# PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

**Prof. Madhavan Mukund**
Computer Science & Engineering
Chennai Mathematical Institute

# INDEX

## *Week 5*



## *Week 6*



## *Week 7*



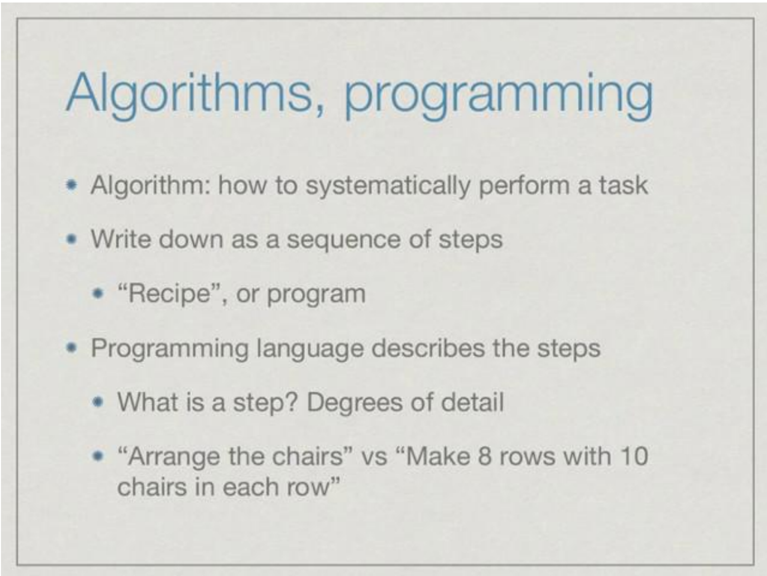
## *Week 8*

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

**Week - 01**
**Lecture - 01**
**Algorithms and Programming: simple gcd**

Welcome to the first lecture on the course on Programming Data Structures and Algorithm in Python.

(Refer slide Time: 00:10)

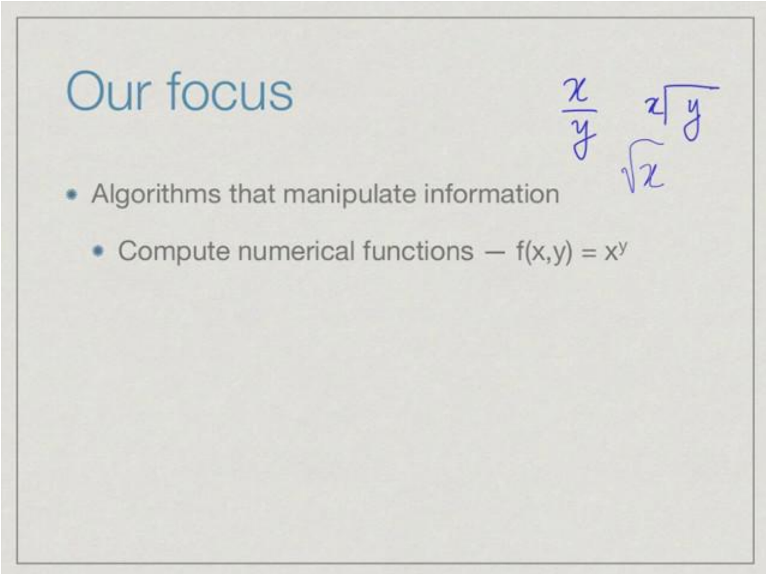



Let's start with the basic definition of what we mean by an algorithm and what programming is. As most of you probably know, an algorithm is a description of how to systematically perform some task. An algorithm consists of a sequence of steps which can we think of as a recipe in order to achieve something. So, the word recipe of course, comes from cooking where we have list of ingredients and then a sequence of steps to prepare a dish. So, in the same way an algorithm is a way to prepare something or to achieve a given task. So, in the context of our thing, a recipe will is what we call a program. And we write down a program using a programming language. So, the goal of programming language is to be able to describe the sequence of steps that are required and to also describe how we might pursue different sequences of steps if different things happen in between.

The notion of a step is something that can be performed by whatever is executing the algorithm. Now a program need not be executed by a machine although that will the typical context of computer programming were we expect a computer to execute our steps. A program could also be executed by a person. For instance, supposing the task at hand is to prepare a hall for a function. So, this will consists of different steps such as a cleaning the room, preparing the stage, making sure the decoration are up, arranging the chairs and so on. This will be executed by a team of people. Now depending on the expertise and the experience of this group of people, you can describe this algorithm at different levels of detail.

For instance, an instruction such as arrange the chair would makes sense if the people involved know exactly what is expected. On the other hand, if this is a new group of people who have never done this before; you might need to describe to step in more detail. For instance, you might want to say that arrange the chairs in the 8 rows and put 10 chairs in each row. So, the notion of a step is subjective, it depends on what we expect of the person or the machine which is executing the algorithm. And in terms of that capability, we describe the algorithm itself.

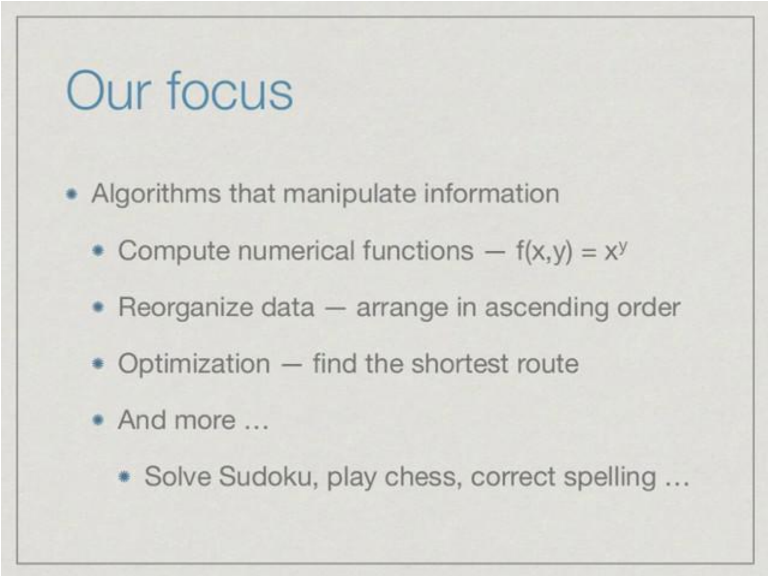
(Refer slide Time: 02:44)




Our focus in this course is going to be on computer algorithms and typically, these algorithms manipulate information. The most basic kind of algorithm that all of us are familiar with from high school is an algorithm that computes numerical functions. For instance, we could have an algorithm which takes two numbers x and y, and computes x to the power y. So, we have seen any number of such functions in school.

For example, to compute square root of x, so what we do in school is we have complicated way to compute square root of x or we might have x divided by y where we do long division and so on. These are all algorithms, which compute values given one or more numbers they compute the output of this function.

(Refer slide Time: 03:35)

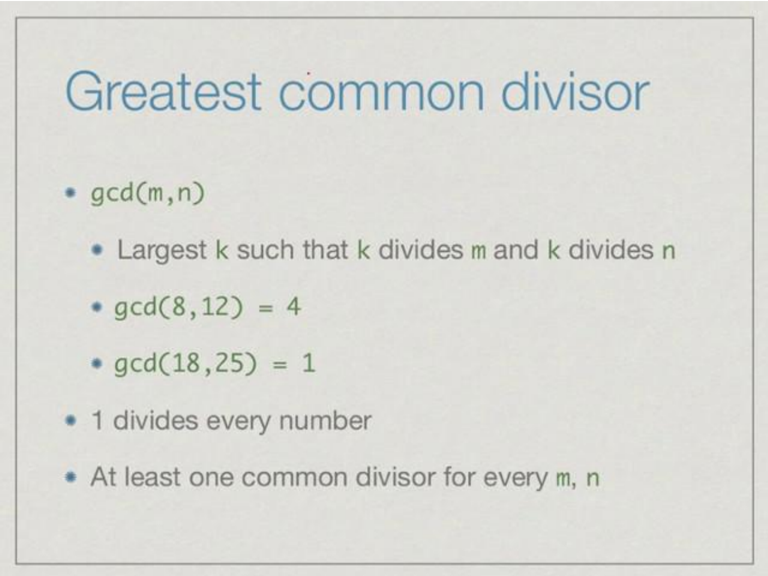



But all of us who have used computers know that many other things also fall within the realm of computation. For instance, if we use a spreadsheet to arrange information and then we want to sort of column. So, this involves rearranging the items in the column in some order either in ascending order or descending order. So, reorganizing information is also a computational task and we need to know how to do this algorithmically. We also see computation around us in the day today's life. For instance, when we go to a travel booking site and we try to book a flight from one city to another city it will offer to arrange the flights in terms of the minimum time or the minimum cost. So, these are optimization problems. This involves also arranging information in a particular way and then computing some quantity that we desire.

In this case, we want to know that a we can get from a to b, and b among all the ways we can get from a to b we want the optimum one. And of course, there are many, many more things that we see day today, which are executed by computer programs. We can play games. For instance, we can solve Sudoku or we can play chess against a program. When we use the word processor to type a document or even when we use our cell phones to type sms messages, the computer suggests correction in our spelling.

We will look at some of these things in this course, but the point is to note that a program in our context is anything that looks at information and manipulates it to a given requirement. So, it is not only a question of taking a number in and putting the number out. It could involve rearranging things. It could involve computing more complicated things. It could involve organizing the information in a particular ways, so these computations become more tractable and that is what we call a data structure.

(Refer slide Time: 05:36)

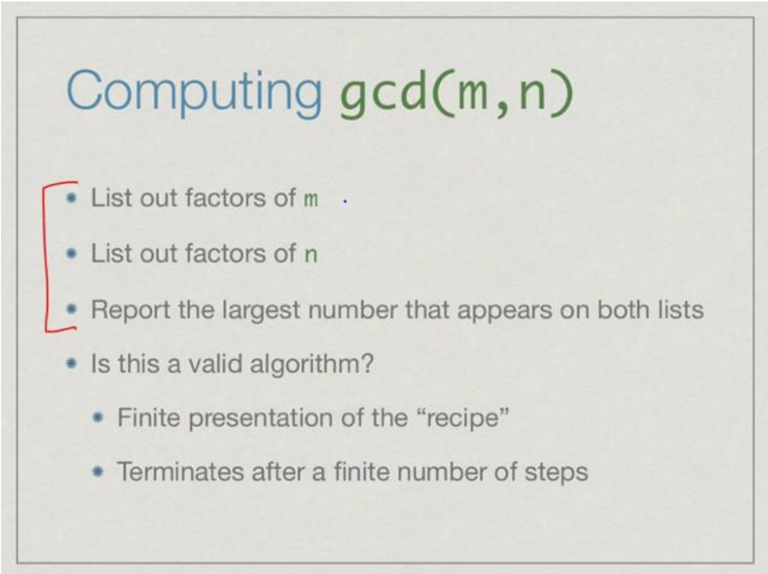



So to illustrate this let us look at the function which most of us have seen and try to understand the algorithmically. So, the property that I want to compute is the greatest common decide divisor of two positive integers m and n. So, as we know a divisor is a number that divides. So k is a divisor of m; if I can divide m by k and get no reminder. So, the greatest common divisor of m and n must be something which is a common divisor. So, common means it must divide both and it must be the largest of these. So, if the largest k such that k divides m and k also divides m.

For instance, if we look at the number 8 and 12, then we can see that 4 is the factor of 8, 4 is the divisor of 8, 4 is also divisor of 12, another divisor of 12 is 6, but 6 is not a divisor of 8. So, if we go through the divisors of 8 and twelve it is easy to check that the largest number that divides both 8 and 12 is 4. So, gcd of 8 and 12 is 4. What about 18 and 25. 25 is 5 by 5. So, it has only one divisor other than 1 and 25, which is 5. And 5 is not a divisor of 18, but fortunately 1 is a divisor of 18. So, we can say that gcd of 18 and 25 is 1; there is no number bigger than 1 that divides both 18 and 25. Since 1 divides every number, as we saw in the case of 18 and 25, there will always be at least one common divisor among the two numbers.

The gcd will always be well defined; it will never be that we cannot find the common divisor and because all the common divisors will be numbers, we can arrange them from smallest to largest and pick the largest one as the greatest common divisor. So, given any pair of positive number m and n, we can definitely compute the gcd of these two numbers.

(Refer slide Time: 07:39)

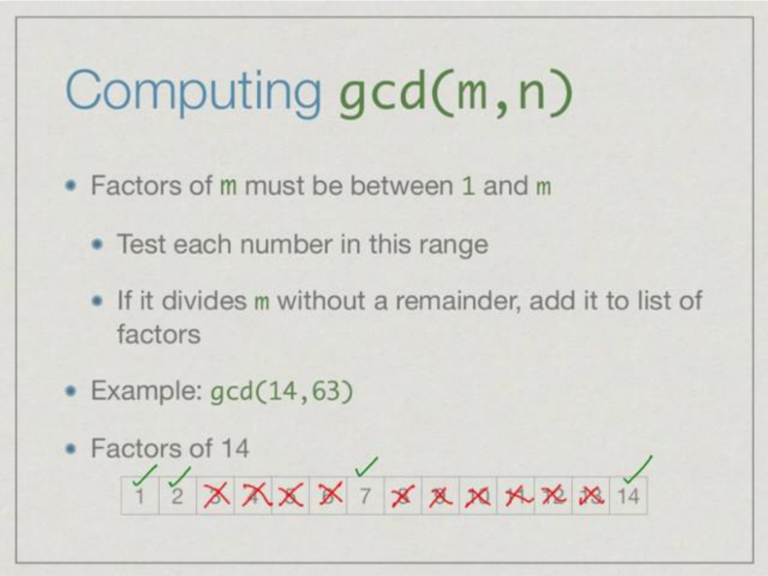



So, how would one go about computing gcd of m, n? So, this is where we come to the algorithmic way, we want to describe the uniform way of systematically computing gcd of m n for any m or any n. So, here is a very simple procedure. It is not the most efficient; we will see better once as we go along. But if we just look at the definition of gcd it says look at all the factors of m, look at all the factor of n and find the largest one which is the factor of both. So, the naive way to do this would be first list out factors of m, then list out all the factor of second number n and then among these two lists report the largest number that appears in both lists. This is almost literally the definition of gcd.

Now question is does this constitute an algorithm. Well, at a high level of detail if we think of list out factors as a single step, what we want from an algorithm are two things. One is that the description of what to do must be written down in a finite way. In the sense that I should be able to write down the instruction regardless of the value m and n in such a way it can read it and comprehend it once and for all.

Here is very clear, we have exactly three steps right. So, we have three steps at constitute the algorithm so it certainly presented in a finite way. The other requirement of an algorithm is that we must get the answer after a finite number of steps. Of this finite number of steps may be different for different values of m and n, you can imagine that if you have a very small number for n there are not many factors they are the very large number for n you might have many factors. So, the process of listing out the factors of m and n may take a long time; however, we want to be guaranteed that this process will always end and then having done this we will always able to find the largest number that appears in both lists.

(Refer slide Time: 09:45)

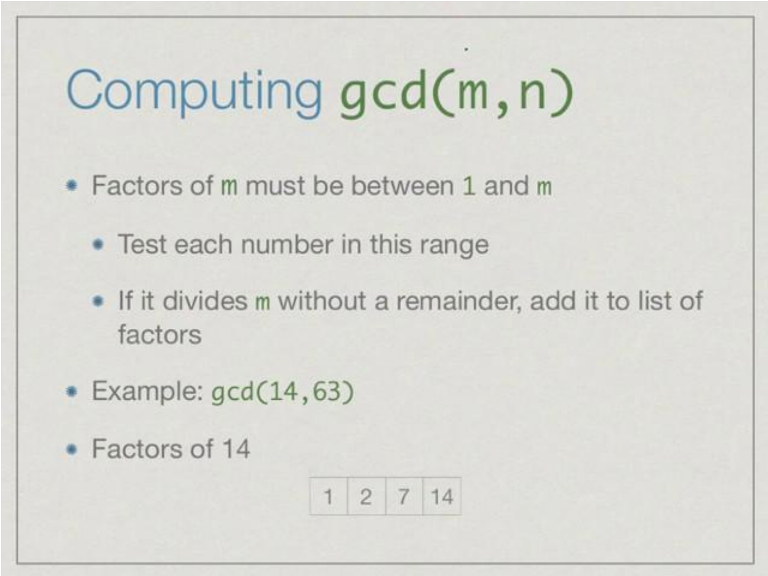


To argue that this process is well defined all we need to realize is that the factors of m must be between 1 and m. In other words, we although there are infinitely many different possibility as factors we don't have to look at any number bigger than m, because it cannot go into m evenly. So, all we need to do to compute the factors of m is to test every number is range one to m and if it divides m without a reminder, then we add it to list of factors. So, we start with empty list of factors and we consider it on 1, 2, 3, 4 up to m and for each such number we check, whether if we divide m by this number we get a reminder of 0 we get a reminder of 0 we add it to the list.

