refer Slide Time: 08:08)

## Subproblem dependency

- $\text{LCW}(i,j)$  depends on  $\text{LCW}(i+1,j+1)$
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |   |
|---|---|---|---|---|---|---|---|---|
|   | s | e | c | r | e | t | . |   |
| 0 | b | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | i | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | s | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | e | 0 | 2 | 0 | 0 | 1 | 0 | 0 |
| 4 | c | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | t | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | . | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

We can actually fill in those values as 0 because that is given to us **by definition** and now we can start, for instance, we can start with this value because its value is known. We will look at whether this t matches that t it is that. So, we take one plus the value two is bottom. So, we get 1 and then we can walk up and do the same thing at every point we will say that if c is not the same as t. So, none of these letters if you look at these letters here right none of these letters are t. So, for all of these letters I will get 0 directly because it says that a i is not equal to b j.

I do not even have to look at i plus 1 j plus 1, I directly says that 0 because is not there. So, in this way I can fill up this column. This is like our grid pack thing I can fill by column by column even though there the dependency was to the left and bottom and here the dependence is diagonally bottom right. I can fill up column by column and I can keep going and if I keep going I find an entry 3. So, the entry 3 is the largest entry that I see and that is actual answer this entry 3.

(Refer Slide Time: 09:13)

## Reading off the solution

- Find  $(i,j)$  with largest entry
  - $\text{LCW}(2,0) = 3$
- Read off the actual subword diagonally

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | . |
|---|---|---|---|---|---|---|---|---|
| s | 0 | e | c | r | e | t | . |   |
| 0 | b | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | i | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | s | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | e | 0 | 2 | 0 | 0 | 1 | 0 | 0 |
| 4 | c | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | t | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | . | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

And now we said earlier that we are focusing on the length of the longest common subword not the word itself in the reason. We need to; we can afford to do that is because we can actually read off the answer once we have got the lengths. So, we ask ourselves why we did we get a 3 here. We got 3 here because we came as 1 plus 2. Since we came as 1 plus 2 it must mean that these two letters are the same. So, we got 2 here is because it is 1 plus 1. So, these two letters must also be the same. Finally, we got one here because this is 1 plus 0. So, these two letters are the same. Therefore, these three letters must be the same.

(Refer Slide Time: 09:54)

## Reading off the solution

- Find  $(i,j)$  with largest entry
  - $\text{LCW}(2,0) = 3$
- Read off the actual subword diagonally

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | b | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | i | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | s | 3 | 0 | 0 | 0 | 0 | 0 |
| 3 | e | 0 | 2 | 0 | 0 | 1 | 0 |
| 4 | c | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | t | 0 | 0 | 0 | 0 | 0 | 1 |
| 6 | . | 0 | 0 | 0 | 0 | 0 | 0 |

If you walk down from that magical value, the largest value and we follow the sequence then we can read off and the corresponding row or column because they are the same, your actual subword which is the longest common subword for these two.

(Refer Slide Time: 10:10)

## LCW( $u,v$ ), DP

```
def LCW(u,v): # u[0..m-1], v[0..n-1]
    for r in range(len(u)+1):
        LCW[r][len(v)+1] = 0 # r for row
        for c in range(len(v)+1):
            LCW[len(u)+1][c] = 0 # c for col
    maxLCW = 0
    for c in range(len(v)+1,-1,-1):
        for r in range(len(u)+1,-1,-1):
            if u[r] == v[c]:
                LCW[r][c] = 1 + LCW[r+1][c+1]
            else:
                LCW[r][c] = 0
                if LCW[r][c] > maxLCW:
                    maxLCW = LCW[r][c]
    return(maxLCW)
```

Here is a very simple implementation in python. So, all it says is that you start with the two words u and v, you initialize this lcw thing at the boundary at the nth row on the nth column and then, now you remember the maximum value. So, you keep that by initializing the maximum value to 0 and then you fill up in this particular case the column order.

For each column, then for each row and that column you fill up the thing using the equation, if it is equal i to 1 plus otherwise as it say 0 and if i see and a new value this is the thing where I update if i see and new entry which is bigger than the entry which is currently the maximum, I update the maximum. So, this is allows me to quickly find out what is the maximum length overall and finally, when I go through this look i would filled up the entire table and i will return the maximum value i saw over.

(Refer Slide Time: 11:08)

## Complexity

- Recall that the brute force approach was  $O(mn^2)$
- The inductive solution is  $O(mn)$  if we use dynamic programming (or memoization)
  - Need to fill an  $O(mn)$  size table
  - Each table entry takes constant time to compute

So, when we did it by brute force we had an order  $m n$  square algorithm. Here we are filling up table which is of size order  $m$  by  $n$  and each entry only require us to check the ith position in the word the jth position in the world and depending on that, if necessary look up one entry i plus 1 j plus 1. It is a constant time update. We need to fill up one table of size order  $m n$  each update takes constant time. So, this algorithm brings us from  $m n$  squared in the brute force case to  $m n$  using dynamic programming.

(Refer Slide Time: 11:43)

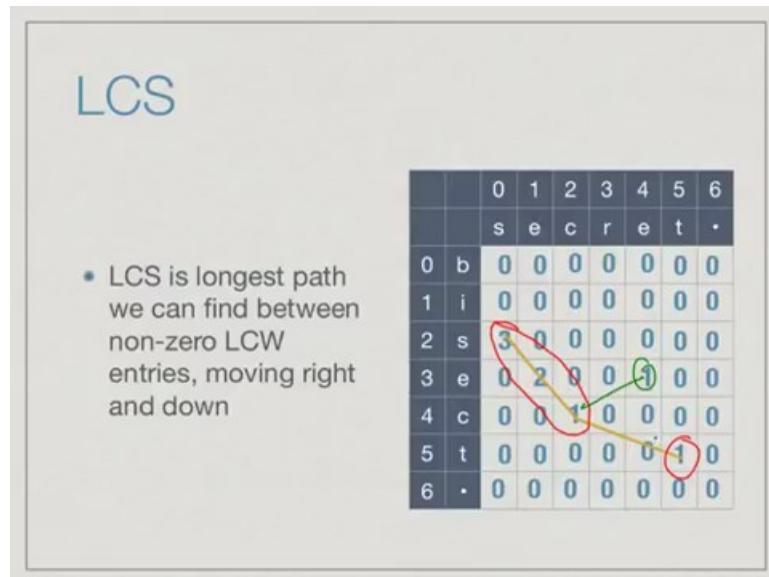
## Longest common subsequence

- Subsequence: can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
  - “secret”, “secretary” — “secret”, length 6
  - “bisect”, “trisect” — “isect”, length 5
  - “bisect”, “secret” — “sect”, length 4
  - “director”, “secretary” — “ectr”, “retr”, length 4

A much more useful problem in practice than the longest common subword is what is called the longest common subsequence. So, the difference between a subword and a subsequence is that we are allowed to drop some letters in between. So, for instance, if you go back to the earlier examples of secret and secretary, there is no problem because the subword is actually the entire thing and again for bisect and trisect also it is the same thing, but if we have bisect and secret earlier if we did not allow us to skip, we could only match sec with sec, but now we can take this extra t and we can skip there 2 and match this t as say that s e c t is a subsequence in the right word, which matches the corresponding subsequence which is also a subword in the left word in the right is not a subword.

For the subsequence i have to drops some letters to get s e c t. Similarly, if I have secretary and director then I can find things like e c t r e c t r in both of them by skipping over with judiciously. So, why is this are better problem?

(Refer Slide Time: 12:54)



Well we will see that, but effectively what skipping mean, skipping means that I get segments which are connected by gaps.. So, I get this segment then I want to continue this segments. So, I look for the next match, I skip, but the next match must come to my right. It must come to the right and below the current match because I cannot go backwards in a word and start the match again. So, I cannot, for instance go here and say that this is an extension because this requires me to go back and reuse the e that I have seen in sec to match.