Let us look at the concrete example, let us try to compute the gcd of 14 and 63. So, the first step in our algorithm says to compute the factors of 14. So, by our observation above the factors of 14 must lie between one and 14 nothing bigger than 14 can be a factor. So, we start a listing our all the possible factors between one and 14 and testing them. So, we know of course, that 1 will always divide; in this case 2 divides 14, because 14 dived by 2 is 7 with no remainder. Now 3 does not divide, 4 does not divide, 5 does not divide, 6 does not divide; but 7 does, because if we divide 14 by 7 we get a remainder of 0. Then again 8 does not divide, nine does not divide and so on.

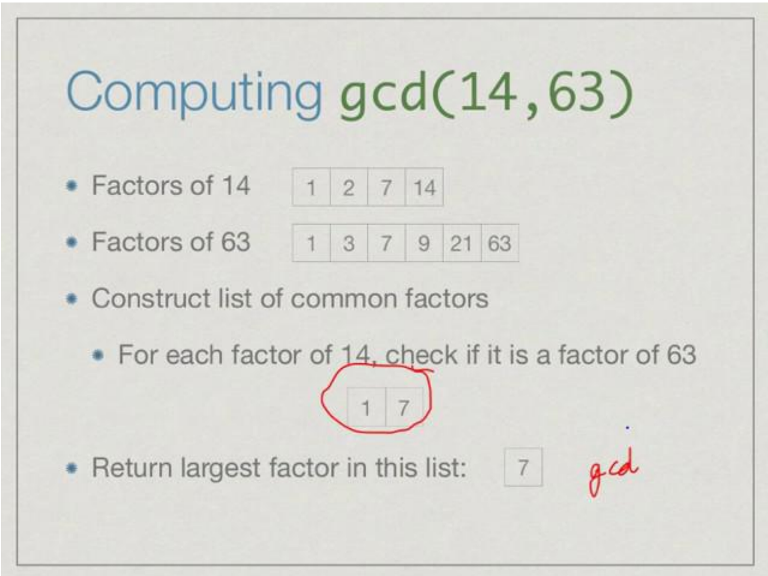
And finally, we find that the only other factor left is 14 itself. So for every number m - 1 and m will be factors and then there may be factors in between.

(Refer slide Time: 11:35)




So, having done this we have identified that factors of 14 and these factors are precisely the 1, 2, 7 and 14.

(Refer slide Time: 11:40)



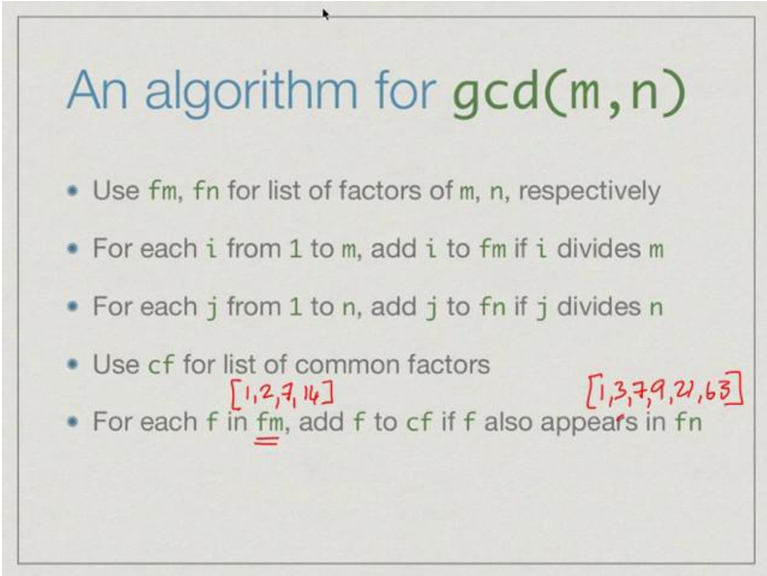

The next step in computing the gcd of 14 and 63 is to compute the factors of 63. So, in the same way we write down the all the numbers from one to 63 and we check which ones divide. So, again we will find that 1 divides, here 2 does not divide; because 63 is not even, 3 does divides, then we find a bunch of numbers here, which do not divide. Then we find that 7 divides, because 7 9's are 63. Then again 8 does not divides, but 9 does. Then again there are large gap of numbers, which do not divide. And then 21 does divide, because 21 3's are 63. And then finally, we find that the last factor that we have is 63. So, if we go through this systematically from one to 63 crossing out each number which is not a factor we end up with the list 1, 3, 7, 9, 21 and 63.

Having computed the factors of the two numbers 14 and 63 the next step in our algorithm says that we must find the largest factor that appears in both list. So, how do we do this, how do we construct a list of common factors. Now there are more clever ways to do this, but here is a very simple way. We just go through one of the lists say the list of factors of 14 and for each item in the list we check it is a factor of 63.

So, we start with 1 and we say does 1 appear as a factor of 63. It does so we add to the list of common factors. Then we look at 2 then we ask does it appear; it does not appear so we skip it. Then we look at 3 and look at 7 rather and we find that 7 does appear so we add 7. Then finally, we look at 14 and find that 14 does not appears so we skip it. In this way we have systematically gone through 1, 2, 7 and 14 and concluded that of these only 1 and 7 appear in both list.

And now having done this we have a list of all the common factors we computed them from smallest to biggest, because we went to the factors of 14 in ascending order. So, this list will also be in ascending order. So, returning the largest factors just returns the right most factor in this list namely 7. This is the output of our function. We have computed the factors of 14, computed the factors of 63, systematically checked for every factor of 14, whether it is also a factor of 63 and computed the list of common factors here and then from this list we have extracted the largest one and this in fact, is our gcd. This is an example of how this algorithm would execute.

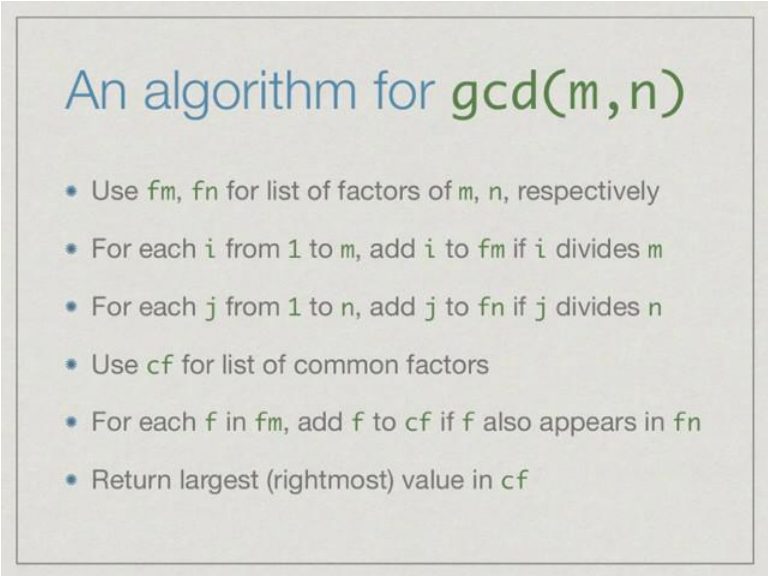
(Refer slide Time: 14:20)




If you have to write it down in little more detail, then we could say that we need to notice that we need to remember these lists, right, and then come back to them. So, we need to compute the factors of 14 keep it side we need to write it down somewhere we need to compute the factor of 63 write it down somewhere and then compare these two lists. So, in other words we need to assign some names to store these. Let us call fm for factors of m and fn factors of n as the names of these lists. So, what we do is that we run through the numbers one to m. And for each i, in this list 1 to m we check, whether i divide m, whether m divided by i as reminder 0 and if so we add it to the list factors of m or fm. Similarly, for each j from 1 to n we check, whether j divides n and if so we add it to the list fn.

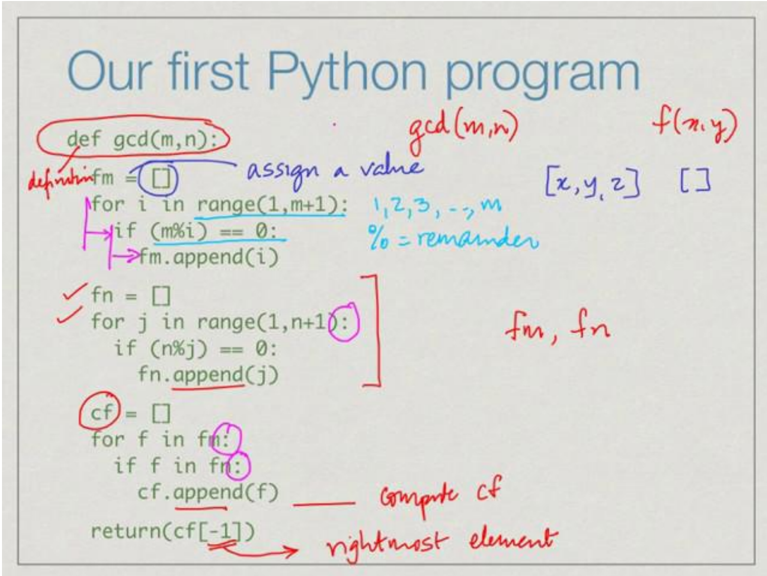
Now we have two lists fm and fn which are the factors of m and factors of n. Now we want to compute the list of common factors, which we will call cf. So, what we do is for every f that is a factor of a first number, remember in our case was 14 where each f so we ran through 1, 2, 7 and 14 in our case right. So, for each f is list we add f to the list of the common factors if it also appears in the other list. So, in the other list if you number is 1, 3, 7, 9, 21 and 63. So, we compare f with this list and if we find it we add it to cf.

(Refer slide Time: 15:58)




And having done this now we want to return the largest value of the list of common factors. Remember that one will always be a common factor. So, the list cf will not be empty. There will be at least one value, but since we add them in ascending order since the list fm and fn, where constructed from 1 to m and 1 to n the largest value will also be the right most value. This gives us a slightly more detailed algorithm for gcd. It is more or less same as previous one except spells out little more details how to compute the list of factors of m and how to compute the list of factors of n and how to compute the largest of common factor between these two lists. So, earlier we had three abstract statements now we are expanded out into 6, slightly more detailed statements.

(Refer slide Time: 16:47)




This already gives us enough information to write our first python program. Of course, we will need to learn little more, before we know how to write it, but we can certainly figure out how to read it. So, what this python programming is doing is exactly what we described informally in the previous step. The first thing in the python program is a line which defines the function. So, we are defining a function gcd of m comma n. So, m and n are the two arguments which could be any number like any function. It's like when we read f of x y in mathematics it means x and y are arbitrator values and for every x and y do something depending on the values that we a call the function with. So, this says that this is a definition, so def for definition of a function gcd m, n.

Now the first step is to compute the list of factors of m. In python we write a list using square brackets. So, list is written as x y z and so on. So, the empty list is just an open bracket and a square close bracket. So, we start off with an empty list of factors. So, this equality means assign a value. So, we assign fm the list of factors of m to be the empty list. Now we need to test every value in the range 1 to n.

Now python has a built in function called range, but because of we shall see, because of peculiarity of python this returns not the range you except, but one less. So, if I say give the numbers in the range 1 to n plus 1, it gives me in the range one to m, one up to the upper limit, but not including the upper limit. So, this will say that i takes the values one two three up to m. For each of these values of i, we check whether this is true. Now percentage is the remainder operation.

It checks whether remainder of m divided by i is 0. If the remainder of m divided by i is 0 then we will append i to the list fn, we will add it to the right append is the English word which just means add to the end of the list. So, we append i to n. So, in this step, we have computed fm. This is exactly what we wrote informally in the previous example we just said that for each i from 1 to m add i to fm if i divides m and now we have done it in python syntax. So, we have defined an empty list of factors and for each number in that range we have checked it is a divisor and then add it.

And now here we do the exactly the same thing for n. So, we start with the empty list of factors of n for every j in for this range if it divides we append it. Now, at this point we have two list fm and fn. Now, we want to compute the list of common factors. So, we use cf to denote the list of common factors. Initially there are no common factors. Now, for every factor in the first list if the factor appears in the second list then we append it to cf.

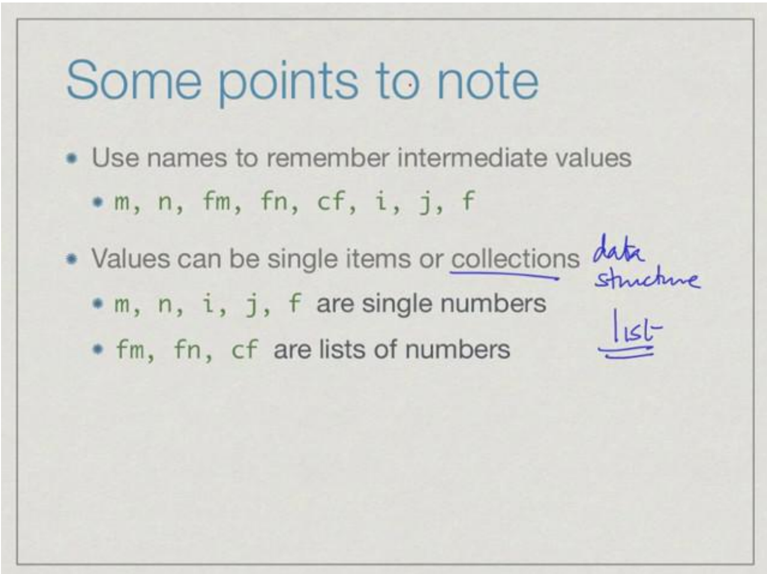
So, the same function append is being use throughout. Just take a list and add a value. Which value? We add the value that we are looking at now provided it satisfies the conditions. So, earlier we were adding provided the divisor was 0 uh the remainder was 0, now we are adding it provided it appears in both list. For every f in the first list if it appears in the second list add it.

After this we have computed fm, cf. And now we want the right most element. So, this is just some python syntax if you see which says that instead of, if we start counting from the left then the number the positions in the list are number 0, 1, 2, 3, and 4. But python has a shortcut which says that you want to count from the right then we count the numbers as minus 1, minus 2 and so on. So, it says return the minus 1'th element of cf which in Python jargon means return the right most element. So, this is the right most element.

At this point it is enough to understand that we can actually try and decode this code this program even though we may not understand exactly why we are using colon in some places and why we are pushing something. See notice that are other syntactic things here, so there are for example, you have these punctuation marks, which are a bit odd like these colons. Then you have the fact that this line is indented with respect to this line, this line is indented to this line.

These are all features that make python programs a little easier to read and write then programs in other languages. So, we will come to these when we learn python syntax more formally. But at this point you should be able to convince yourself that this set of python steps is a very faithful rendering of the informal algorithm that we wrote in the previous slide.

(Refer slide Time: 22:08)

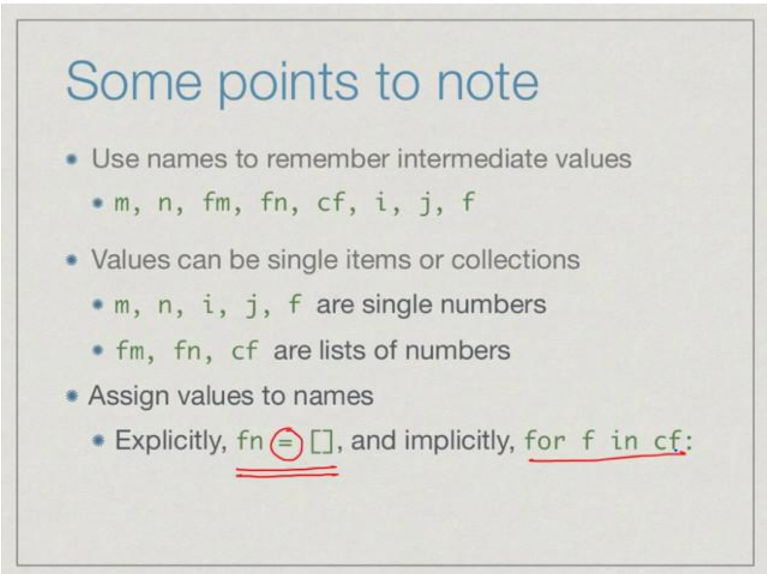



Let us note some points that we can already deduce from this particular example. So, the first important point is that we need a way to keep track of intermediate values. So, we have two names to begin with the names of our arguments m and n. Then we use these three names to compute this list of factors and common factors and we use other names like i, j and f. In order to run through these. We need i to run from 1 to n. We need j to run from 1 to n. Of course, we could reuse i. But it is okay. We use f to run through all the factors in cf. So, these are all ways of keeping track of intermediate values. The second point to note is that a value can be a single item.

For example, m n are numbers, similarly i, j and f at each step are numbers. So, these will be single values or they could be collections. So, there are lists. So fm is a list, fn is a list. So, it is a single name denoting a collection of values in this case a list a sequence has a first position and next position and a last position. These are list of numbers.

One can imagine the other collections and we will see them as we go along. So, collections are important, because it would be very difficult to write a program if we had to keep producing a name for every factor of m separately. We need a name collectively for all the factors of m regardless of how big m is. These names can denote can be denote single values or collections of values. And a collection of values with the particular structure is precisely what we call data structure. So, these are more generally called data structures. So, in this case the data structure we have is a list.

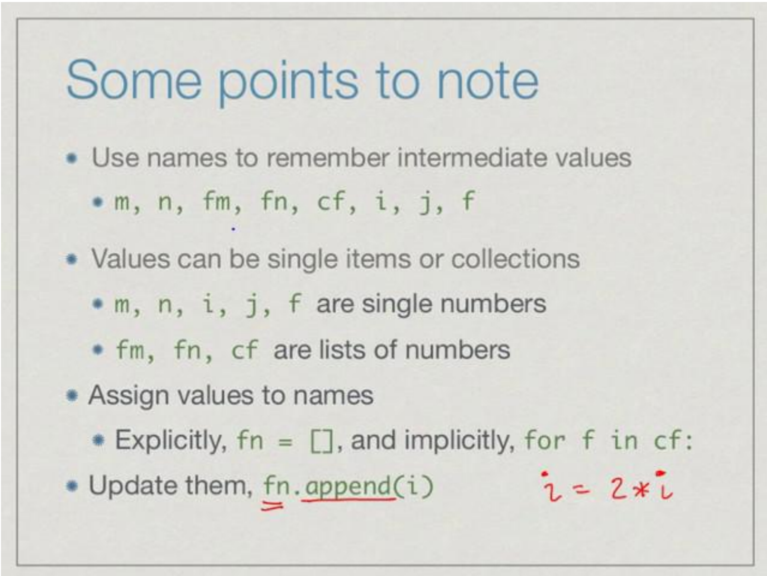
(Refer slide Time: 23:56)




What can we do with these names and values well one thing is we can assigned a value to a name. So, for instance when we write fn is equal to the empty list we are explicitly setting the value of fn to be the empty list. This tells two things this says the value is empyt list, so it is also tells python the fn denotes the lists these are the two steps going on here as we see.

And the other part is that when we write something like for each f in the list cf, which is implicitly saying  that take every value in cf and assign it one by one to the values f to the name f. Right though they do not have this equality sign explicitly implicitly this is assigning the new values for f as we step the list cf right. So, the main thing that we do in a python program is to assign values to names.

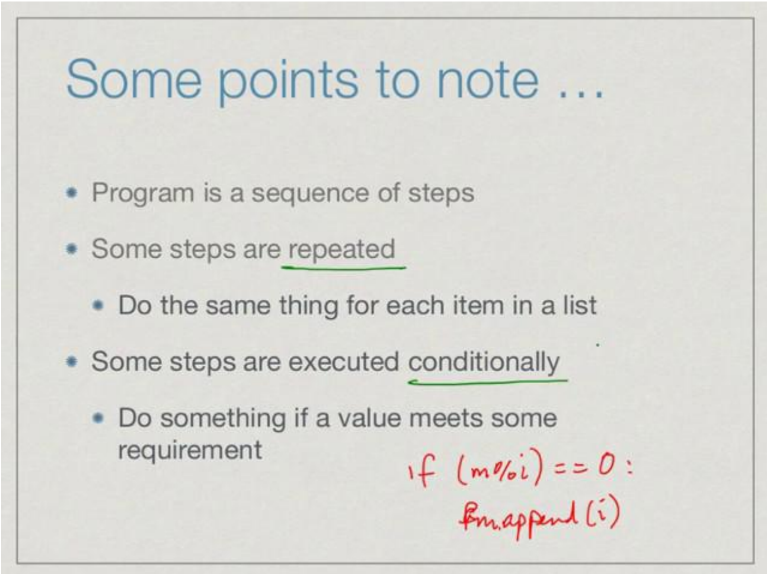
(Refer slide Time: 24:37)




And having assigned a value we can then modify the value. For instance every time we find a new factor of n we do not want to through any old factor we want to take the existing list fm and we want to add it. So, this function append for instance modifies the value of the name fn to a new name which takes the old name and sticks an i at the end of it.

More generally you could have a number and we could want a replaces by two times a number. So, we might have something like i is equal to two times i. So, star stands for multiplication this does not mean that i is equals to two times i arithmetically because; obviously, unless i is 0 i cannot be equal to two times itself. What is means is that take the current value of i, multiply it by two and assign it to i. So, we will see this as we go along, but assignment can either assign a completely new value or you could update the value using the old value. So, here we taking the old value of the function of the list fn and we are appending a value it would getting a new value of fn.

(Refer slide Time: 25:49)




The other part that we are need to note is how we execute steps. So, we said at the beginning of today's lecture a program is a sequence of steps. But we do not just execute the sequence of steps from beginning to end blindly. Sometimes we have to do the same thing again and again. For instance we have to check for every possible factor from 1 to m if it divides m and then put it in the list. So, some steps are repeated we do something, for examples here for each item in a list.

And some steps are executed only if the value that we are looking at meets particular conditions. When we say something like if m percent i is 0, if the remainder of m divided by a is 0 then append. So, the step append i to fm the factors of m this happens only if i matches the condition that it is a factor of m. So, we have repeated steps where same thing done again and again. And they have conditionals steps something which is done only if a particular condition holds.

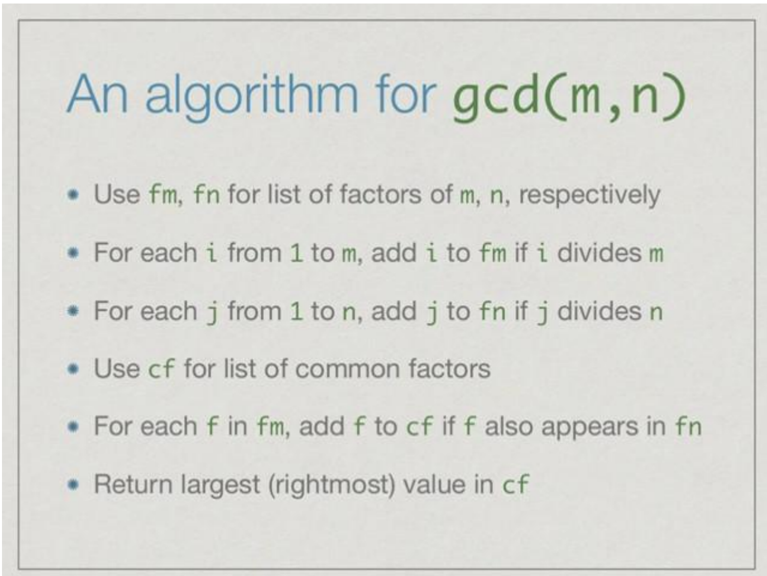
So, we will stop here. These examples should show you that programs are not very different from what we know intuitively, it is only a question of writing them down correctly, and making sure that we keep track of all the intermediate values and steps that we need as we long, so that we do not lose things. We will look at this example in more detail as we go long, and try to find other ways of writing it, and examine other features, but essentially this is a good way of illustrating programming.

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

**Week - 01**
**Lecture - 02**
**Improving naive gcd**

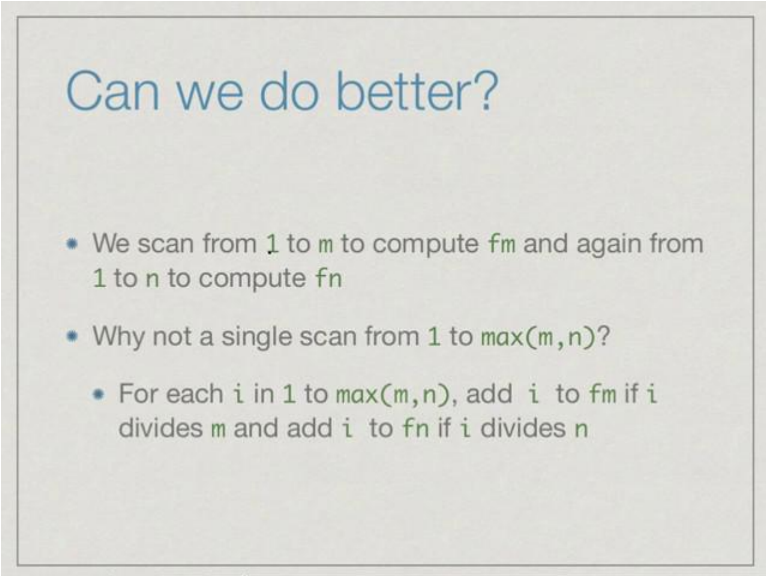
In the first lecture we used gcd as an example to introduce some basic concepts in programming. We will continue to look at the same example and see how to refine our program and explore new ideas.

(Refer Slide Time: 00:14)




Here was our basic algorithm for gcd, which as we said more or less follows the definition of the function. We construct two lists of factors for the inputs m and n. So, we construct fm the factors of m, fn the factors of n, and then from these we compute cf the list of factors in both lists or common factors. Our goal is to return the greatest common divisor of the largest number in this common list which happens to be the last one in this list, since we add these factors in ascending order.

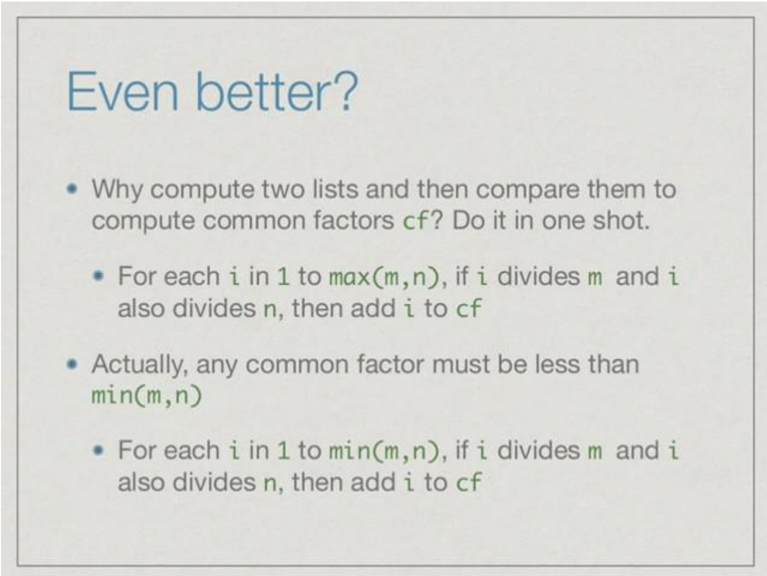
(Refer Slide Time: 00:51)




So can we do better than this? The way we have proceeded, we first scan all numbers from 1 to m to compute the list fm of factors of m, and then we again start from 1 to n to compute fn. So, an obvious improvement is to just directly scan the numbers from 1 to the larger of m and n and in one scan compute list fm and fn.

In another words for each i in this list 1 to the maximum of m and n we first check if i divides m, if so we add it to the list of factors of m, and then we check if i divides n and if so we add it to list fn. Instead of doing two separate scans over 1 to m and then 1 to n and repeating the past we do it in one scan.

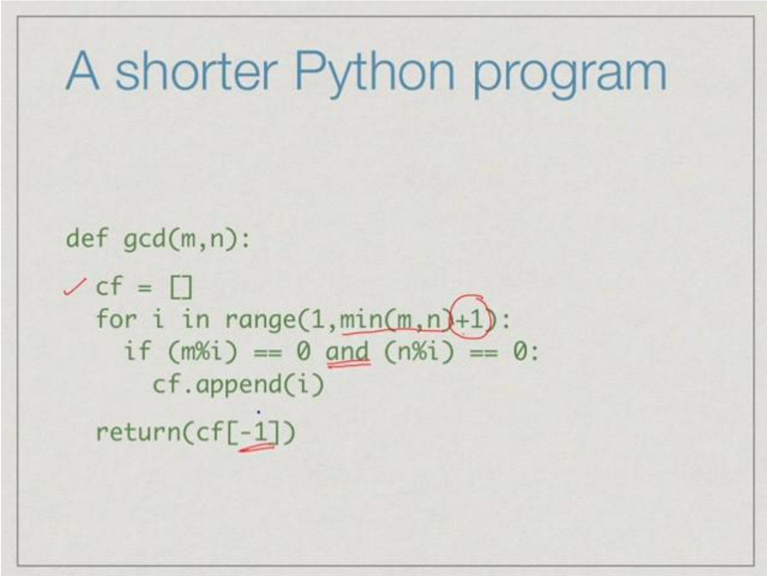
(Refer Slide Time: 01:46)




But even this can be improved upon. If we are doing it in one pass and we are checking if numbers divide both - m and n, then why not we just directly check for common factors. In another words instead of computing two lists and then combining them we can just directly do the following: for each i from 1 to the maximum of m and n, if i divides both m and n then we directly add i to the list of common factors. If it divides neither or if it divides only one of them then it is not a common factor and we can discard.

In fact, notice that rather than going to the maximum of m and n we should go to the minimum of m and n, because once we cross the smaller number we will not get a factor for the smaller number. Remember that the factors of m lie between 1 and m and for n lie between 1 and n. If m is smaller than n for example, if we input m plus 1 though it made a factor of n it certainly cannot be a factor of m. So our better strategy is for each i in the range 1 to the minimum of m, and n if i divides both m and n then we add i to the list of common factors.

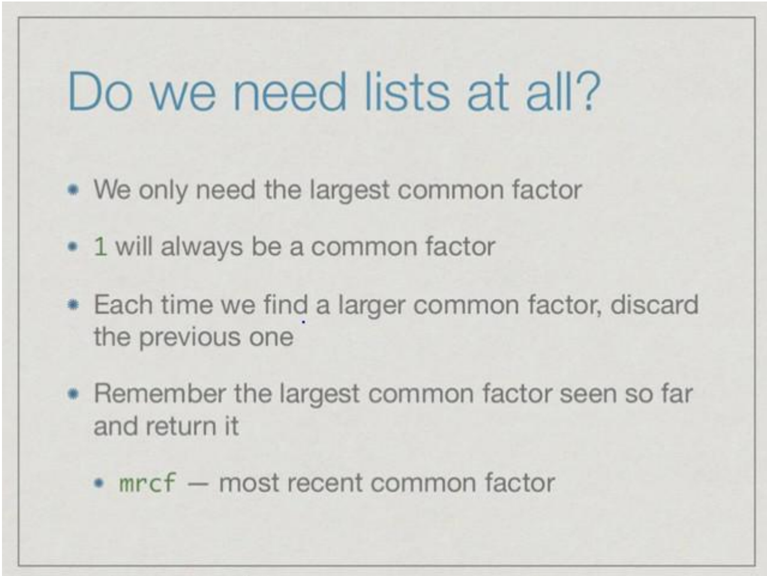
(Refer Slide Time: 03:03)




Here is a much shorter Python program implementing this new strategy. So instead of computing the lists fm and fn we directly compute the list of common factors. We let i range from 1 to the minimum of m plus m and n and remember that Python requires us to give the limit of the range is one more than the limit we want to go up to, so we go from 1 to the minimum m n plus 1. And now we have an extra connective it is called a logical connective and which says that we want two conditions to be proved, we want the remainder when m is divided by i to be 0, in another words i divides m and we also want the remainder when n is divided by i to be 0, so i should divide both m and n and if so we add i to the list of common factors.

And having done so once again we are doing it in ascending order, so the common factors are being added as we go along the larger ones come later. So, we finally want the last element which in Python is given as the minus 1th element of the list cf.