The second e in secret which is not allowed, I can keep going forward which in the table corresponds to going to the right and back to the right and down. So, I am going increasing the order of index in both words and I can group together these things and this is what the longest kind of subsequences. So, we could in principle look at the longest common subword answer and look for these clever connections, but it turns out there is a much more direct way to do it in an inductive way.

(Refer Slide Time: 13:56)

## Applications

- Analyzing genes
  - DNA is a long string over A,T,G,C
  - Two species are closer if their DNA has longer common subsequence
- UNIX diff command
  - Compares text files
  - Find longest matching subsequence of lines

The motivations, well one of the big motivations for subsequence matching comes from things like genetics, for instance, when we compare the genes sequence of two organisms they are rarely equal. So, what we are looking for are large matches, where there might be some junk genes in between which you want to discard. So, you want to say that two genes sequences or two organisms are similar, if there are large overlaps over the entire genome not just looking for individual segment along, but by just throwing away the minimal things on both sides, we can make that align as we call.

And other important example is something called a diff, which is Unix command compared to text files. So, this treats in fact, line by line two files as a word. So, each line is compared to each line in the other file if the line match they considered to be equal and this is the good way of comparing one version of the file with other version of the file. Supposing you are collaborating on a document or program with somebody else and you send it by email and they send it by.

So, they had made some changes then diff tells you quickly, what are the differences between the file you sent and the file you got back and diff essentially is doing the same thing is trying to find the longest match between the file that you sent of the file you got back and the shortest way in which you can transform one to the other by change in the

few lines its. These are some typical example of this longest common subsequence problem and therefore, it is usually much more useful in practice in the longest common subword problem.

(Refer Slide Time: 15:29)

### Inductive structure

- If  $a_0 = b_0$ ,
- $LCS(a_0 \dots a_{m-1}, b_0 \dots b_{n-1}) = 1 + LCS(a_1 a_2 \dots a_{m-1}, b_1 b_2 \dots b_{n-1})$
- Can force  $(a_0, b_0)$  to be part of LCS
- If not,  $a_0$  and  $b_0$  cannot both be part of LCS
  - Not sure which one to drop
  - Solve both subproblems  $LCS(a_1 a_2 \dots a_{m-1}, b_0 b_1 \dots b_{n-1})$  and  $LCS(a_0 a_1 \dots a_{m-1}, b_1 b_2 \dots b_{n-1})$  and take the maximum

What is the inductive structure of the longest common subsequence problem? As before we have the words laid out. So, we can say a 0 to a m or a minus 1 a n minus 1, it does not matter how you choose it, but in the picture it says a n, but if you want to you can remove this last. So, a 0 to a n minus 1 is the first word b 0 to b n minus 1 is the second word and now there are two cases. The first case is the easy case, supposing I have these two things are equal then like before I can inductively solve the problem for a 1 and b 1 onwards and add this. I can extend that solution by saying a 0 match is b 0 and then whatever matches.

What is the subsequence, the subsequence actually is some kind of a matching, it says that you know it will say that this matches this and then this matches this, these are the same and this match is this and then this match is this and so on. Only thing is that these lines cannot overlap, they must be kept going from left to right without overlap. So, this kind of pairing up equal letters, the maximum way in which again to do this is a longest common subsequence.

Now, what we are saying is that if I can actually match the first two things then I should match them and then I can go ahead and match the rest as I want, and the reason is very simple, supposing the best solution did not match these supposing you claim that the best solution actually requires meet match a 0 and b 1. Well, if I could match a 0 and b 1 I can also undo it and match a 0 and b 0 and then continue because a 0 and b 1 if they match and a 1 if match is to right. So, I can take that solution and change it to a solution where a 0 matches b 0. The first two letters are the same and might is well go with that and say it is one plus the result of optimally solved in the rest, what if they not the same this is the interesting case.

Supposing, these are not the same then what happens. Then can we just go ahead and ignore a 0 and b 0, no right. So, it could be that a 0 actually matches b 1 or it could be that b 0 matches a 1, we do not know, but we certainly know that a 0 does not match b 0. So, we have to drop one of them because we cannot make a solution a 0 matching b 0, but we do not know which one. So, what we do is we take two sub problems, we say let us assume b 0 is not part of the solution then the best solution come out of a 0 to a and minus 1 and b 1 to b n, b 0 is exclude because I cannot match it to the a 0 and whatever a 0 matches must match to the right. So, i must go ahead with it.

But maybe this is the wrong choice. The other choice should be to keep b 0 and drop a 0 and which case I do a 1 to a m minus 1 and b 0 to b m. These are two different choices which one to choose, well since we do not know we solve them both. We solve if a i a 0 is not b 0 we solved both these problems a 1 to a m minus 1 **b 0** and a 0 to a m minus 1 b 1 solve of them take the maximum one whichever one is better it is **the one**.

(Refer Slide Time: 18:45)

## Inductive structure

- $\text{LCS}(i,j)$  stands for  $\text{LCS}(a_i a_{i+1} \dots a_m, b_j b_{j+1} \dots b_n)$
- If  $a_i = b_j$ ,  $\text{LCS}(i,j) = 1 + \text{LCS}(i+1,j+1)$
- If  $a_i \neq b_j$ ,  $\text{LCS}(i,j) = \max(\text{LCS}(i+1,j), \text{LCS}(i,j+1))$
- As with LCW, extend positions to  $m+1, n+1$
- $\text{LCS}(m+1,j) = 0$  for all  $j$
- $\text{LCS}(i,n+1) = 0$  for all  $i$

This in general will take us deeper in the words. So, we said a 0 b 0 will require solved it for a 1 and b 0 or a b a 0 and b 1. So, in general we have a i and b j right. Again since we have a i and b j then you will use the same logic if a i is equal to b j then it is one plus the rest. So, this is the good case, if a i is not equal to b j then what we do is we look at the same thing, we drop b j and solve it and symmetrically we drop a i and solve it and take the better of the two. We take max of the solution from i and the solution from j plus 1.

If we say like we had before that lcs of i j is the length of the longest common starting to i and j if a i is equal to b j it will be one plus the length start it from i plus 1 j plus plus 1. If it is not equal it will be the maximum of the two sub problems where either increment i or increment j and has with the longest common subword when we go to the last position m and n we get 0.

(Refer Slide Time: 20:01)

## Subproblem dependency

- $\text{LCS}(i,j)$  depends on  $\text{LCS}(i+1,j+1)$  as well as  $\text{LCS}(i+1,j)$  and  $\text{LCS}(i,j+1)$
- Dependencies for  $\text{LCS}(m,n)$  are known
- Start at  $\text{LCS}(m,n)$  and fill by row, column or diagonal

So, here the dependency is slightly more complicated because depending on the case, I either have to look at  $i + 1$ ,  $j + 1$  or  $i + 1, j$  or  $i, j + 1$ . So, I had for this square, I had looked at its right neighbor, right diagonal neighbor and the bottom neighbor, but once again the ones which have no dependency appear. So, earlier we had for longest common subword we had only this dependency this mean that even a square like this had no dependencies because there is nothing to its bottom right.

But now, for instance if we look at this picture, since we are looking bottom right and left, if I look at this its dependencies are in three directions; two of the directions are empty, but this direction there is dependence. So, I cannot fill up this square directly the can only square, I can fill up directly is this one because it has nothing to its right nothing in diagonally and nothing below.

(Refer Slide Time: 20:54)

## Subproblem dependency

- $LCS(i,j)$  depends on  $LCS(i+1,j+1)$  as well as  $LCS(i+1,j)$  and  $LCS(i,j+1)$
- Dependencies for  $LCS(m,n)$  are known
- Start at  $LCS(m,n)$  and fill by row, column or diagonal

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| s | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| e | b | 4 | 3 | 2 | 1 | 1 | 0 |
| c | i | 4 | 3 | 2 | 1 | 1 | 0 |
| r | s | 4 | 3 | 2 | 1 | 1 | 0 |
| t | e | 3 | 3 | 2 | 1 | 1 | 0 |
|   | c | 2 | 2 | 2 | 1 | 1 | 0 |
|   | t | 1 | 1 | 1 | 1 | 1 | 0 |
|   | . | 0 | 0 | 0 | 0 | 0 | 0 |

So, I start from there and I put a 0 and as before we can go down this because now once we have this, we have everything would to its left and once we have this and because we are beyond the word where at the m, this dummy position, the row and column becomes 0, but the important thing to remember is the row and column become 0 not because they have no dependency, but because we can systematically, fill it up exactly like in the grid parts we can fill up the bottom row and the left most column there, here the right most column.

Now, once we have this we can fill up this part right and then again we can have two and three c entries we can fill up this. We have three entries we can fill up this, we have three entries we can fill up this and we can fill up this column and we can do this column and we can do this column by column and we propagate it and then finally, the value the propagates here is our longest length of the longest common subsequences, we could also do this row by row.

(Refer Slide Time: 21:44)

## Recovering the sequence

- Trace back the path by which each entry was filled
- Each diagonal step is an element of the LCS
- “sect”

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | . |
|   | s | e | c | r | e | t | . |   |
| 0 | b | 4 | 3 | 2 | 1 | 1 | 0 | 0 |
| 1 | i | 4 | 3 | 2 | 1 | 1 | 0 | 0 |
| 2 | s | 4 | 3 | 2 | 1 | 1 | 0 | 0 |
| 3 | e | 3 | 2 | 1 | 1 | 0 | 0 | 0 |
| 4 | c | 2 | 2 | 2 | 1 | 1 | 0 | 0 |
| 5 | t | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 6 | . | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Now, how do we trace out the actual solution when the solution grows, whenever we increment the number? So, we can ask why is this 4? So, we say that this is 4 not because we did plus 3 because s is not equal to b, we did 4 because we got the max value from here, why is this 4 again i is not equal to s. So, we got the max value from here why is this 4, Oh s is equal to x. We must have got it by 3 plus 1, why is this because e plus e. So, we must have got it from here. So, we follow the path according to the choices that we made in applying the inductive function in order to generate the value at each parts.

In other words, for each cell  $i j$  that we write we remember whether we wrote it because it was one plus that diagonal neighbor or the maximum of the left in the right in which case we record whether the left or the bottom it was the maximum. Now, in this a picture every time we take a diagonal step it means we actually had a match. So, this is the match the first one here is a match s equal to s, this is the match e equal to e, this is the match c equal to c. Now, after this point we are flat and then a at this point again we have a match. So, we get s e c and t.

So, we can read off the diagonal steps along this kind a explanation of the longest number largest number we got and each diagonals step will contribute to the final solution, now there could be more than 1, because we have not got any example in this case, but

sometimes the max could be one of in both directions if I am taking max of the left the right neighbor and the bottom they could be the same. I could have the situation like this supposing I landed up here then I do not know whether I got it from here or from here. So, I might have two different extensions which lead me to a solution. So, the longest common subsequence need not be unique, but you can recover at least one by following this path

(Refer Slide Time: 23:32)

## LCS(u,v), DP

```
def LCS(u,v): # u[0..m-1], v[0..n-1]
    for r in range(len(u)+1):
        LCS[r][len(v)+1] = 0 # r for row
    for c in range(len(v)+1):
        LCS[len(u)+1][c] = 0 # c for col
    for c in range(len(v),-1,-1):
        for r in range(len(u),-1,-1):
            if (u[r] == v[c])
                LCS[r][c] = 1 + LCS[r+1][c+1]
            else
                LCS[r][c] = max(LCS[r+1][c],
                                  LCS[r][c+1])
    return(LCS[0][0])
```

So, here is the python code is not very different from the earlier one. We can just see we have just initialize the last row and the bottom row on the last column and then as before you walk up row by row, column by column and filling using the equation and in this case, we do not have to keep track of the maximum value and keep updating because the maximum value automatically propagates to the 0 0 value.

(Refer Slide Time: 23:56)

## Complexity

- Again  $O(mn)$  using dynamic programming (or memoization)
  - Need to fill an  $O(mn)$  size table
  - Each table entry takes constant time to compute

Just like the longest common subword, here once again we are filling in a table of size  $m$  times  $n$ . Each entry only requires you to look at most do three other entries. So, one to the right one to the bottom right in that the one below. So, it is a constant amount of work. So,  $m n$  entries constant amount of work per entry, this takes time  $m$  times  $n$ .

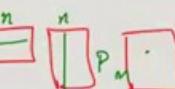
**Programming, Data Structures and Algorithms in Python**  
**Prof. Madhavan Mukund**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Week - 08**  
**Lecture - 04**  
**Matrix multiplication**

This is a final example to illustrate dynamic programming. We look at the problem of matrix multiplication.

(Refer Slide Time: 00:02)

### Multiplying matrices



- To multiply matrices A and B, need compatible dimensions
  - A of dimension  $m \times n$ , B of dimension  $n \times p$
  - AB has dimension  $mp$
- Each entry in AB take  $O(n)$  steps to compute
  - $AB[i,j] = \underbrace{A[i,1]B[1,j]}_{\text{row}} + \underbrace{A[i,2]B[2,j]}_{\text{row}} + \dots + \underbrace{A[i,n]B[n,j]}_{\text{row}}$
- Overall, computing AB is  $O(mnp)$

If you remember how matrix multiplication works, you have to take two rows, two matrices with compatible entries, and then we compute a new matrix in which for each entry there, we have to multiply a row here by a column of the same length and add up the values. If we have a matrix which is  $m$  by  $n$  then this must have  $n$  times  $p$ , so that we have a final answer which is  $m$  times  $p$ . In the final entry, we have to make  $mp$  entries in the final product matrix; and each of these entries, require us to compute this sum of these  $n$  entries, so that takes us order  $n$  time. With total work is, usually easy to compute us  $m$  times  $n$  times  $p$ .

So,  $AB[i,j]$  is this long sum and this sum has 1, 2, 3 up to  $n$  entries. Computing matrix multiplication for two matrices has a very straight forward algorithm which is a product triple nested loop which takes order  $m$  times  $n$  times  $p$ , if all of the dimension are the

same this is an order **n cubed algorithm**. Now there are more clever ways of doing it, but that is not the purpose of this lecture, but the naive straightforward, **way of** multiplying two matrices is m times n times p. Our interest is when we have a sequence of such multiplications to do.

(Refer Slide Time: 01:34)

## Multiplying matrices

- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Bracketing does not change the answer ...
  - ... but can affect the complexity of computing it!

Supposing, we want to multiply three matrices together A times B times C, then it turns out it does not matter **whether** we first multiply AB and then multiply C or we first multiply A and then multiply BC, A times BC, because this is stated as the associative the order **in** which we group the multiplication does not matter just for normal numbers. If we do 6 times 3 times 2, it does not matter whether you do 6 times 3 first, or 3 times 2 first finally, the product is going to be the same. So, the bracketing does not change the answer the final value is the same, but it turns out that it can have dramatic effects on the complexity of computing **the answer**. Why **is this** the case.

(Refer Slide Time: 02:18)

## Multiplying matrices

- Suppose dimensions are A[1,100], B[100,1], C[1,100]
  - Computing A(BC)
    - BC is [100,100],  $100 \times 1 \times 100 = 10000$  steps
    - A(BC) is [1,100],  $1 \times 100 \times 100 = 10000$  steps
  - Computing (AB)C
    - AB is [1,1],  $1 \times 100 \times 1 = 100$  steps
    - (AB)C is [1,100],  $1 \times 1 \times 100 = 100$  steps
- A(BC) takes 20000 steps, (AB)C takes 200 steps!

Suppose, we have these matrices A, B and C, and A and B have these columns and rows. A is basically got 1 by 100, A just has 1 row and 100 columns and B has a 100 rows and 1 column. And C has again 1 row and 100 columns. These are matrices which look like this. Now what happens is that when I multiply d times C then I get something which is 100 into 1 into 100. So, we will get an output which is a 100 by 100 matrix, so that has 10000 entries, so it is going to take us 10,000 steps.