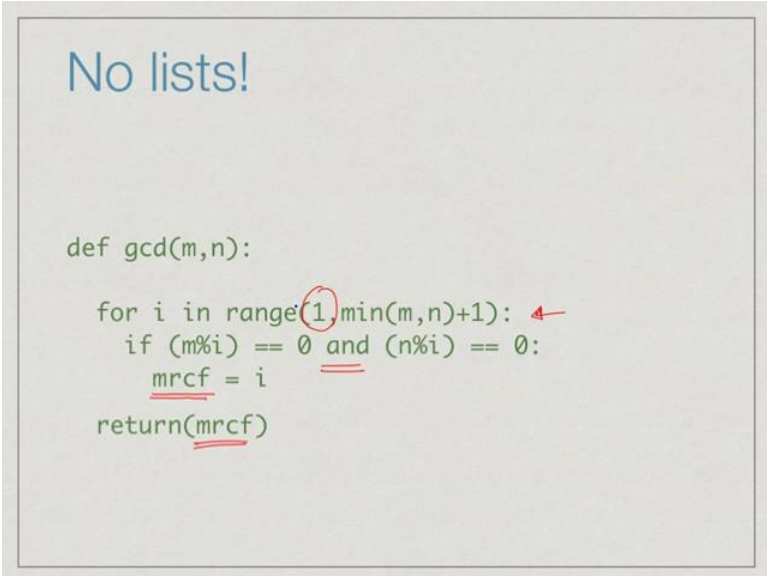
(Refer Slide Time: 04:15)




So having done this, maybe we can simplify things further. Do we need a list of common factors at all? Remember that we only need the largest common factor. We observed that there will always be at least one common factor namely 1. So, the notion of the largest common factor will always be well defined for any pair m and n.

Each time we can start with 1 and each time we find a larger common factor we can discard the previous one, we do not need to remember all the common factors we only need the largest one. So this can be greatly simplifying our strategy because we do not need to keep the list of every possible common factor in this list; we just need to keep the largest one that we have seen. We can use a name say mrcf for the most recent common factor, and keep updating this name with the value of the common factor that we saw last.

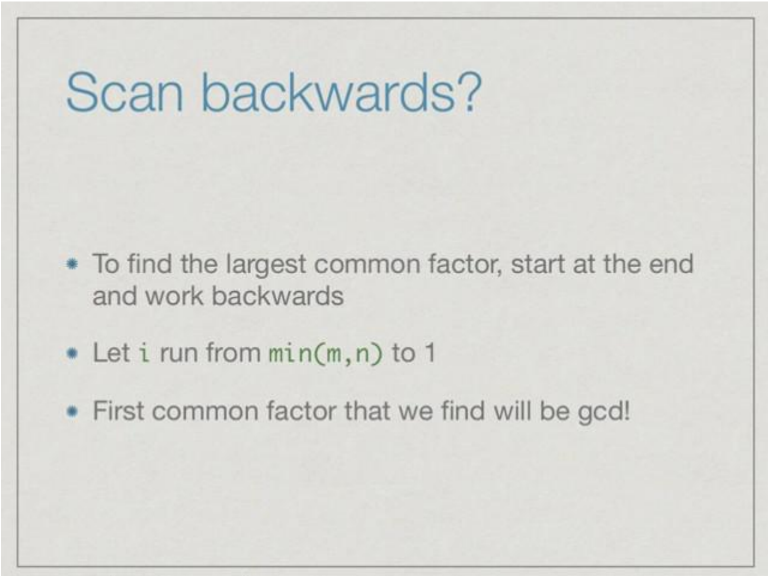
(Refer Slide Time: 05:17)




So here is a Python implementation of this idea where we do not have a list at all. We directly scan all the possible common factors from 1 to the minimum of m and n. Whenever we find a common factor we update the value of our name mrcf to be the current common factor that we have found. Now remember that 1 will always be a common factor, so initially mrcf will be assigned the value 1, it will not be that we go through this repeated execution of this and never assign mrcf because the condition is never true.

Since one is the common factor we will at least have mrcf equal to 1, but if we find a larger common factor the one will be replaced by the later common factor. At the end of this for iteration or loop what we end up with is the largest common factor that we found between 1 and the minimum of m and n which is indeed the gcd of m and n.

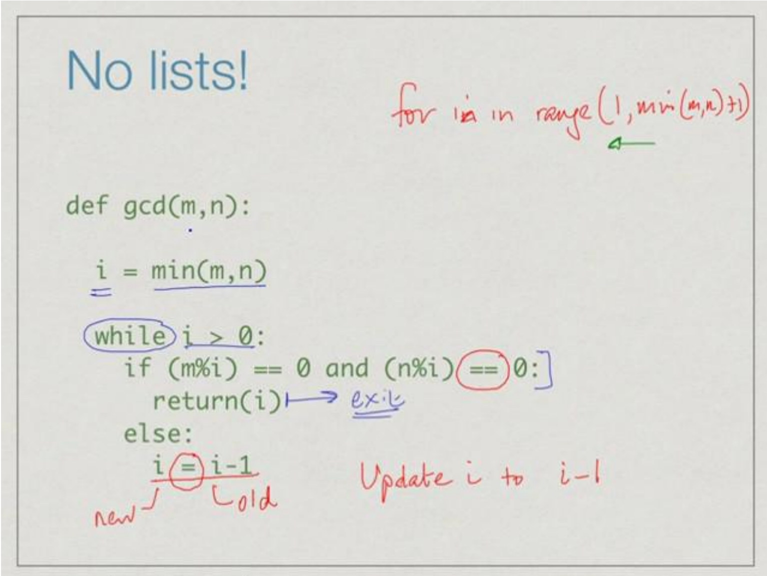
(Refer Slide Time: 06:18)




We can still do some further optimizations. Since, we are looking for the largest common factor, why do we start at the beginning which will give us the smallest common factor. So, we can start at the end of the list and work backwards. Instead of running from 1 to the minimum of m and n we can start from the minimum of m and n and work backwards to 1. Again the guarantee is that the 1 will always show up as a common factor, so if there are no other common factors at the very end we will find 1 as the greatest common factor.

And in this process we do not have to ever go past any common factor that we find, if we are working backwards from largest to smallest the very first common factor we find must be in fact the gcd of m and n.

(Refer Slide Time: 07:09)




How would we write this in Python? Well, you can modify that for i in range, so notice that normally this function goes from a smaller value to a bigger value, you can modify this to go backwards instead. But instead of doing this which we will see how to do later on when we actually get into formal Python, let us explore a new way of going through a list of values.

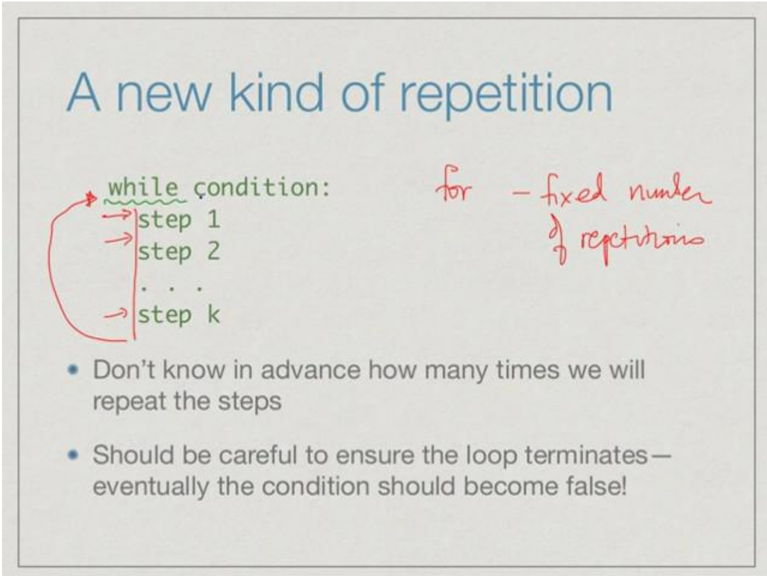
We start by assigning to the index i, the value that we want to start with namely the minimum of m and n, remember we want to start at the largest possible value for i and go backwards. So what we have is, a new word instead of for called while, so while as the English word suggests is something that happens while the condition is true. So, while the i that we are looking for is positive. So while i is greater than 0, what do we do? We check if i is a common factor. This is the same as before, we check whether i divides m and i also divides n.

If we find the common factor we are done, we just return the value of i that we have found and we implicitly exit from this function. Every time you see a return statement in a function, the function terminates and the value in the return is what the function gives back to us. So we start with i equal to the minimum of m and n and we check whether i is a common factor if it is so we exit and return the value of i that we last found. And this is the only value that we need, we do not need any other common factors. So, we return the very first time we see a common factor.

On the other hand, if i is not a common factor we need to proceed by checking the next one which is to go backwards and this is the achieved by this update. So, remember that we said that we can assign values or update values using this equality operation. This equality operation is not mathematical equality as it looks, but rather it is the assignment of a value. It says, take the old value of i and make it the new value. So it says, update i to i minus 1, take the current value of I, subtract 1 and replace it in i. The mathematical equality is written as double equal too this is what we use in our conditions.

So, it is important to remember this that double equal to means equality as in the left hand side is equal to the right hand side, whereas the single equality in Python and many other programming languages means assign a value to a variable. So, this is the final optimization that we have of this naive algorithm which is to basically scan for common factors from the beginning to the end. So now we are doing it from the end to the beginning and keeping only the first factor that we find.

(Refer Slide Time: 10:03)

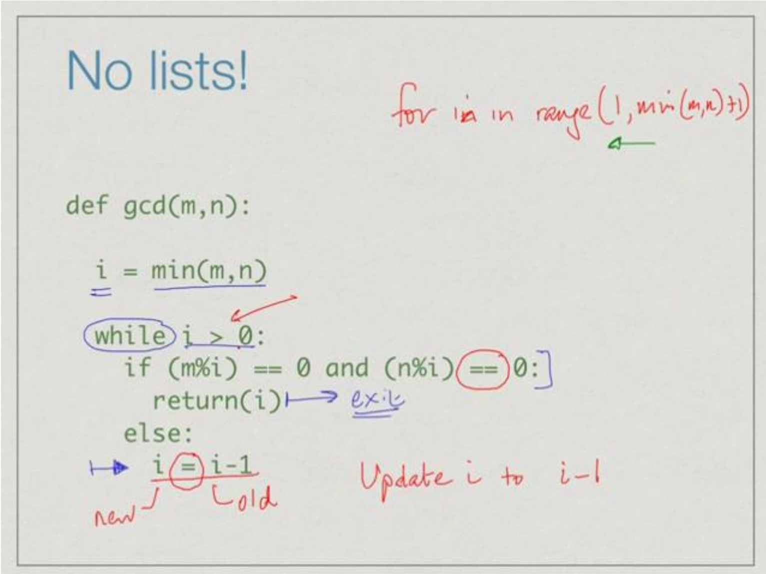



What we saw in this example is a new kind of loop. So, this new kind of loop has a special word while. And while is accompanied by condition, so long as the condition is true, we do whatever is within the body of the while. Now notice that Python uses indentation, so these statements here are offset with respect to the while. This is how Python knows that steps 1 to k belong to this while. So, these are the steps that must be repeated at the end of this thing, you come back and you check whether the condition is still true, if it is true you do it one more time and so on. So, while is useful when we do not know in advance how many times we will repeat the steps.

When we were scanning for the list of factors we knew that we would start with one and go up to the minimum of m and n. We could predict in advance that we would do precisely that many steps and so we could use this for loop, so for loop has a fixed number of repetitions. On the other hand, a while loop is typically used when you do not know in advance when you are going to stop. So in this case we going to start with the minimum of m and n, work backwards and stop as soon as we find the factor, but we have no idea in advance whether this will come early or will have to go all the way back to 1 which we know is guaranteed to be a valid factor.

One of the problems that one could face with the while is that we keep coming back and finding that the condition is true. So we never progressed out of the while. So, so long as the condition is true these steps will be executed and then you go back and do it again. If you have not changed something which makes the condition false you will never come out.

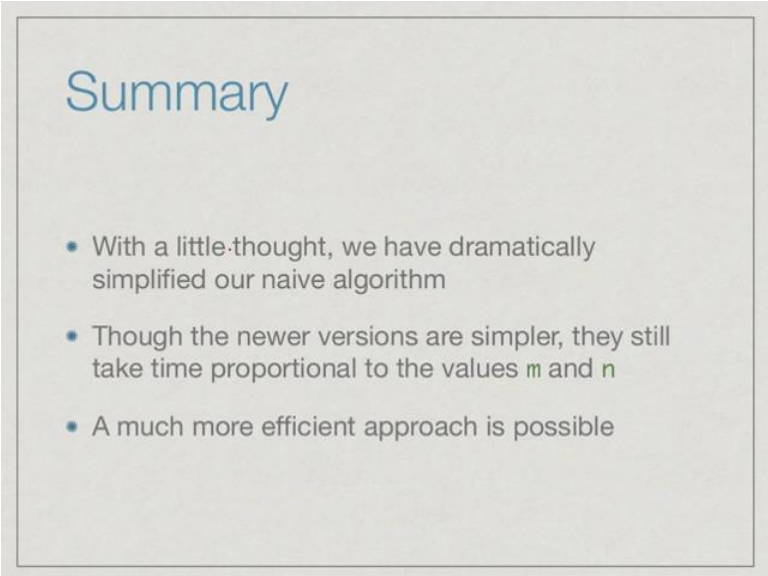
(Refer Slide Time: 11:51)




In our previous example, in order to make the condition false we need to i to become 0. So we start with the minimum of m and n. So, what we guarantee is that every time we go through this while and we do not finish what we wanted to do we reduce i by 1, and so since we start with some fixed value and we keep reducing i by 1 eventually we must reach 0.

So in general when you use a while loop you must make sure that you are making progress towards terminating the loop otherwise you have a dangerous kind of behavior called an infinite loop where the computation just keeps going on and on without returning a value and you have no idea whether it is just taking a very long time to compute the answer or whether it is never going to finish.

(Refer Slide Time: 12:35)




So in this lecture what we have seen is that we can start with a very naive idea which more or less implements the function as it is defined and work our ways to dramatically simplify the algorithm. Now one thing from a computational point of view, is that though the newer versions are simpler to program and therefore to understand, the amount of time they take is not very different. We are still basically running through all values in principle from 1 to the minimum of m and n. If we start from the beginning then we will run through all these values anyway because we scan all these numbers in order to find the common factors.

In the last version where we were trying to work backwards and stop at the first common factor it could still be that the two numbers have no common factor other than 1. So again, we have to run all the way back from minimum of m and n back to 1 before we find the answer. Although, the programs look simpler computationally, they are all roughly the same and that they take time proportional to the values m and n.

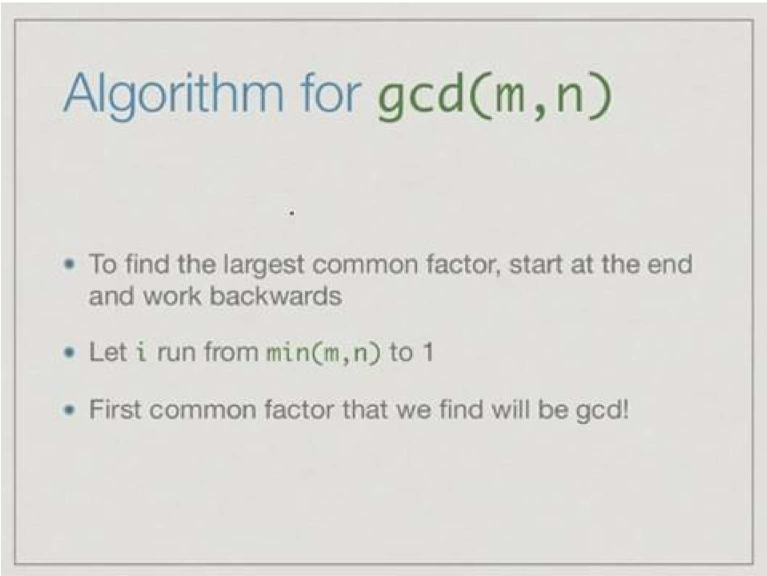
What we will see in the next lecture is that we can actually come up with a dramatically different way to compute gcd, which will be much more efficient.

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

**Week - 01**
**Lecture - 03**
# Euclid's Algorithm for gcd

Let us continue with our running example of gcd to explore more issues involved with program.