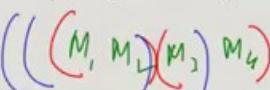
Now when I multiply A by this I am going to get 1 into 100 into 100 that is another 10000 step. If I do A, after I do BC and I do 10000 plus 10000, so I do 20000 steps. Now if I do it the other way, if I take A times B first then this whole thing collapses into a 1 by 1 single entry. So, I get 1 into 100 into 1 in 100 steps, I just collapse this row and this column into a single entry that is like computing one entry in a matrix multiplication with the resulting thing is exactly that one entry.

Now I have this one entry, and again I have to multiply it by this thing and that will take me for each of the columns in this I will get one entry, so that will take me 100 steps. I take 100 steps to collapse this A into B into a single cell, and another 100 steps after that to compute that product into C. So instead of 20000 steps, I have done it in 200 steps. This is the way in which the sequence of multiplications though multiplication is associative and it does not matter what you do you will get the same answer; the

sequence in which you do the associative steps can dramatically improve or worsen the amount of time you spend doing this.

(Refer Slide Time: 04:09)

## Multiplying matrices

- Given matrices  $M_1, M_2, \dots, M_n$  of dimensions  $[r_1, c_1], [r_2, c_2], \dots, [r_n, c_n]$ 
  - Dimensions match, so  $M_1 \times M_2 \times \dots \times M_n$  can be computed
  - $c_i = r_{i+1}$  for  $1 \leq i < n$  
  - Find an optimal order to compute the product
    - That is, bracket the expression optimally

In general, we have a sequence  $M_1$  to  $M_n$  and each of them has some rows and columns, and what we are guaranteed is that each adjacent pair can be multiplied. So,  $r_1$   $c_1$ ,  $r_2$   $c_2$  the first two are such that  $c_1$  is equal to  $r_2$ , the number of columns in the first matrix is equal to the number of rows in second matrix, similarly,  $c_2$  is equal to  $r_3$  and so on. These dimensions are guaranteed to match, so the matrix multiplication is always possible.

Our target is to find out in what order we would do it. So, we can at best do two matrices at a time, we only know how to multiply A times B, we cannot take three matrices and directly multiply them. If we have to do three matrices, we have to do in two steps A times B and then C, or B times C and then A. Now same way with  $n$  matrices, we have to do two at a time, but those two at a time could be a complicated thing. I could do  $M_1$   $M_2$  then I can combine that and do that combination with 3 or I can do  $M_1$   $M_2$ ,  $M_3$   $M_4$  and then do combination of  $M_1$   $M_2$  multiplied by  $M_3$   $M_4$  and so on.

What is the optimal way of computing the product, what is the optimal way of putting brackets in other words? Brackets are what tell us, so when we say  $M_1, M_2, M_3, M_4$  then one way of computing it just to do this right do  $M_1, M_2$ , then  $M_3, M_4$ . And other way of doing it would be to say do  $M_1, M_2$  and then do that multiplied by  $M_3$  and then

M 4 and so on. So, different ways of bracketing correspond to different evaluation orders for this multiplication. And what you want to do is kind of calculate without doing the actual computation which is the best sequence and which to do this calculation, so that we optimize the operations involved.

(Refer Slide Time: 05:48)

### Inductive structure

- Product to be computed:  $(M_1 \times M_2 \times \dots \times M_n)$
- Final step would have combined two subproducts
  - $(M_1 \times M_2 \times \dots \times M_k) \times (M_{k+1} \times M_{k+2} \times \dots \times M_n)$ , for some  $1 \leq k < n$
  - First factor has dimension  $(r_1, c_k)$ , second  $(r_{k+1}, c_n)$
  - Final multiplication step costs  $O(r_1 c_k c_n)$
  - Add cost of computing the two factors

What is the inductive structure? Finally, when we do this remember we **only** do two at a time. At the end, we must end with the multiplication which involves some group multiplied by some group it must look like this, and must have this whole thing collapsing to some  $M_1$  prime, and this whole thing collapsing to  $M_2$  prime and **finally** multiplying  $M_1$  prime by  $M_2$  prime. In other words  $M_1$  prime is for some  $k$  it is from  $M_1$  all the way up to  $M_k$ , and  $M_2$  prime is from  $k + 1$  up to  $n$ . So, this is my  $M_1$  prime and this is my  $M_2$  prime, and this  $k$  is somewhere between 1 and  $n$ .

In the worst case I could be doing  $M_1$ , and on the other side I can do it I could have done  $M_2$  up to  $M_n$  this whole thing. This whole thing is my, the other worst cases I could have done  $M_1$  2 not worst, but extreme cases  $M_1$  into  $M_n$  minus 1, I might have already computed, and now I want to finally, multiply it by  $M_n$  or it could be anywhere in between. If I just pick an arbitrary  $k$  then the first one is **has**  $r_1$  rows  $c_k$  columns.

So, second one as  $r_k$  plus one rows  $c_n$  column, but we know that  $c_k$  is equal to  $r_k$  plus 1. So this matrix will work. The final computation is going to be  $r_1$  into  $c_k$  into  $c_n$  right,  $m$  into  $n$  into  $p$  - the rows into the column, common number of column row into the

final number of columns. So this final multiplication, we know how much it is going to cost. And to this, we have to recursively add the inductive cost of having computed the two factors; how much time it will take us to **do** M 1 prime how much time it will take us to **do** M 2 prime.

(Refer Slide Time: 07:31)

## Subproblems

- Final step is  $(M_1 \times M_2 \times \dots \times M_k) \times (M_{k+1} \times M_{k+2} \times \dots \times M_n)$
- Subproblems are  $(M_1 \times M_2 \times \dots \times M_k)$  and  $(M_{k+1} \times M_{k+2} \times \dots \times M_n)$
- Total cost is  $\text{Cost}(M_1 \times M_2 \times \dots \times M_k) + \text{Cost}(M_{k+1} \times M_{k+2} \times \dots \times M_n) + r_1 c_k c_n$
- Which k should we choose?
- No idea! Try them all and choose the minimum!

We have that the cost of M 1 splitting at k is a cost of M 1 to M k plus the cost of M k plus **one** to M n plus the last multiplication r 1 into c k to c n. So, clearly this cost will vary for different case, and there are many **number of cases**. We said there are n minus 1 choices of **k**, anything from 1 to n minus 1 we can choose as k. There are n minus 1 sub problems. So, **when we did** the longest common sub sequence problem, we had two sub problems. We could either drop the first letter a i or the second letter b j, and then we have to consider two sub problems. We had no way of knowing which is better, so we did them both and took the max.

Now here we have n minus 1 different choices of k, we **have** no way of knowing which of this case is better. So, again we try all of them and take the minimum. There we **were** doing the maximum because we want the longest **common** subsequence, here we want the minimum cost so we choose that k which minimizes this split, and recursively each of those things would minimize their split and so on. So, that is the inductive structure.

(Refer Slide Time: 08:37)

### Inductive formulation



- $\text{Cost}(M_1 \times M_2 \times \dots \times M_n) =$   
minimum value, for  $1 \leq k < n$ , of
$$\text{Cost}(M_1 \times M_2 \times \dots \times M_k) +$$
$$\text{Cost}(M_{k+1} \times M_{k+2} \times \dots \times M_n) +$$
$$r_1 c_k c_n$$
- When we compute  $\text{Cost}(M_1 \times M_2 \times \dots \times M_k)$  we will get subproblems of the form  $M_j \times M_{j+1} \times \dots \times M_k$

Finally, we say that the cost of multiplying  $M_1$  to  $M_n$  is the minimum for all choices of  $k$  of the cost of multiplying  $M_1$  to  $M_k$  plus the cost of multiplying  $M_k$  plus 1 to  $M_n$  plus. Of course, for that choose of  $k$ , we have to do one final multiplication which is to take these resulting sub matrices, so that is  $r_1$  into  $c_k$  into  $c_n$ . So, when we take this, so we have  $M_1$  to  $M_n$ , so we have picked  $M_1$  to  $M_k$ .

Then as before what will happens is that we will have to split this somewhere, so we will we will now end up having some segment which is neither  $M_1$  nor  $M_n$ , it starts at some  $M_j$  and goes to  $M_k$ . In general, we need to express this quantity for arbitrary left and right point, we cannot assume that the left hand point is 1; we cannot assume the right hand point is  $n$  right.

(Refer Slide Time: 09:25)

In general ...

- $\text{Cost}(M_i \times M_{i+1} \times \dots \times M_j) =$   
minimum value, for  $i \leq k < j$ , of  
 $\text{Cost}(M_i \times M_{i+1} \times \dots \times M_k) +$   
 $\text{Cost}(M_{k+1} \times M_{k+2} \times \dots \times M_j) +$   
 $r_i c_k c_j$
- Write  $\text{Cost}(i,j)$  to denote  $\text{Cost}(M_i \times M_{i+1} \times \dots \times M_j)$

In general, if we have a segment from  $M_i$  to  $M_j$ , then we want the smallest value of among all the values of  $k$  from  $i$  to  $j$  minus 1, we want the minimum cost which occurs from computing the cost of  $M_i$  to  $M_k$ , and  $M_k$  plus 1 to  $M_j$ , and the cost of this final multiplication which is  $r_i$  into  $c_k$  into  $c_j$ . This quantity we will write as  $\text{cost } i \text{ } j$   $\text{cost } i \text{ } j$  is a cost of computing the segment from  $M_i$  to  $M_j$  which involves picking the best  $k$ . So,  $M_i$  to  $M_j$  is called  $\text{cost } i \text{ } j$ , and we use this same recursive inductive definition choose the best.

(Refer Slide Time: 10:07)

Final equation

- $\text{Cost}(i,i) = 0$  — No multiplication to be done
- $\text{Cost}(i,j) = \min \text{ over } i \leq k < j$   
 $[ \text{Cost}(i,k) + \text{Cost}(k+1,j) + r_i c_k c_j ]$
- Note that we only require  $\text{Cost}(i,j)$  when  $i \leq j$

The base case well if we are just looking at a segment of length 1, supposing we just want to multiply one matrix from 1 to 1, or 3 to 3, or 7 to 7 nothing is to be done. It is a just matrix there is no multiplication, so the cost is 0. We can then write out this cost  $i$   $j$  equation saying the minimum over  $k$  of cost  $i$   $k$  plus cost  $k$  plus 1  $j$  plus  $r_i c_k c_j$  which is the actual multiplication. And of course,  $i$  is always going to be less than or equal to  $j$ , because we are doing it from left to right, so we can assume that the segment is given to us with two end points where  $i$  is less than or equal to  $j$ .

(Refer Slide Time: 10:49)

## Subproblem dependency

- Cost( $i,j$ ) depends on Cost( $i,k$ ), Cost( $k+1,j$ ) for all  $i \leq k < j$
- Can have  $O(n)$  dependent values, unlike LCS, LCW
- Start with main diagonal and fill matrix by columns, bottom to top, left to right

So,  $i$  is less than or equal to  $j$ , and we look at cost  $i$   $j$  as a kind of matrix then this whole area where  $i$  is greater than  $j$  is ruled out. So, we only look at this diagonal. And the entries along the diagonal are the ones which are of the form  $i$  comma  $i$ , so all these entries are initially zero right. There is no cost involved with doing any multiplication from position  $j$  to position  $j$  along this diagonal. Now, in order to compute  $i$  comma  $j$ , I need to pick a  $k$ , and I need to compute for that  $k$  all the values I mean I have to compute  $i$   $k$ , and so it turns out that this corresponds to saying that if I want to compute a particular entry  $i$  comma  $j$ , then I need to choose a good  $k$ .

And in order to choose a good  $k$ , I need so this I can express in many different ways. I can say pick, so for example, supposing I want to compute this particular thing then I have to say pick this entry  $i$  to  $i$  and then I want the entry  $i$  plus 1 to  $j$ . So, I have  $i$  plus 1

to  $j$  this entry. These two entries I have to sum up; otherwise, I have to take this entry and sum it up with this entry.

In general, if I have a thing there, I will say if I choose this entry as my  $k$  point then I must add this entry to get it up, and take this sum or I have to take this entry and add this entry and then add this and so on. In general, we could have order  $n$  values I need to compute for this I need to compute, I need all the values here and I need all the values here.

(Refer Slide Time: 12:40)

## Subproblem dependency

- $\text{Cost}(i,j)$  depends on  $\text{Cost}(i,k), \text{Cost}(k+1,j)$  for all  $i \leq k < j$
- Can have  $O(n)$  dependent values, unlike LCS, LCW
- Start with main diagonal and fill matrix by columns, bottom to top, left to right

I need something to the left and to the bottom, but if I start in the diagonal then it becomes easy, because I can actually say that if I have initially these values filled in, this is the base case. Then to the left and below, I can fill in this because I know it is values to the left, I know this value to the left and below, so I can fill up this diagonal. In the next step, I can fill up this diagonal, because I have all the values to left below. Now at this entry, I have values to the left and below, so I can fill up this diagonal. So, I can fill it up diagonal by diagonal. This is the order in which I can fill up this table using this inductive definition because this is the way in which the dependency is stored.

(Refer Slide Time: 13:14)

## MMCost(M1,...,Mn), DP

```
def MMC(R,C):
    # R[0..n-1],C[0..n-1] have row/column sizes
    for r in range(len(R)):
        MMC[r][r] = 0
    for c in range(1,len(R)):  # c = 1,2,...n-1
        for r in range(c-1,-1,-1):# r = c,c-1,...,0
            MMC[r][c] = infinity # Something large
            for k in range(r,c) # k = r,r+1,...,c-1
                subprob = MMC[r][k] + MMC[k][c] +
                           R[r]C[k]C[c]
            if subprob < MMC[r][c]:
                MMC[r][c] = subprob
```

This is the code for this particular thing. So, you can go through it and just check the only thing that you need to notice that we have used some.

(Refer Slide Time: 13:31)

## Subproblem dependency

- Cost(i,j) depends on Cost(i,k), Cost(k+1,j) for all  $i \leq k < j$
- Can have  $O(n)$  dependent values, unlike LCS, LCW
- Start with main diagonal and fill matrix by columns, bottom to top, left to right



So what we are doing is, when we compute as we said one entry. Supposing, we are computing this one entry then we want to compute the minimum across many different pairs right this entry this entry and so on. Remember what we did when you computed the maximum longest common sub word, we assume that the maximum was zero, and every time we saw a bigger value we updated it. Here, we want the minimum entry. So,

what we do is we assume the minimum as some large number and every time we see an entry, if it is smaller than the minimum we reduce it.

(Refer Slide Time: 13:57)

## MMCost(M1,...,Mn), DP

```
def MMC(R,C):
    # R[0..n-1],C[0..n-1] have row/column sizes
    for r in range(len(R)):
        MMC[r][r] = 0
    for c in range(1,len(R)):  # c = 1,2,...n-1
        for r in range(c-1,-1,-1):# r = c,c-1,...,0
            MMC[r][c] = infinity # Something large
            for k in range(r,c) # k = r,r+1,...,c-1
                subprob = MMC[r][k] + MMC[k][c] +
                           R[r]C[k]C[c]
            if subprob < MMC[r][c]:
                MMC[r][c] = subprob
```

That is what it is happening here, when we start the loop we assume that the value for  $r$   $c$  is actually infinity. Now what is infinity? well you can take infinity So that we can take for instance the product of all the dimensions that appear in this problem. You know that the total dimension will not be more than that. So, you can take the product of all the dimensions you can take a very large number, it does not matter something related to what your problem as. So, we have not defined it in the code. The important thing is we are computing minimum.

Instead of starting with 0, and updating it when you do maximum; you start with the large value and keep shrinking it. So every time, we find a sub problem which is smaller than the current value that we have seen then we replace that value as the new minimum, so that is all that is important here. Everything else is just a way to make sure that we go it, go through this table diagonal by diagonal, and for each diagonal we scan the row and the column and compute the minimum across all pairs in that row and column.