(Refer Slide Time: 00:09)

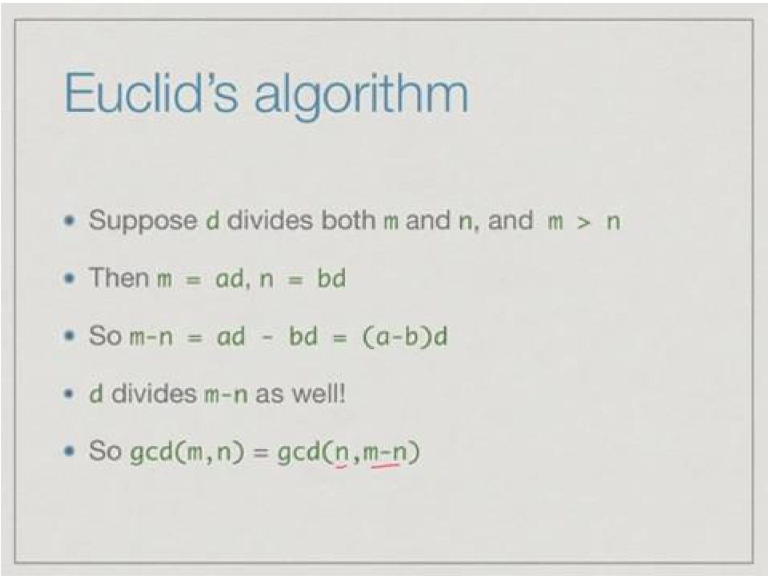



We started with the basic definition of gcd, which said that we should first compute all the factors of m, store it in a list, compute all the factors of n, store it in another list, from these two lists, extract the list of common factors and report the largest one in this common factor list. Our first simplification was to observe that we can actually do a single pass from 1 to the minimum of m and n and directly compute the list of common factors without first separately computing the factors on m and the factors of n. We then observe that we don't even need this list of common factors since our interest is only in the greatest common factor or the greatest common divisor. So, we may as well just keep track of the largest common factor we have seen so far in a single name and report it at the end.

Our final simplification was to observe that if we are interested in the largest common factor, we should start at the end and not the beginning. So, instead of starting from 1 and working upwards to the minimum of m and n its better to start with minimum of m and n and work backwards to one, and as soon as we find a common factor we report it and exit.

Remember always that 1 is guaranteed to be a common factor. So when we start from minimum of m and n and go backwards, if we don't see any other common factor, we are still guaranteed that we will exit correctly when we hit one. So what we notice that was, that though these different versions are simpler than the earlier versions they all have the same efficiency in terms of computation, which is that they will force us in the worst case to run through all the numbers between 1 and the minimum of m and n, before we find the greatest common factor whether we work forwards or backwards.

(Refer Slide Time: 02:08)

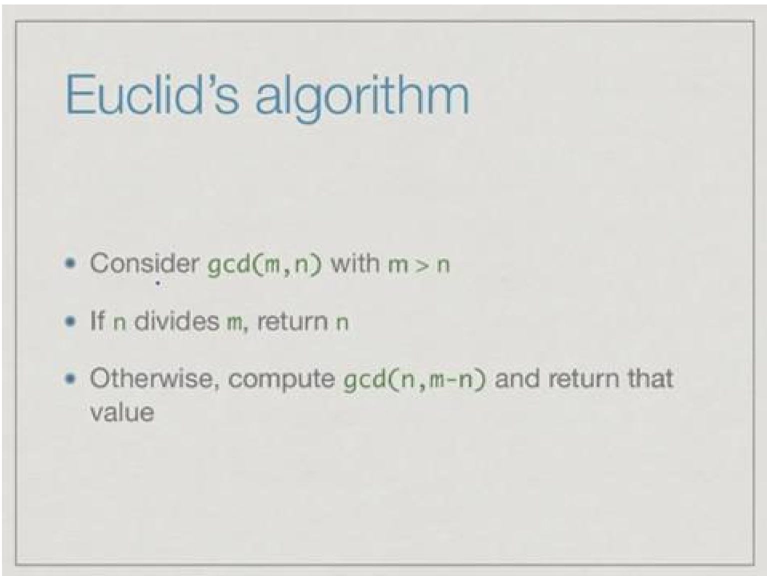



So at the time of the ancients Greeks, what was possibly the first algorithm in modem terminology was discovered by Euclid, and that was for this problem - gcd. So what Euclid said was the following. Suppose we have a divisor d which divides both m and n, so this is a common divisor and we are looking for the largest such d. Let us assume also for the sake of argument that m is greater than n. So if d divides both m and n, we can write m as a times d and n as b times d for some values a and b, so m is multiple of d and so is n.

So if we subtract the equations then the left hand side is m minus n. So, we take m and subtract n from m, so correspondingly we subtract b d from a d. So, m minus n is equal to a d minus b d, but since d is a common term this means m minus n is a minus b times d. This is where we are using the assumption that m is greater than n, so a minus b will be a positive number. But the important thing to note is that m minus n is also a multiple of d. In other words, if d divides both m and n, it also divides m minus n. And since d is the largest divisor of m and n, it will turn out that d is also the largest divisor which is common to m, n and m minus n.

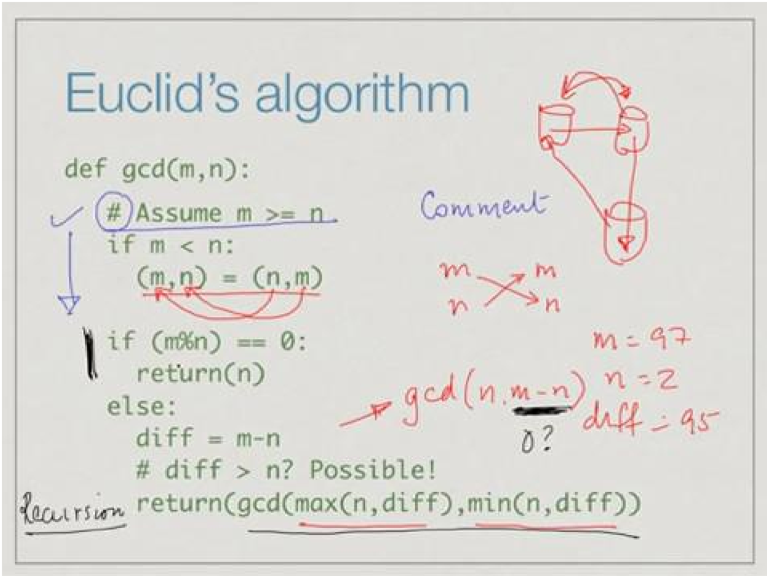
In other words, the gcd of m and n is the same as the gcd of the smaller of the two, namely n and the difference of the two m and n, m minus n. So, we can use this to drastically simplify the process of finding the gcd.

(Refer Slide Time: 04:00)




So here is the first version of Euclid's algorithm. So, consider the value: gcd of m n assuming that m is greater than n. So if n is already a divisor of m, then we are done and we return n. Otherwise, we transform the problem into a new one and instead of computing the gcd of m and n that we started with, we compute the gcd of n and m minus n and return that value instead.

(Refer Slide Time: 04:32)




So, here is a python implementation of this idea. There are a couple of new features that are introduced here, so let us look at them. The first is this special statement which starts with symbol hash. So in python, this kind of a statement is called a comment.

So a comment is a statement that you put into a program to explain what is going on to a person reading the program, but it is ignored by the computer executing the program. So, this statement says that we are assuming that m is bigger than or equal to n. So, this tells us that when the program continues this is the assumption. Of course, it is possible that the person who invokes gcd does not realize this, so they might invoke it with m smaller than n and so we fix it.

This is a special kind of assignment which is peculiar to python; it is not present in most other programming languages. So what we want to do is, basically we want to take the values m and n and we want to exchange them, right. We want to make the new value of m, the old value of n and the new value of n, the old value of m, so that in case m and n were in the wrong order we reverse them. So, what this python statement does is it takes a pair of values and it does a simultaneous assignment so it says that the value of n goes into the value of m and the value of m goes into the value of n.

Now it is important that it is simultaneous, because if you do it in either order, if you first copy the value of n into m, then the old value of n is lost. So, you cannot copy the old value of m into the new value of n because you have lost it. So imagine that you have two mugs of water, and now you want to exchange their contents. Now you have to make space, you cannot pour this into that without getting rid of that and once you got rid of that you cannot pour that into that, so you need third mug normally.

You need to first transfer this here and keep it safe, and then you need to transfer this there and then you need to copy it back. So this is the normal way that most programming languages would ask you to exchange two values, but python has this nifty feature by which you can take a pair of values and simultaneously update them and in particularly this simultaneous update allows us to exchange the values without worrying about having this extra temporary place to park one value.

Anyway, all that this first part is doing is to ensure that this condition that we have assumed is satisfied. So now we come to the crux of the algorithm. If m divides n that is remainder of m divided by n is 0 then we have found n to be the gcd and we return n. If this is not the case, then we go back to what we discovered in the last slide and we now are going to compute gcd of n and the difference m minus n. We would ideally like to compute gcd of n and m minus n. So, we compute the difference m minus n and we could just invoke this.

But, it is possible, for example - if m is say 97 and n is 2 then the difference will be 95. The difference could very well be larger than n, and we would ideally like to call this function with the first number bigger than the larger number. So we will just ensure this even though our function does take care of this. What we want to do is, we want to call gcd with n and m minus n instead we will call gcd with the maximum value of n and the difference as a first argument and the minimum value of n and the difference. So it will make sure that the bigger of the two values goes first and the smaller of the two values go. And whatever this gcd, the new gcd returns is what this function will return.

This is an example of what we will see later, which is quite natural, which is called Recursion. Recursion means, that we are going to solve this problem by solving the smaller problem and using that answer in this case directly to report the answer for our current problem. So we want to solve the gcd of m and n, but the gcd of m and n instead we solve the gcd n and m minus n and whatever answer that gives us we directly report it back as the gcd for this, so we just invoke the function with the smaller values and then we return it.

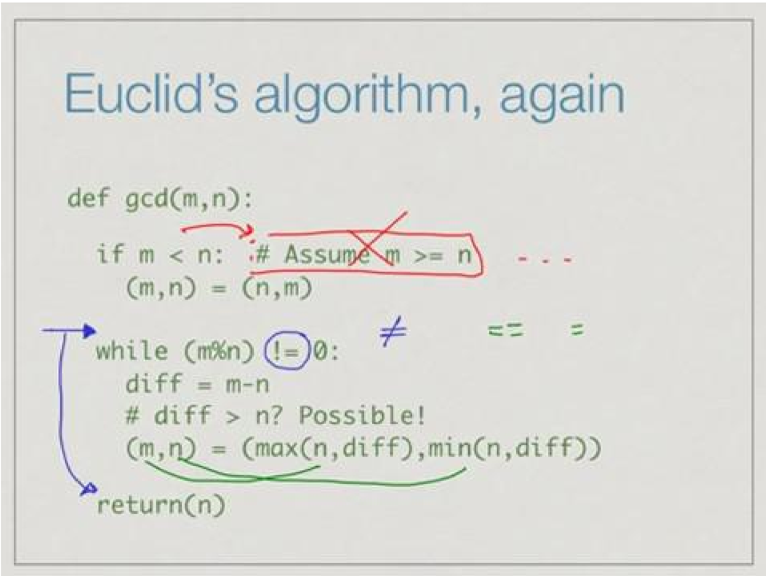
Now whenever you do a recursive call like this, it is like a while loop; it will invoke the function again, that in turn will invoke a smaller function and so on. And you have to make sure that this sequence in which gcd keeps calling gcd with different values does not get to an infinite progression without a stopping point. So, formally what we have to ensure is that this guarantee of finding an n which divides m, so this is where gcd actually exits without calling itself. We have to make sure that eventually we will reach this point. Now what is happening if you see here is that the values that are passed to gcd are getting smaller and smaller.

Now what can we have for m minus n? What can be the value? Can it be 0? Well, if m minus n is 0 that means m is equal to n, if m is equal to n then certainly m is divisible by n. If m minus n is 0 then it could have exited, so it cannot be 0. It must be at least 1, so whenever we call this function m minus n it's at least one. On the other hand each time we are reaching smaller values. So, we start with some value and m minus n keeps decreasing.

What happens when it actually reaches 1? Well, when it reaches 1 then 1 divides every other number, so m percent n or m divided by n, the remainder will be 0, so we will return gcd of 0. In other words, we had guaranteed that this function because it keeps reducing the number that we invoke the function with will eventually produce a call where gcd terminates. This is important and we will come back to this later but whenever you write a function like this, you have to make sure that there is a base case which will be reached in a finite number of steps no matter where you start.

This is Euclid's algorithm, the first version where we observe that the gcd of m and n can be replaced by the gcd n and m minus n. And what we have seen in this particular implementation are three things rather, we have seen how to put a comment in our code, we have seen that python allows this kind of simultaneous updation of two variables at the same time so m comma n equal to n comma m. We have also seen that we can use the same function with new arguments in order to compute the current functions. So there is no problem with saying that in order to compute gcd of m and n, I well instead compute gcd's on some other value and use that answer to return my answer.

(Refer Slide Time: 11:53)




Let us look at a different version of this algorithm, where we replace the recursive call by a while loop. We saw while in our last version of this standard algorithm when we were counting down from minimum of m comma n to 1, so we kept checking whether i was greater than 0 and we kept decrementing. Well, here we are doing the recursion using a while, so the first thing to notice here is that I have moved this comment which used to be in a separate line to the end of the line.

What python says is that, if there is hash then the rest of the line can be ignored. So, it reads this line it sees a valid condition and then sees the hash, so it's as though this statement was not part of the python program when it is executed. Comment can either be in a separate line or it can be in end of a line. Of course, remember that you cannot put anything after this which you want python to execute because once it sees a hash the rest of the line is going to be ignored, so it cannot be in middle of a line you cannot put a comment in middle of a line, but you can put it on separate line or you can put it at end of the line.

So anyway so this is our comment as before. So up to here there is no change except that I have shifted the comment position. Now we reach this point where we actually have to do some computation. At this point if we have found n such that n divides m we are done and we can directly return n. So, this is what our recursive code would do. If we have not found such an n we have to do some work. The condition is to check whether m divided by n actually produces a remainder. So, this not equal to symbol is return with this exclamation mark, so this is the same as the mathematical not equal to.

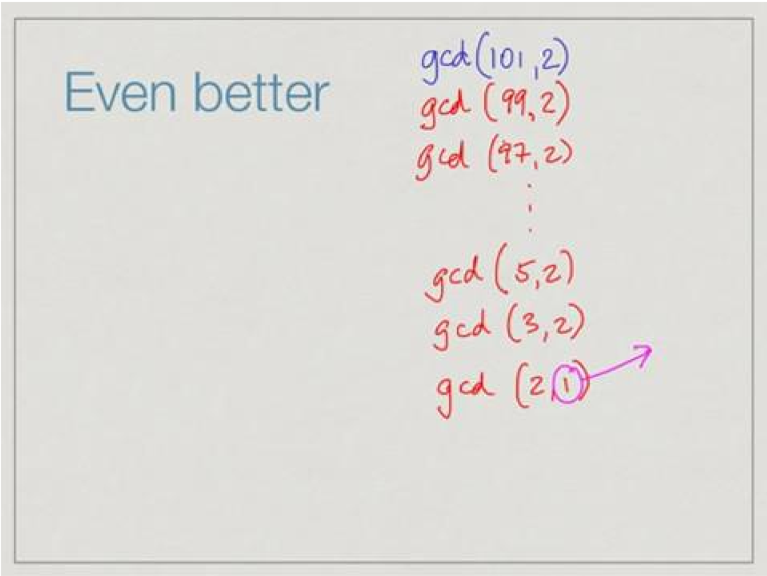
Remember that this double equal to was what we use for the mathematical symbol of equality. This is our symbol for not equal to. So, so long as there is remainder, that is the remainder m divided by n is not 0 we do what we did before we compute the difference and we replace m by the maximum of the two values and n by the smaller of the two values. We have a pair m n whose gcd we are trying to find right, with assumption that m is bigger than n at each step we replace m by the larger of n and the difference and n by the smaller of n and the difference.

This exactly what we are doing in the recursive call, we are saying pretend we are computing gcd of that. Here in this while loop effectively we are saying replace the gcd of m n by the computation of maximum n diff and minimum n diff. We keep doing this until we hit a condition where n actually divides m, and exactly like we said in the recursive case that there will be a boundary case where at worst case n will become 1 and 1 will divide everything.

In the same way here the difference will keep reducing, the difference cannot be 0, because if difference is 0 it could have divided, so difference can at most go down to 1 and when it hits one we are done. This a while version of the same recursive function we wrote earlier, so if it helps you can look at these side by side and try to understand what this recursive things is doing and what the while is doing and see that they are basically doing the same thing.

And the idea that the recursion must terminate is exactly analogous to the claim that we said earlier that when you write a while you must make sure that you make progress towards making the while condition false, so that the while exits. So, just like the recursion can go on forever, if you are not careful and you do not invoke it with arguments which guarantee termination, the while can also go on forever if you do not make progress within the while in order to make sure that the while condition eventually becomes false.

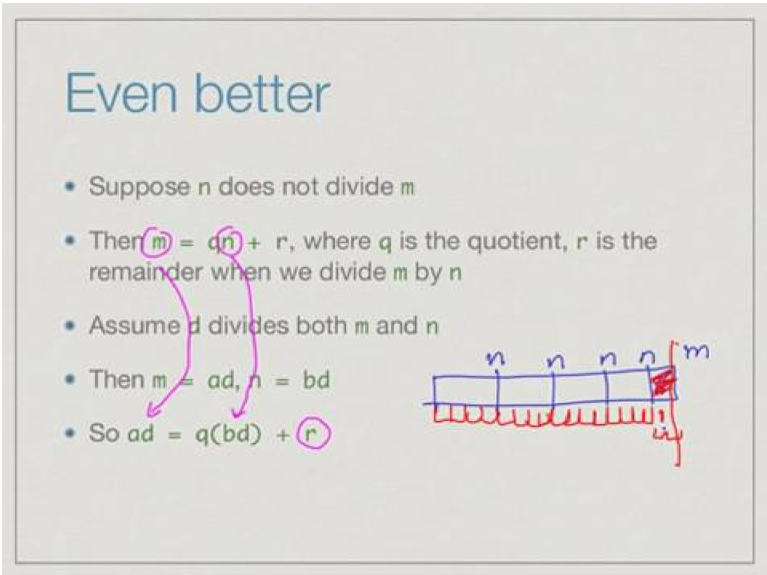
(Refer Slide Time: 15:42)




We can actually do a little better than this. Let us see one problem with this by doing a hand execution. So supposing we start with some number like gcd of 101 and 2, then our algorithm will say that this should now become gcd of the difference and n, the difference is 99 so will have 99 and 2, and then this will become gcd of 97 and 2 and so on. So, we will keep doing this about 50 steps then eventually we will come down to gcd of 5 and 2, and then gcd of 3 and 2. Now when we compute the difference we get gcd of 2 and 1, so now the difference will become smaller. Then at this point we will report that the answer is 1. So, it actually takes us about 50 steps in order to do gcd of 101 into 2.

One of our criticisms of naive approach is that it takes time proportional to the numbers themselves. If you had numbers m and n we would take in general number of steps equal to minimum of m and n. Now here, in fact we are taking steps larger than the minimum because the minimum is 2, if you were just computing factors we will see that the only factor of 2 is 2 and it is not a factor 101 we would have stopped right at beginning. This actually seems to be worst then our earlier algorithm in certain cases.

(Refer Slide Time: 17:05)

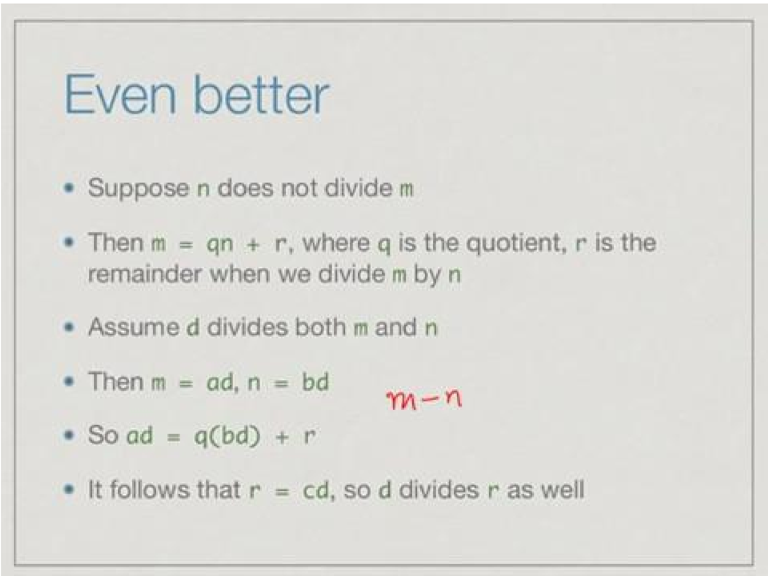



Here is a better observation suppose n does not divide m. In other words if I divide m by n i will get a quotient and a remainder. So, I can write n as q times n plus r where q is quotient and r is the remainder, so you may remember these terms from high school arithmetic. N goes into n q times and leaves a remainder r and we are guaranteed that r is smaller than n, otherwise r it could go one more time it will become q plus 1. We have the remainder r which is smaller than n. So for example if i say 7 and i want to divide it by 3 for example, this will be 2 times 3 plus 1, so this will be my quotient and this will be my remainder. And the important thing is remainder is always smaller than what I am dividing by.

Now, let us assume as before that we have a common divisor for both m and n. In other words like before we can write m itself as a times d and n as b times d for some numbers a and b, because m is is multiple of d and so is n. If you plug this into the equation above here, then we see that m which is a times d is equal to q times n which is b times d plus r. So, d divides the left and d divides one part of the right. You can easily convince yourself that d must also divide r.

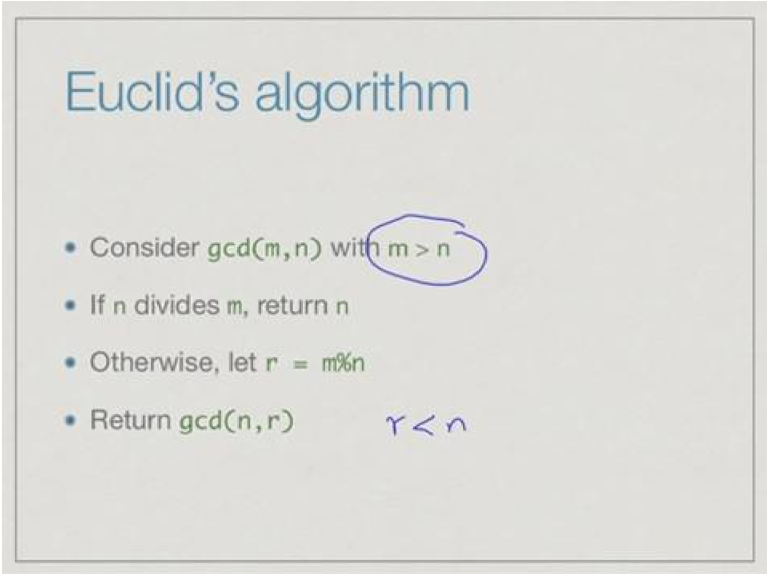
The way to think about it if you want to pictorially is that I have this number m and I can break it up into units of n and then there is a small bit here. On the other hand if I look at d, d evenly divides everything. So it divides each of these blocks it also divides the whole thing. If I continue with d, it is going to stop exactly at this boundary because d also divides n, therefore d must also divide this last bit which is r exactly. In other words, we can argue very easily that r must also be a multiple of d. So d must divide r as well.

(Refer Slide Time: 19:21)




If d divides m and b divides n then d must divide the remainder of m divided by n. And as we saw before with the difference, the last time we said we would look at the difference m divided by n. Now we are saying we look at the remainder of m divided by n and d must divide that and d will be in fact the gcd of n and this remainder.

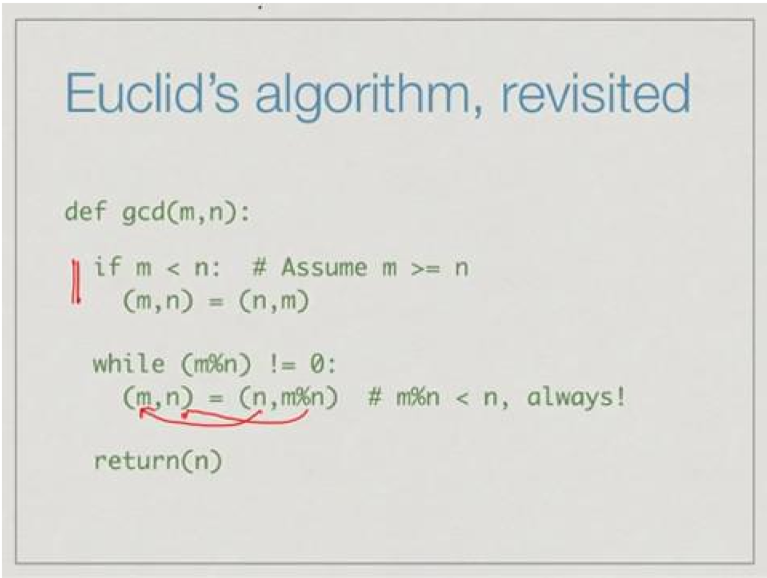
(Refer Slide Time: 19:44)




This is an improved and this is the actual version of the algorithm that Euclid proposed, not the difference one but the remainder one. It says consider the gcd of m and n assuming that m is bigger them n. Now if n divides m we are done we just return n, this is the same as before.

Otherwise, let r be the remainder with the value of m divided by n get the remainder and return the gcd of n and r, and at this point one important thing to remember is that r is definitely less than n. So we do not have to worry about this condition here, we do not have to take the max and the min as we did for the difference because the remainder is guaranteed to be less than n otherwise n would go one more time.

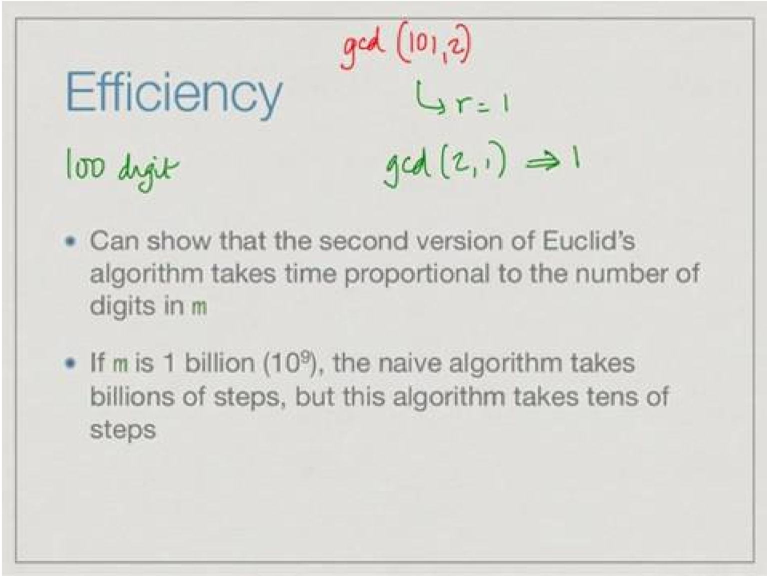
(Refer Slide Time: 20:31)




As before we have very simple recursive implementation of this, and this is even simpler because we do not have to do this max min business. So, like the previous time we first flip m and n in case they are not in the right order. Then if n divides m if the remainder of m divided by n is 0 we return n and we are done, otherwise we return the gcd of n and the remainder, so this is the remainder. And remember that the remainder is always less than n so we do not have to worry about flipping it and taking max and min at this point. And analogous tp the previous case we can do this whole thing using a while instead of doing it with recursive thing.

We first exchange m and n if they are in the wrong order, then so long as the remainder is not 0 we replace m by the smaller of the two numbers and we replace n by the remainder and we proceed. Now we are guaranteed that this remainder will either go to 0, but if it goes to 0 it means it divides or if it's not 0 in the worst case the remainder keeps decreasing because it is always smaller than the number that we started with. So it keeps decreasing and it reaches 1 then in the next step it will divide. So finally, we will return at least one.

(Refer Slide Time: 21:48)




If we go back to the example that we were looking at, so if we saw that gcd 101, 2, and we did it using the difference we said we took about 50 steps. Now here if we do the remainder I am going to directly find that r is equal to 1 right if I divide 101 by 2 it goes 50 times remainder 1. In one step I will go to gcd 2 comma 1 and I will get 1.

In fact, what you can show is that this version with the remainder actually takes time proportional to number of digits, so if I have say hundred digit number it will take about a hundred steps. So for instance if we have a billion as our number, so billion will have about 10 to the 9 will have about ten digits. Then if I do the naive algorithm then it could take some constant times of billion numbers of steps say a billion steps. But this algorithm because of the claim it takes time proportional to number of digits since 10 to the 9 has approximately 10 digits it will only take about 10 steps, so there is a dramatic improvement in efficiency in this version.

This is something that we will touch up on while we are doing this course. This course is about programming, data structures and algorithms. So the programming part talks about what is the best way to express a given idea in a program in a way that it is easy to make sure that it is correct and easy to read and maintain, so that is the programming part. How do you write, how do you express your ideas in the most clear fashion. But the idea itself has to be clear and that is where data structures and algorithm start. So you might write beautiful prose, but you may have no ideas or you may have very brilliant ideas but you may express yourselves clumsily, neither of them is optimal.

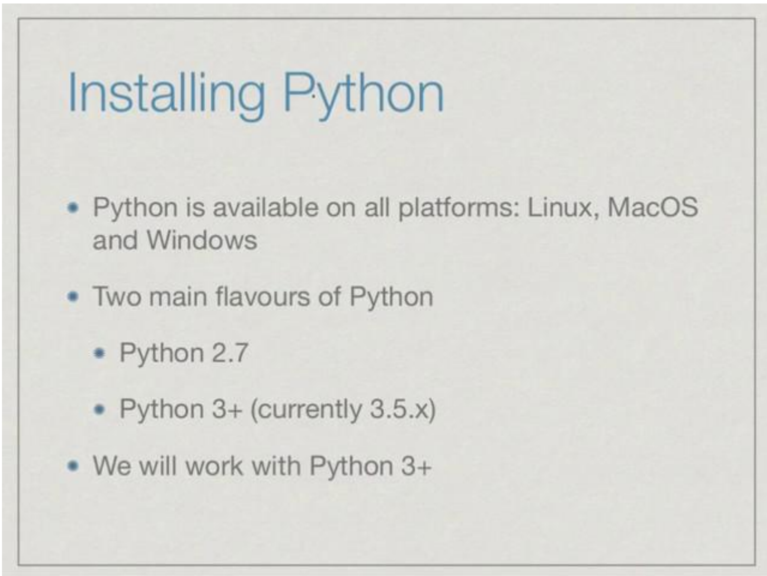
This is like writing in any other language. You may have brilliant ideas to express, but if you cannot convey them to the person you are talking to the ideas lose their impact. So, you need ideas and you need a language to express them. Programming is about expressing these ideas, but the ideas themselves come from algorithms and data structures.

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

**Week - 01**
**Lecture - 04**
**Downloading and installing Python**

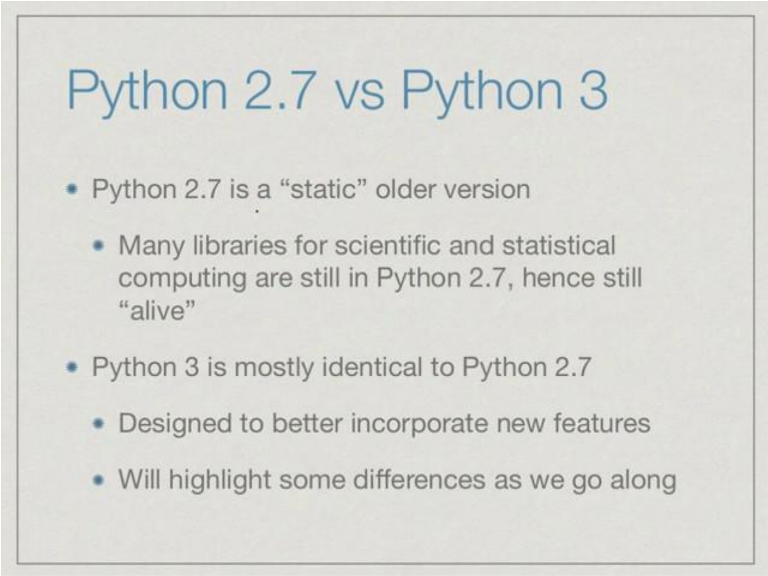
For our final lecture of this first week, we will see how to actually use Python on our system.

(Refer Slide Time: 00:10)




Python is a programming language, which is available on all platforms. So, whether you are working on Linux or on a Mac or on Windows, you will be able to find a version of python that works on your system. One of the small complications with python is that there are two flavors or two versions of python, which are commonly found. So, there is an older version called python 2.7, and there is a newer version called python 3. Python 3 is a one that is being actively developed, python 2.7 is more or less a static version and currently python 3 has the version 3.5.2 or something like that. So, there is not much difference whether you are using 3.5 or 3.4, but there are difference between 2.7 and 3. And for the purpose of this course, we will work with python 3.