(Refer Slide Time: 14:50)

## Complexity

Complexity > Table Size



- As with LCS, we have to fill an  $O(n^2)$  size table
- However, filling  $MMC[i][j]$  could require examining  $O(n)$  intermediate values
- Hence, overall complexity is  $O(n^3)$

As with this LCS problem, we have to fill an order  $n$  squared size table, but the main difference between LCS and this is that in order to fill an entry in the LCS thing we have to look at only a constant number of entries. So, order  $mn$  or order  $n$  squared, but the point was each entry takes constant time, so the effort involved is the same as size of the table,  $m n$  table or takes  $m n$  time. Here, unfortunately it is not the case right. We saw that an entry will take time proportional to its distance from the diagonal. In general, that will add an order  $n$  factor.

Though we have order  $n$  squared by two entries actually order  $n$  squared entries, we would have to account for order  $n$  work per entry because each entry has to scan the row to its left in the column below not just one entry away from it. And so this whole thing becomes order  $n$  cubed and not order  $n$  square. So, it is this is some point to keep in mind that there could be problems, where the complexity is bigger than the table size. All the examples we saw before that is the reason to do this example.

In all the examples we saw before, the Fibonacci we had a table which is linear in  $n$  and it took time linear in  $n$  to fill it. For the grid path and for longest common subsequence, we had tables, which are  $m$  by  $n$ , but it took only  $n$  by  $m$  time or  $m$  by  $n$  time to fill that. Here, we have a table which is  $n$  by  $n$ , but it takes  $n$  cube time to fill it up because each entry it requires more than constant time, it actually takes time proportional to  $n$ .

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

**Week - 08**  
**Lecture - 05**  
**Wrap-up, Python vs. other languages**

(Refer Slide Time: 00:02)

## Python vs other languages

- Python is a good programming language to start with because
  - No declaration of names in advance
  - Indentation avoids punctuation — { }, (), ;
  - No explicit memory management
- Are there any down sides to this?

We have come to the last lecture of this course. So, instead of going into more features of Python of which there are many that we have not described. Let us take some time instead to reflect about Python as a language and compare it to some of the other languages which exist. Though you may not be familiar with them, I would like to highlight some aspects of Python which are different from other languages and also argue whether they are better or worst.

Why did we choose Python to do this course? Well, Python is a very good language to start programming for many reasons. One reason is that its syntax is very simple and part of the simplicity comes from the fact that you do not have to declare names in advance, so you do not have to start doing a lot of things before you write your code. You can jump into the Python interpreter for example, and keep writing statements without worrying about what x is and what y is, as you go along you can define values and

manipulate them. So, that makes it very convenient and very nice to start programming because you do not have to learn a lot of things in order to write even basic programs.

The way in which Python programs are laid out with indentation to indicate when a loop begins and when an 'if' block begins. We do not have to worry about a lot of punctuation: braces, brackets, semicolons, which are part and parcel of more conventional languages. Once again this makes a language little easier to learn and a little less formidable for a new comer to look at.

The other thing which is little more technical which we will... I will talk about a little bit in this lecture is that we do not have to worry about storage beyond understanding what is mutable and immutable. If you need to use a name **x**, we use it; if we need a list **l** we use it, if we need to add something **to an l**, we just say append. We never bother about where the space is coming from or where it is going. These are all plus points of Python. So, what are the minus points? So **are there things** that are bad about **Python?**

(Refer Slide Time: 02:05)

## Debugging

- Declaring names helps debug code
  - “Simple” typos are caught by compiler
  - Mistyped name will be “undeclared”
- Static typing — assigning types to names
  - Again catch “simple” typos by type mismatch

The first **thing is** to do with the lack of declarations - the lack of declarations is often a very good thing, because you do not have to worry about writing a lot of stuff before you start programming, but it is also a **bad thing** because many programming errors come

from simple typing mistakes. Very often you need to write x and you write y. Now in Python if you write a y and you assign it a value somewhere where you meant an x, Python will not know that you were supposed to use x and not y because every new name that comes along is just happily added to the family of names which your program manipulates.

These kind of typos can be very very hard to find, and because you have this kind of dynamic name introduction with no declarations, Python makes it very difficult for you as a programmer to spot these errors. On the other hand if you declare these names in advance which happens in other languages like C or C++ or Java. Then if you use a name which you have not declared then the compiler will tell you that this name has not been seen before and therefore something is wrong. So, a miss typed name can be easily caught as an undeclared name if you have declarations.

Whereas, in Python it will just happily go ahead and create a new value for that name and pretend that there are two names now while you think there is only one and create all sorts of unpredictable errors in your later code. The other side of this is typing. So, in Python we have seen that names do not have types they only inherit the type from the value that they have. You could say at some point x equal to 5 and later on assign x to a string and later on assign x to a list and Python will not complain, at each point given the current value of x legal operations are defined as per that value.

Now this is again nice and convenient but it can also lead to errors for the same reason you might be thinking of x as an integer, but somewhere halfway through your program you forgot that it is an integer and start assigning it some different type of value. Now if you had announced to Python that x must always be an integer then this name must only store an integer value, presumably as a compiler it would catch it internally.

A lot of errors are either typos in variable names or misguided usage of names from one type to another, both of these can be caught very easily by compilers if you have declarations of names, both of these get uncaught or they are left uncaught by Python and they allow you to propagate errors and these errors can be very difficult to find. So the down side of having this flexibility about adding names and changing their types,

values as you go along is that debugging large programs requires a lot more care on your part, writing large programs requires care, debugging is very difficult.

(Refer Slide Time: 04:48)

## Classes and objects

- Handle integrity of compound values
- Date is a tuple (day,month,year)
  - Range for day is 1–31, month is 1–12
  - Valid combinations depend on all three fields
    - 29 - 02 is valid only in a leap year
- `d.setdate(d,m,y)` vs separate `d.setd(d)`,  
`d.setm(m)`, `d.sety(y)`

The other part **has** to do with the discussion that we had towards **the second half of** the course about user defined data types in terms of Classes and Objects. So, the first thing that is a direct consequence of not having declarations is that we cannot assert that a name **has** a type. In particular we saw that if we want to use something as a list we have to first declare it to be an empty list, so we have to write something like `l` is equal to this to say that hence forth I can append to it.

The first append or the first operation on the list that I do will have to be legal, for that I have to tell it that it is a list. This is more or less like a declaration except it is not quite a declaration. I am actually creating an empty object of that type. **In** the same way if I want a dictionary I have to give it a completely new name like this. I have to say `d` is equal to an empty dictionary. So, there is no way to assign a type to a name without creating an object of that type.

This is actually a problem with **the** kind of user defined types that we have, for instance it is very convenient to be able to define an empty tree without having to create a tree, a

node. For instance, if you had type declarations you can say that the name t is of type tree and then you can use this value like none - all programming languages have such a value which denotes something that does not exist. You can say that there is not one none, but many nones and by context this none is a none of type tree.

Python also uses it. There is only one value none and you can use none for anything, but when it has none, it has no type - that is the difference. If in Python a name has the value none, it has by definition no type, whereas if you had declarations you can say that this name has a type, it just does not happen to have a value.

And this is typically what you want for an empty node or an empty tree. We had if you remember a very cumbersome way of dealing with this in order to make recursive tree exploration work better, we actually added a layer of empty nodes at the frontier and extended our tree by one layer just so that we could easily detect when we reach the end of a path. Now this can be avoided actually, if you have type declarations, so this is another feature which makes actually... the lack of declaration makes things a little bit more complicated in Python which one doesn't normally come across in beginning programming.

The other thing is much more serious. So, this is more to do with convenience and representation of empty objects, but without declarations you really cannot implement the kind of separation of public and private that you want in an object. Remember our goal in an abstract data type was to have a public interface or a set of functions which are legal for a data type and have a private implementation. For instance, we would have a stack, we might implement as a list, but we would only like pop and push. Same way a queue may be also a list, but we only want the add and remove queue at opposite ends and we do not want to confuse this with the list operations; we do not want things like append and extend to be used indiscriminately.