(Refer Slide Time: 01:06)




What is the difference between these two versions? Well, python began with a few features and it kept developing into more versatile programming language. So, python went through much iteration and python 2.7 was a version that was reached when the developers of python decided that there should kind of make a clean start. And some of the new features which had been added in and ad hoc way on to the language should be integrated in a better way which makes it a more robust programming language.

Python 3 essentially is a modern version of python, which incorporates features that where added on to python as it group in a way that makes it more consistent and more easy to use, but as often happens a lot of people had already been using python, and python 2.7 has a lot of software written using that version. In particular a lot of software that people find convenient to use such as scientific and statistical libraries of functions where they do not have to use it themselves, they'll just invoke these libraries are still written using python 2.7. And if you run it from python 3 sometimes these functions do not work as they are expected.

So, this has forced python 2.7 to live on. Eventually we hope that somebody will take the effort to move python 2.7 libraries to python 3. And of course, newer code is largely being developed on python 3, but you should remember that when somebody says that they are using python they could be talking about 2.7 and not 3, and you have to make adjustments.

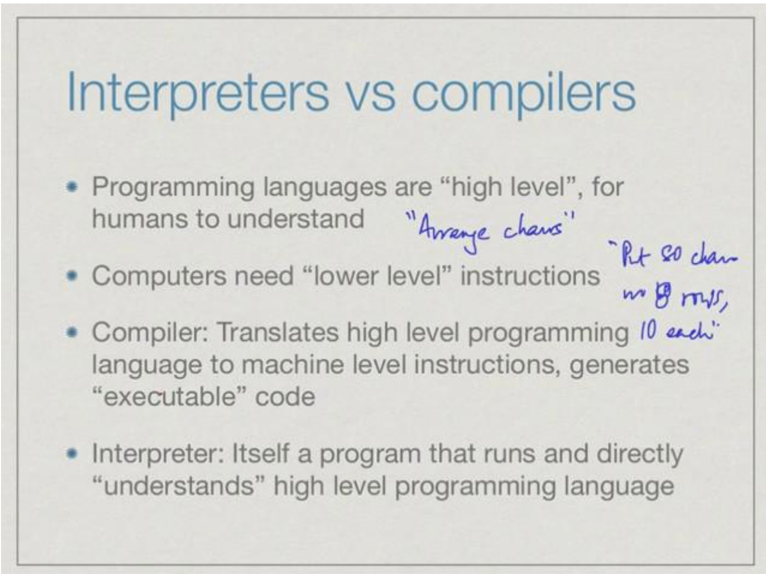
For the purpose of the introductory material that we will be doing in the course, there is almost no change between python 3 and python 2.7; however, there are some features that we will see which are slightly different in 2.7 and we will explore them in 3, and I will try to highlight these differences as we go long. But going forward in python 3 is the current version and it has been the current version for some years now at least for 4 or 5 years. It is definitely the language, which is going to dominate in the future, so it is better that you start with a new version then go back to the old version.

(Refer Slide Time: 03:23)


As far as this course is concerned, any version of python 3 should be fine. The latest version as I said is some 3.5.x, where I thing x is 2, but if you do not have 3.5, but you have 3.4 or 3.3 do not bother everything should work fine. But if you are interested, you can install the latest, latest version. If you are using Linux, it should normally be there by default because many Linux utilities require python and so python should be on you are system, but it could be that the utility is use python 2.7. So, make sure that you install python 3. You can use the package manager to do this. Now if you are using a MAC or you are using Windows then python may or may not be installed especially python 3 may not be installed on your system.

There is the URL given here. If you search on Google, you will find it. Just search for python 3.5 install or download and you will get to this URL. So, www.python.org downloads release python 350. 350, is really refer into 3.5.0. So, actually the current version as I said is not 3.5.0, but 3.5.2. So, you will find instructions there - please download the version that is appropriate for your system and install it. These are designed to be fairly self-explanatory install files; if you have a problem please search online for help with the problem you are facing or ask someone around you. It is not the purpose of this course to spend a lot of time telling you how to install software. So, I hope you are able to do this, so that we can get ahead with the actual programming part.

(Refer Slide Time: 05:05)



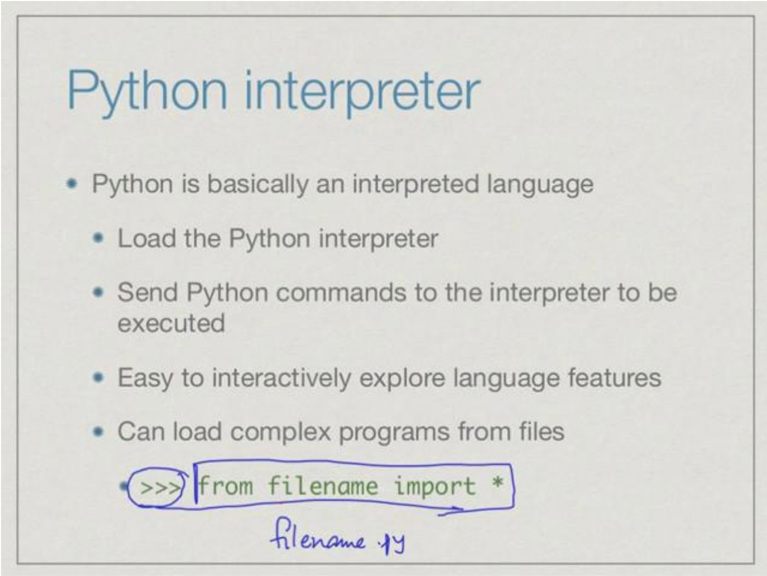

One more thing to keep in mind, if you are familiar with other programming languages, is the distinction between interpreters and compilers. So, the main difficultly is that programming languages like python or C or C++ or Java are written for us to understand and write instructions on. So, these are somewhat high level instructions. In the other hand, computers need low level instructions. So, when we talk about names and values like i, j or we talk about list, the underline computer may not be able to directly analyze these things, so we need a translation. If you remember the very first lecture, we talked about arranging chairs. So, we said arrange the chairs as a high level thing, and we said put 80 chars in 10 in 8 rows, 10 each right.

We said that they could be a difference in the level of detail in which you give instructions and this is precisely what happens. In order to execute something so called executable file that we come across we something which is return at a level that the machine can understand. Whereas, the programs that we are going to explore on this course and which all programmers normally work with are at a higher level, which cannot be directly understood by computer, so we have to bridge this gaps somehow.

A compiler is a program which takes a high level programming language and translates programs on that language to a machine level programming language. So, it takes the high level program in python, if in not python, in C or C++ or Java or something and produces something with directly a machine can execute. In the other hand, the other way of dealing with the high-level language is to interpret it. So, an interpreter in normally English is somebody who stands between people talking different languages and translates back and forth.

An interpreter is a program which you interact with, and you feed the interpreter instructions in your language, in this case python; and the interpreter internally figures out how to run them on the underline machine. So, whether you are running it on Windows, or Mac, or Linux interpreter guarantees that the answer that you see at the high level looks approximately the same independent of the actual platform on which you are running it. So, python is by and large an interpreted language and we should be aware of this fact.

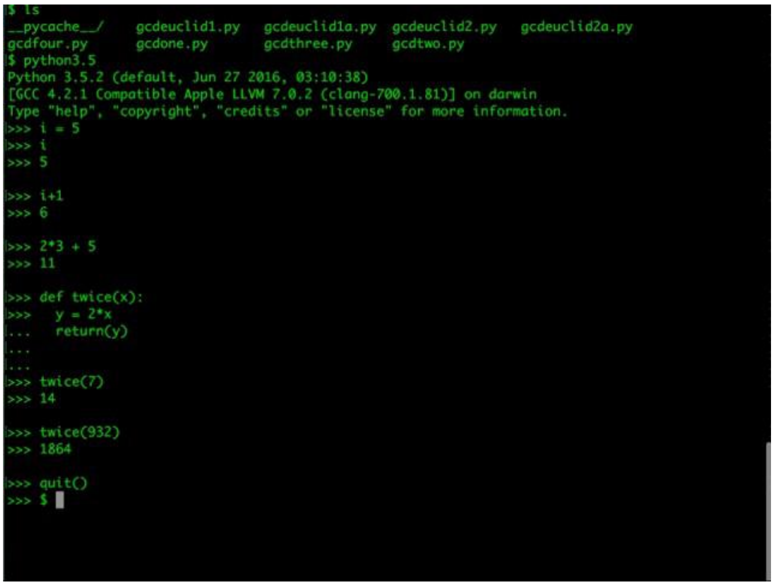
(Refer Slide Time: 07:31)




We use python typically in the following way; we first run the interpreters. So, remember interpreter is the program. We first invoke the interpreter; and when the interpreter is running, we pass python commands to the interpreter to be executed. The nice thing about dealing with an interpreter is that you can play with it like you play with a calculator; you can feed it commands and see what it does, so it is very interactive. Of course, it is tedious, if you have to type in large programs, so there is a way to load a program which has been written already using a standard text editor and loading it from a file. So, what I have shown below in green is so this is what we will see in a minute is the prompt that the interpreter shows you.

When you enter the interpreter, it will ask you to execute a command and this is a command that you provide the interpreter. It says. So, I have stored. I have a file called say file name dot p y typically to indicate it is a python program from that file import all the definitions and functions and code that is written there. So, this will tell the interpreter to take everything that is written in that code and put it into its current environment, so that those functions can be used. So, these things will become a little clear and then in the demo that I am just going to show you and then you can play around with this. And then the next week, we will get into the real details about exactly what goes into a python program.

(Refer Slide Time: 09:16)




Here is a window showing the terminal which on Windows would be like a command prompt and using unique like shell. So, if I say ls, it shows me the list of files in my current area. And all this files with extension dot p y are actually python programs. In this, I invoke the python interpreter by saying python 3.5 because that is the version which I am using. If I invoke it, it will produce some messages telling me what type of function system I'm on. So, it tells me that I am using for instance 3.5.2 and it has may that it is a fairly recent version, it tells me that it is on a Apple and blah blah blah, but what is important is then produces a prompt place where I can enter commands and this is signified by these three greater times.

Now, at the python interpreter prompt, you can directly start writing things. So, for example, you can say i is equal to 5. What it says as a take a name i assign to value of 5. Now if I type i, it tells me that the value is 5; if I type an expression like i plus 1, it tells me that is 6. So, you can use it as a calculator. So, you can do simple arithmetic if you want. So, you can keep interacting with it. Now, you can also define functions remember how we defined a function, we use def, use a function name and so on. So, we can say for example, def twice x. This is the function twice, this takes the single argument x. And as you might expect I would like to return two times x.

Now a python uses as we mentioned in one of the earlier lectures, indentation in order to specify that something is a part of something else. So, the definition consists of a bunch of it steps. So, I must tell it that these bunches of steps belong to this definition by indenting it; it does not matter how you indent it as long as you use the same indentation uniformly. If you are using two spaces, use two spaces use a tab, but do not mix up the number of spaces and do not mix up tabs and spaces, because this gets you confuse the error messages form python. So, let us use two spaces.

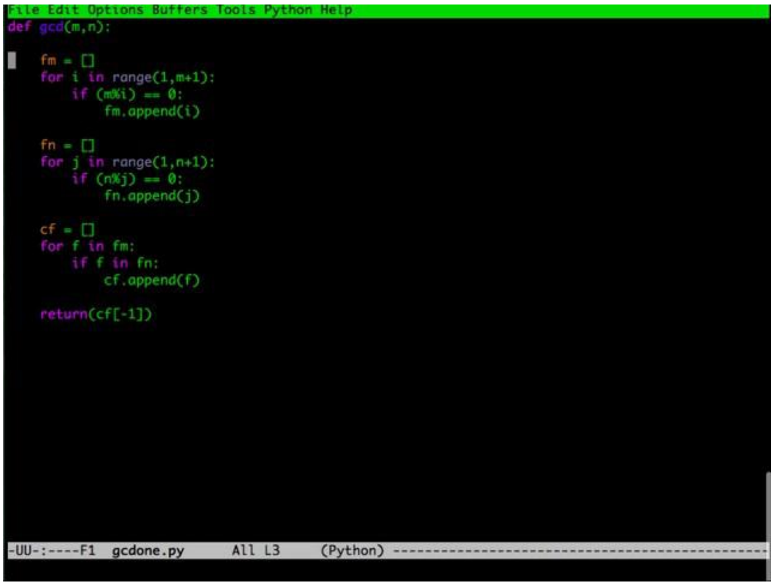
Let us to the sake of illustration create a new name y, and say y is two times x. Now it is still continuing to ask me for the definitions, so the prompt has change to dot dot dot. Now I must induct it a same way and say return y. So, what I have done is I say this function takes in value x, computes two times x, and stores it in the name y, and returns the value of the name y, right. Now, when I am done with this, I give a blank line and this function is now defined. Now, twice 7 makes sense, what twice 932 will also make sense right. So, python is very convenient in that you can have few define functions as you go along on the fly.

Now, we could also define our gcd right here, but as you might expect sometimes a function is too complicated to typing without make in a mistake, and secondly, you might want to play around with the function and change it and not have to keep typing it again and again. For this, what we need to do is first type the function in to a file and then load the file here. Let us get out of this. So, one way to get out of this is to type quit the brackets.

(Refer Slide Time: 12:49)

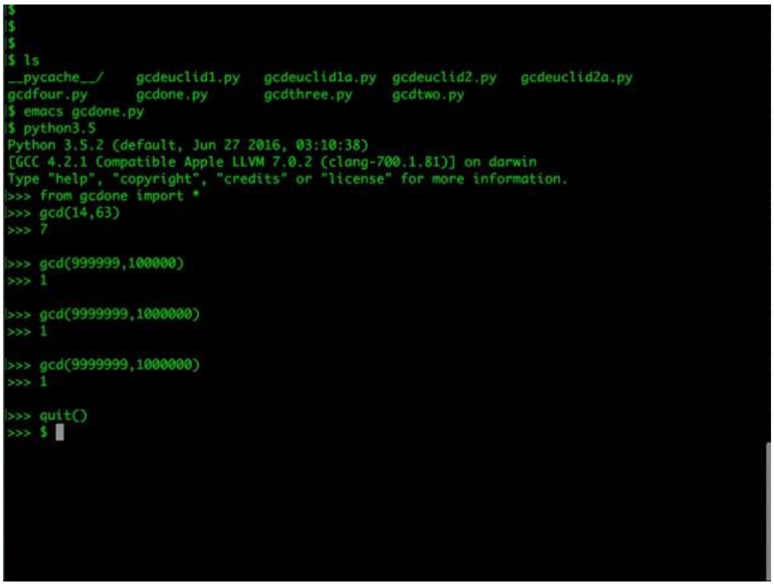

And then you get back to this prompt which is dollar which is the outside terminal or the command prompt. So, I have actually already created something. Let us start with, so I use an editor called emacs, you can use any takes editor if you are using Windows, you can use notepad, if you are using and Linux, you can use emacs or vi or you can use some simpler editor like gedit or k, anything that is comfortable, but it should just be a text editor it should not do any formatting, do not use word processes like you know office or something like that. You something we just manipulates text files.

(Refer Slide Time: 13:27)




If I look at gcd 1 dot py, so one nice thing what emac is it shows me colors to indicate certain things. So, def this is the very first gcd program we wrote, which takes computes the list f m then the list f m then the list cf, and then it returns the last elements in cf. So, this is the first version of gcd. So, this is the exactly the code we wrote before. The point to remember is that I have made sure that all these indentations are at the same number of spaces in. So, this is something to remember. Now, you typing something like this right then you save it and exit.

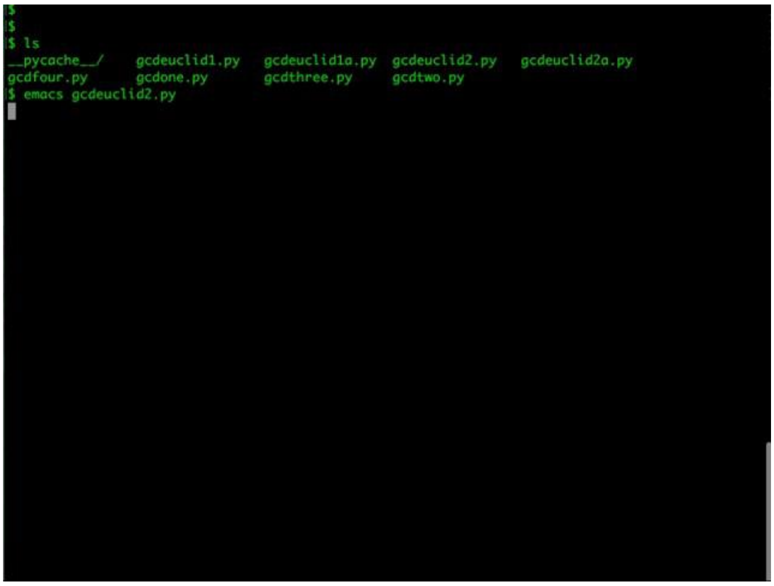
(Refer Slide Time: 14:03)




Now you go back to your python, and you save from that file gcd 1 import star what this means is take the file gcd1 dot py and load all the functions which had defined there and make them available to me here. Now, if I say gcd of 7 comma let us for example, 14 and 63 for instance, it tells me the gcd 7. Now if you take some large number like 9999 and 10000 then it takes, so may be one more digit let us see, you will notice that it is not giving me an answer and then it gives me answer. So, it this is just to illustrate that this was the slow gcd right. So, see how much time it took.