The other part of it is that we do not want the data itself to be accessible, we do not want to say that if given a point p, I can use p dot x and p dot y from outside and directly update the values. Because, this is sensitive to the fact that tomorrow I might change from x dot y... x and y representation to r and theta, we said that we might have situations

where we might prefer to represent our point using r and theta. Now if some programmer started using point in the days when we are using x and y and started manipulating p dot x and p dot y directly outside the code.

See if it is inside the class code, then as the maintainers of the class we would make sure that wherever we used to use p dot x and p dot y we now use p dot r and p dot theta. So, it is an internal thing we change from x, y to r, theta we internally update all the code within the class to be in terms of r, theta and not x, y. But if somebody outside is using x and y and these values no longer exist then what happens is that for the outside person their code stops working, because we have changed an internal implementation of point.

And this is a very dangerous situation which happens quite often, and this is where it is important to separate public from private if they do not have any access to the private implementation of the point then they cannot use p dot x and p dot y. Outside this problem is avoided. So just to reiterate, supposing we have a stack implemented as a list and we only allow public methods push and pop, a person who is exploiting this fact that it is a list could directly add something to the end and violate the heap property for instance or the stack property.

So, we could get situations where the data structure is compromised because the person is using operations which are legal for the private implementation, but illegal for the abstract data type which we are trying to present to the user. So for this, the only way we can get around this is to actually have some way of saying that these names are private and these names are public. Now it is a not that Python does not have declarations.

If you remember Python has this global declaration which allows you to take an immutable value inside a function and say it refers to the same immutable value outside. There are situations in which Python allows declarations, but there are many other things where it could allow declarations and make things more usable, but it does not and this is one example.

In languages like java or... you will find a lot of declarations saying private and public, and this looks very bureaucratic but it is really required in order to separate out the

implementation from the interface. Actually, in an ideal world, the implementation must be completely private. So, you should never be able to look at x and y directly. For instance if you want x there should be a function called get x which gives you the x value, there is function called set x which sets the x value.

This may look superficially the same as saying  $p \cdot x$  equal to v, but the difference is that we do not know actually there is an x, x is an abstract concept for us. So, x coordinate is just... it is a property of the point which we can set. Now how we set it is through this function. So, if inside the functions we start setting r theta as the representation, then changing the x value will correspondingly change r and theta indirectly. So, when we say get x, for instance, it does not actually read the x value, it gets  $r \cos \theta$  and we say set x it will change the r to account for the new x and recompute the theta.

In this way if we have only these functions to access the thing then we have only conceptual values inside and these conceptual values are manipulated through these functions and we do not know the actual representation. So this is the ideal world, everything is hidden and we only have these functions, but this is cumbersome for every part of the data type we have to use these functions. And partly the reason why we have to have this private public declaration is that the programmers are not happy with having to always invoke a function, sometimes they would like to directly assign. They would like some parts of the data type to be public. So, they can say  $p \cdot x$  equal to 7.

But in general this is the style that one would ideally advocate for object oriented programming - make all the internal names and variables private and only allow restricted access, so that they are used in the appropriate way and the use does not get compromised if the internal representation changes. If we move x, y to r, theta, get x and set x will still work, whereas  $p \cdot x$  may not be meaningful anymore.

Another reason to have this style of accessing values is sometimes you do not want individual values to be actually accessed individually. Supposing, we have a date, a date is typically a three component field which has the day, month and year. And these have

their own ranges: the valid days for a month range from 1 to 31 and the months range from 1 to 12, but not every month has 31 days.

The valid combination for a day and month depends on both these quantities and in fact it depends on the year also because we cannot have 29th of February unless it is a leap year. So, if we update day or month we have to be careful that we are updating it legally, we can start with the month February with the legal date like 15 and then change the date from 15 to 31, and now end up with an illegal date which is 31st of February. So what we need actually is to have a composite update operation which sets the date by providing all three values, rather than supplying three separate operations to update the three fields separately.

Even if you have these functions you do not always want to give these functions individually, because there might be some constraints between the values which have to be preserved and you can preserve those by controlling access to them. So you can say, you cannot set the day separately and the month separately, you must set them together. This is how one other reason why it is good to keep all the implementation private, not allow direct update and then control the way in which you update the values.

(Refer Slide Time: 14:09)

## Storage allocation

- Python needs to allocate space dynamically
  - Each assignment to a name could a new type
  - Name declarations allow some static allocation
    - Still need dynamic allocation for lists, trees etc that grow at run time
    - Static arrays can optimize access time: base address plus offset

Now let us come to Storage allocation. How the names and values we use in our program are actually allocated and stored in memory so that we can look them up. Because in python we use names on the fly we keep coming up with names and the values keep changing, Python cannot decide **in** advance that it needs a space for x, for one integer because tomorrow this x might be a list and it does not even know which names are coming. So Python has to allocate space always in a dynamic manner, it **has** to keep as you use a name, it **has** to find space for it and the space requirement may change if the value changes **it's** type.

On the other hand if I have a static declaration, then if I say that i is an integer and j is an **integer** and k is an **integer** then the compiler can directly declare in advance some space in **the** memory to be **reserved for** i, j and k. Now this is particularly useful for arrays, we **mentioned** that in **an** earlier lecture, the difference between arrays and **lists**.

In a statically declared situation, an array of size hundred will actually be allocated as a block of hundred contiguous values without gaps. This means that I can get to any entry in the array by looking at... the knowing the first value and then how many values **to** skip. So, if I want the 75th value, I can go to the first value and calculate where the 75th value will be if I just jump over that many values and get **there** directly. This gives us what is called a Random Access Array. It does not take any more time to get **to** the first element **than** the last element.

Whereas in a list, as we saw even in our object based implementation to get to the ith element we have to start with the head and **go** to the **first, second, third,** so it takes **time** proportional to the position. If we have static allocation we can also exploit the fact that we can get peculiar random access arrays which Python actually does not have. Now just because we have declaration does not mean everything is declared statically.

Even in languages like Java and C++, we would do this object oriented style where we would have a template for a class and then we **would** create object of this class dynamically as a tree or list grows and **shrinks, we** will create more objects or remove them. So there is a dynamic part also, but it is exclusively for this kind of user defined data types and the static part takes care of all **the** standard data types that you use for

counting and various standard things and **particular** arrays very often. **If** you know in advance, you need a block of data of a particular size, arrays are much more efficient than **lists**.

(Refer Slide Time: 16:39)

## Dynamic storage

- What happens when we execute `del(x)`?
- Or when we delete a list node by bypassing it?
- Do these “dead” values continue to use memory?

One part of storage allocation is getting it, the other part is de allocation, giving it up. So, we saw that in Python you can remove an element from the list by saying `del(l[0])`. In general you can un allocate any **name's** value by saying `del(x)`. So what happens when we... does this give up **the** storage? When we say `del(x)` we are saying that the space that **x** is currently using for **it's value** is no longer needed. So what happens to that space? Who takes care of it? Similarly when we had a list, if you remember, when we wanted to delete a list, if we had a sequence of things and we wanted to delete a list, we did not actually delete it, we just bypassed it, we said that the first element points to the third.

Now, logically this thing is no longer part of my list but where has it gone? Has it gone anywhere or is it still there in my memory, blocking space? So what happens with the dead values which we have unset by using del or we have bypassed by manipulating the links inside an object and so on.

(Refer Slide Time: 17:40)

## Garbage collection

- Python, Java and other languages reclaim space using automatic “garbage collection”
  - Periodically mark all memory reachable from names in use in the program
  - Collect all unmarked memory locations as free space
  - Run time overhead to schedule garbage collector
- In C, need to explicitly ask for and return dynamic memory

So it turns out that languages like Python, also other languages like Java... this has nothing to do with declaring variables, it has to do with how space is allocated. They use something called Garbage collection. Garbage in the programming language sense is memory which has been allocated once, but it is not being used anymore and therefore is not useful anymore, because we cannot reach it in some sense. So, it is like that list example: there was an element in our list which we could reach from the head. At some point we deleted it, now we can no longer reach that value nor can we reuse it because it has been declared to be allocated, so it is garbage.

So roughly speaking how does garbage collection work? Well, what you do is you imagine that you have your memory and then you have somewhere some names, n, m, x, l and so on in your program and you know where these things point, this is somewhere here, this is somewhere here, this is somewhere here, this is somewhere here. So, you start with the names that you have and you go on... you mark this thing, you say this is mine, this is mine, this is mine, and this is mine.

Now this could be a list, I could be a list, it could be that this in turn points to the next element in the list, so it goes here so then you point to this and say this is also mine. It is what the names point to plus what their pointed values point to, you keep following this

until you mark all the memory that you can actually reach from your name, so the names in your program and all the memory that they can indirectly reach. Now I can go through and say that everything which has not been marked is not in use, so I can go and explicitly free it. This is the second phase. You collect all the unmarked memory locations and make them free and proceed.

Now, this is a process which runs in parallel with your program, at some point logically speaking you have to stop your program and mark the memory, release space and then resume your program. So there is an overhead. Some languages like C do not have this garbage collection built in. So in C if you need dynamic memory, so remember that all the things that you declare in advance are allocated statically, you cannot undo them, you cannot say hence forth I do not need them.

But if you have something like a list or a tree where you are growing and shrinking, you will ask for a memory, saying, I want memory worth one node, then you will populate it. When you delete something it is your responsibility as a programmer to say this was given to me sometime back, please take it back. You have to, as a programmer, ask for dynamic memory and more importantly when you are no longer using it, return it back. So, C has this kind of programmer driven manual memory allocation.

(Refer Slide Time: 20:23)

## Memory leaks

- Manual memory allocation is error prone
- Forgetting to return junk space to free list results in memory “leaking” out of the system
  - Performance suffers over time as space shrinks
- All modern languages use garbage collection
  - Run time overhead more than compensated by reduction of errors due to manual management

This is quite error prone. As you can imagine there might be just a simple case where a programmer writes a delete operation in a tree or a list and forgets to give the memory back. So, this means that every time something is added and removed from a list or a tree, in such a program, that thing will be residing in memory leaving it to be used when it is actually not used. So over a period of time this thing will keep filling up. If you take the flip side and you look at the free memory you think of the free memory as a fluid then the free memory is shrinking. So this is called a Memory leak, the memory is leaking out of your system, it is not really leaking out, it is getting filled out.

This is the terminology. So you might see somewhere in some text, the word memory leak. It is just referring to the fact that memory that has been allocated to a program is not being properly deallocated when it is no longer in use, and the symptom of this will be that as the program runs longer and longer, it will start taking up more and more memory and making the whole program very much more sluggish. So, the performance will shrink, will suffer over time as the space shrinks.

So, virtually speaking all modern languages use garbage collection because it is so much simpler. Though there is a runtime overhead with this mark and release kind of mechanism, the advantages that you get from not having to worry about it and not trusting the programmer as such, that you avoid the memory leaks by making sure that your garbage collection works rather than relying on the good sense of the programmer to make sure that all the memory allocated is actually deallocated.

(Refer Slide Time: 21:58)

## Functional programming

- Many features of Python are modelled on functional programming
  - `map`, `filter` and other “higher order” functions
  - List comprehensions

For a final point of this last lecture, let us look at a completely different style of programming which is called Functional programming. So, Python and other languages we have talked about **are** what are called imperative programming languages. So in an imperative programming language you actually give a step by step process to compute things. You assign names to keep track of intermediate values, you put things in **lists** and then you have to basically do the mechanical computation: we have to simulate it more or less, so a sequence of instructions. So you have to know a mechanical process for computing something more or less before you can implement it.

Functional programming is something which tries to take the kind of inductive definitions that we have seen directly at **heart** and just use these as the **basis** for computation. So, you will directly specify inductive functions in a declarative way what to conclude, so here is a typical declaration. For example, this is a Haskell style declaration. So, the first line with this double colon says that factorial **is** a function and this is **its** type, it takes **an integer** as input and **produces an integer as** output. This is saying that factorial is a box that looks like this. **It takes ints and produces ints**, so this is the thing that you **have**. And then it gives you the rule to compute it. It **says** factorial of 0 is 1 and then the rules are **processed in order**, so **if** it comes to the second rule, it means at this point that n is not 0. If n is not 0, then n times factorial n minus 1 is the answer.

The actual way that Haskell works is by rewriting. We will not get into that, but the main point is that there is no mention here about the intermediate names, it is just taking the parameters passed to the function and inductive or recursive calls and how to combine them. So here is another example. If you want to add up the elements in a list, so this Haskell's type for a list of integers, so it takes a list of integers and produces an integer. So, you would say that for a list which is empty, the sum is 0 and again coming here sum is not... this is not empty if it comes here because we go in order.

If the first list does not match, the second list must be not empty, then it has functions such as head or tail to take the first element or the last element. So, if you have a non empty list, the sum is given by taking the first element and then inductively adding it to the inductive result of computing the rest. This is a completely different style of programming which is called Declarative programming and you can look it up. It is very elegant and it has its own uses and features.

Python has actually borrowed some of its nice features from functional programming. And one of them in particular that we have seen is this use of map and filter and in general the idea that you can pass functions to other functions. So, these are all naturally coming from functional programming and map and filter which allows to take a function and apply to every element of a list. And then resulting from this, very compact way of writing lists using list comprehensions in one line, combining map and filter. These are features of functional programming which are integrated into Python.

(Refer Slide Time: 25:12)

## Summary

- No programming language is “universally” the best
  - Otherwise why are there so many?
- Python’s simplicity makes it attractive to learn
  - But also results in some limitations
- Use the language that suits your task best
- Learn programming, not programming languages!

To conclude, one can never say that one programming language is the best programming language, because if **there** were a best programming language, then obviously everybody **would** be using that language. The very fact that there are so many programming languages **around, it** is obvious that no programming language is universally better than every other. So what happens is that, you choose a language which is best for you.

In particular here we were trying to learn programming and various aspects of programming **data structures, algorithms. And** Python being a simple language to start working with and to use and **it's** nice built-in things like dictionaries and stuff like that, exception handling, are done in a very nice way, so it makes it very attractive to learn. But as we saw, the same things that make it attractive to learn, the lack of declarations also limit **it's expressiveness. We** cannot talk about privacy in terms of implementations or objects and so on.

So the moral of **the** story is that **when** you have a programming task, you know what you want to do, look around for **the language** that suites your task the best. Do not be afraid of learning programming languages to do the task, because once you have learnt one programming **language, it** is actually not that much difference between one and the other, it is just minor differences.

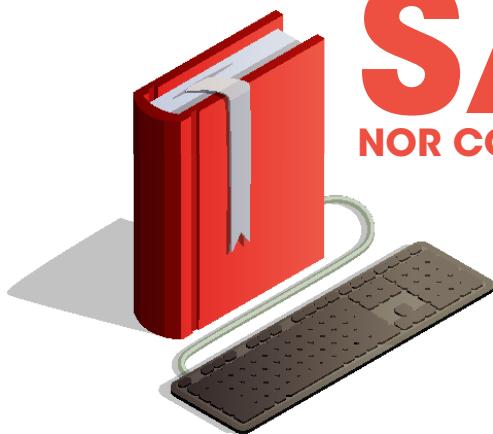
Of course, there are different styles like functional programming which look very different, but more or less if you have a little bit of background, you can switch programs from one language to another by just looking up the reference manual and working with it. Of course, if you use a programming language long enough, then you should be careful to learn it well, but for many things you can just get by on the fly by just translating one language to another.

So, the main message is that you should focus on learning programming. Learning how to translate, first of all coming up with good algorithms, how to translate algorithms into effective implementations, what are the good data structures, so your focus should be on algorithms, data structures, the most elegant way in which you can phrase your instructions. And then worry about the programming language.

It is a mistake to sit and learn a programming language; nobody learns a programming language, you learn features of a programming language and put them to use as you come up with good programs. So, with this I wish you all the best and I hope that you have a fruitful career ahead in programming.

Thank you.

**THIS BOOK  
IS NOT FOR  
SALE  
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in