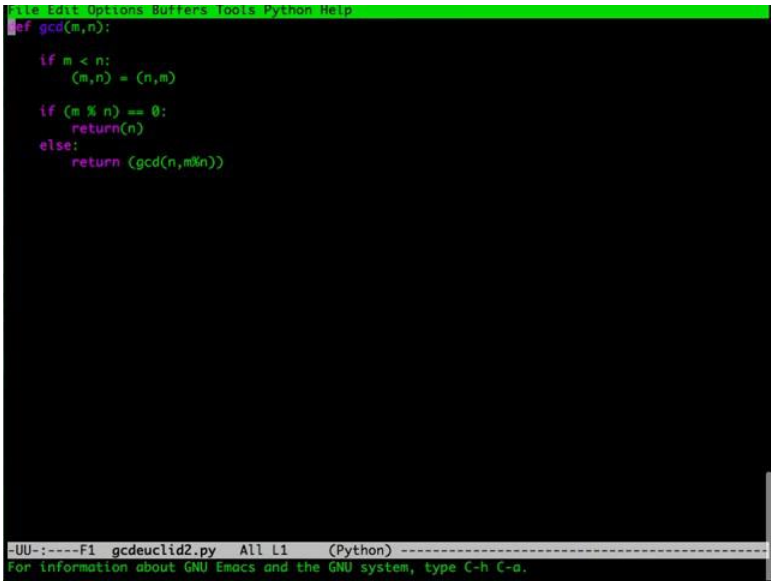
It has the visible gap of a few seconds before it produces the answer. And this is the illustration that this is not a very efficiency gcd. So, one of the problems with this python interpreter which I will see if we can solve is that if I have already loaded one file then it is safer to exit and then reload other file rather than to update the file.

(Refer Slide Time: 15:25)



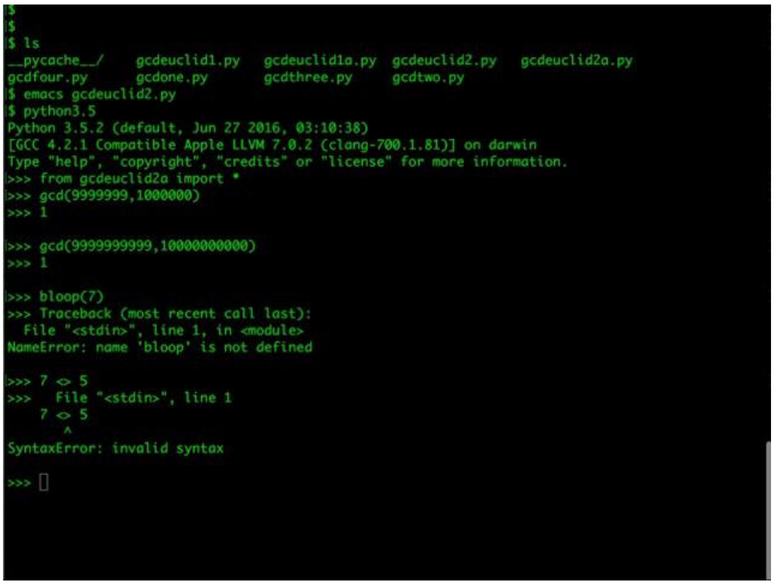

Let me reload for instance the last version of Euclid's thing, which we wrote which is the reminder version.

(Refer Slide Time: 15:32)




It says that if m less then n exchange the values if then the second line here says that if the reminder of m divided by n is 0 that is n is a divisor of m then return n otherwise replace the g c d call by the call to n and its reminder. So, this we also had a version of this where we return to the while loop. Let us use the while version. The while version says that so long as the reminder is not 0, we keep updating m and n to n and the reminder, and finally you return the value of n.

(Refer Slide Time: 16:13)

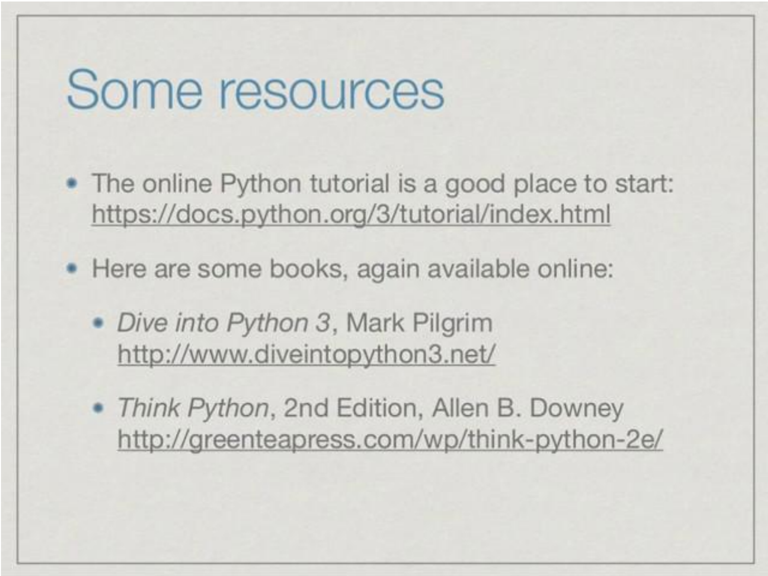



I am going to take this particular thing and load it into python. So, again I first invoke the interpreter python then I say from gcdeuclid2a import star. Now I am going to give that same large value that we saw before and which I think was say 9999999 and 1000000. And now you see, you get an instant answer. In fact, you will see that if I even if I give it several more digits, it should hope fully work fast. So, there is a dramatic improvement in speed which is even visible in this simple example, if we replace the naive idea by a clever idea.

The power of algorithm is to actually make a program which would otherwise be hopelessly slow work at a speed which is acceptable to you. Do a load python on your system, invoke the python interpreter and play around with the code that we have seen in this particular week's thing, make errors see what python tells you when you import a file which has errors. For instance now if I try to ah invoke a function which does not exists like, if I use a function which I have not defined and which python does not understand then it will give me a mistake like this. It will say loop is not defined. If I write something strange like 7 less than greater than 5, then it will say that this is invalid syntax.

The interpreter will look for an expression if the expressions do not make sense then it is going to complain. And sometimes the error messages are easy to understand, sometime they are less easy to understand; as we go along we will look into this. But, the purpose of the interpreter is to either execute what you have given it or tell you that what you have written is somehow not executable and explains why. So, do play around with it and a get some familiarity because this is what going to be our bread and butter as we go along.

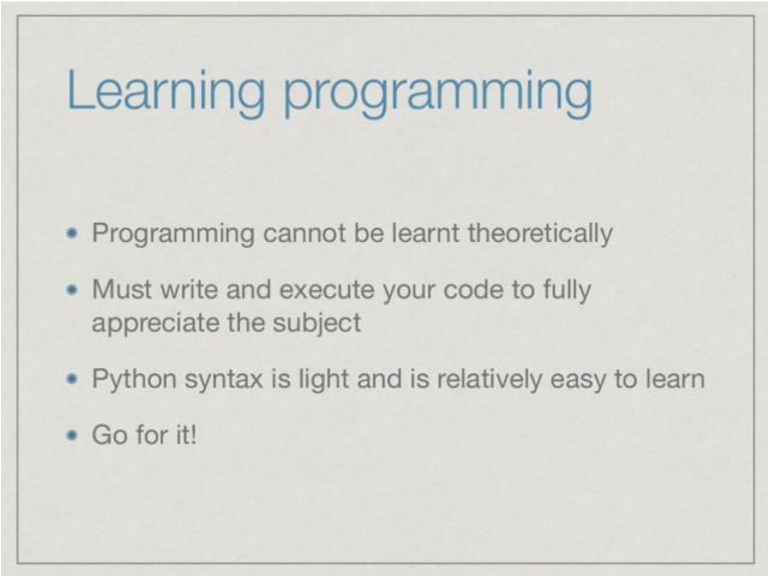
(Refer Slide Time: 18:18)




We are going to be looking at some specific features of python in this course, but you may find as we go along that there is something that you do not understand or something new that you would like to try out your own. So, it is always a good idea to have access to other resources. The python online documentation is actually an excellent place to look for details about python and in particular, there is a very readable tutorial; especially, if you already have some familiarity with programming the python is probably the best place to start learning python for yourself. So, here is a URL, docs.python.org/3 this is for python 3 tutorial index dot html. If you just go to docs.python.org/3, you will find there are also more detailed reference manuals and so on, which you might need at a later stage.

Do keep this as one of the places that you look when you have difficulties. And there are two books which probably useful to understand python beyond what is covered in the lectures if you feel that something is not clear. So, there is this book called dive into python which is adapted for python 3. And there is book called think python which is about generally about computational thinking in the context of python. Both of these have the nice advantage that they are available online, so you do not have a buy anything; you can just browse them through your browser on the net.

(Refer Slide Time: 19:41)




Before we leave you for this week, remember that learning programming is an activity; you cannot learn programming theoretically. You have to write and execute code to appreciate the subject. You have to make mistakes; learn from your mistakes; figure out what works, what does not work and only then will you get a true appreciation for programming. Reason we are going with python is because python has a very simple syntax compared to other programming languages. We have already without formally learning python, seen some fairly sophisticated programs for gcd and hopefully you have understood them even if you cannot generate them. It is not very difficult to explain what a python program is doing with a little bit of understanding.

Do take the time to practice the examples that we had seen this time. We will be giving programming exercises as we go along; and unless you do these exercises and become somewhat handy at manipulating python yourself, you will never truly learn both programming and python. The other thing to remember is that once you have learned one language, even though the features and the syntax vary from language to language, it is very easy to pick up a another language, because all of programming has at it is base very similar principles.

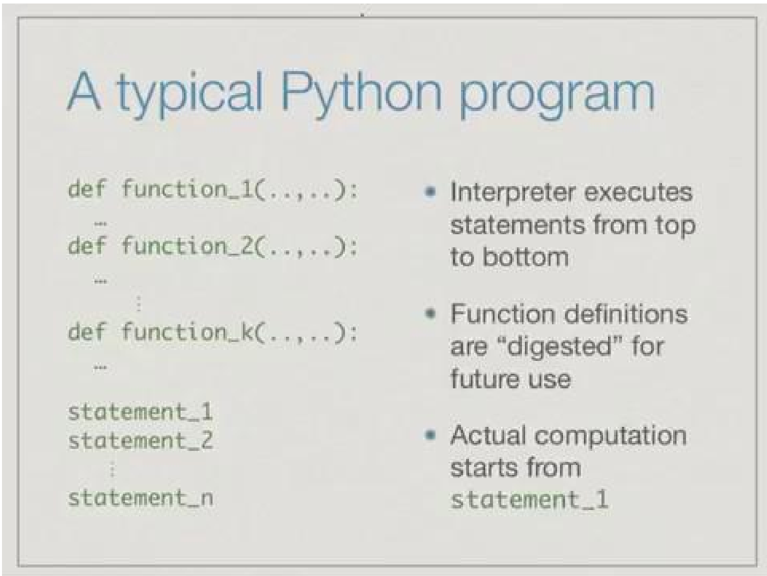
Although the syntax may vary, the ideas do not. The ideas are eventually what write the program, but to be a fluent speaker of a programming language, you must practice it. So, do try.

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

**Week - 02**
**Lecture - 01**
**Assignment statement, Basic types – int, float, bool**

Last week, we were introduced to notation of algorithms using the gcd example. We also saw informally some python code, which we could understand but we have not actually been introduced to formal python syntax. Let us start with some real python step.

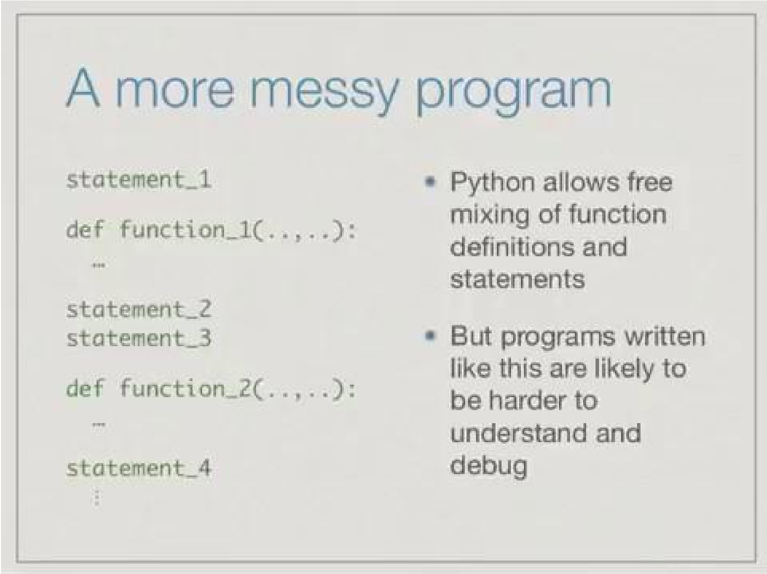
(Refer Slide Time: 00:17)




A typical python program would be written like this, we have a bunch of function definitions followed by a bunch of statements to be executed. So remember that we said python is typically interpreted, so an interpreter is a program, which will read python code and execute it from top to bottom. So, the interpreter always starts at the beginning of you of python code and reads from top to bottom.

Now, function definition is a kind of statement, but it does not actually result in anything happening, the python interpreter merely digests the function kind of remembers the function definition. So that later on if an actual statement refers to this function it knows what to do. In this kind of organization the execution would actually start with the statement which is called statement 1. So, first you will digest k functions and then start executing 1, 2, 3, 4 up to statement n.

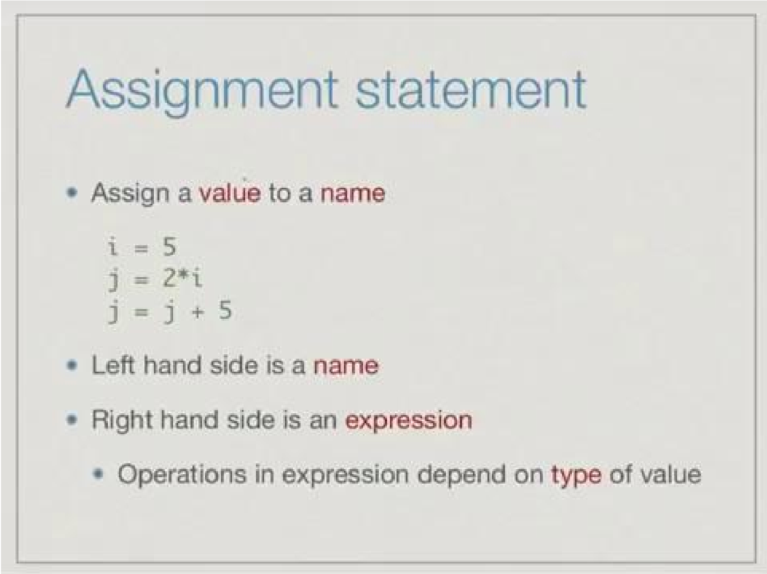
(Refer Slide Time: 01: 09)




Now there is no reason to do this. So, python actually allows you to freely mix function definitions and statements, and in fact, function definitions are also statements of a kind its just they do not result in something immediately happening, but rather in the function been remembered.

But one of things that python would insist is that if a function is used in a statement that has to be executed that function should have been defined already; either it must be a built in function or its definition must be provided. So, it may use this kind of jumbled up order, we have to be careful that functions are defined before they are used. Also jumbling up the order of statements and function definitions in this way, makes it much harder to read the program and understand what it is doing. Though it is not required by python as such as, it strongly recommended that all function definition should be put at the top of the program and all the statements that form the main part of the code should follow later.

(Refer Slide Time: 02:07)

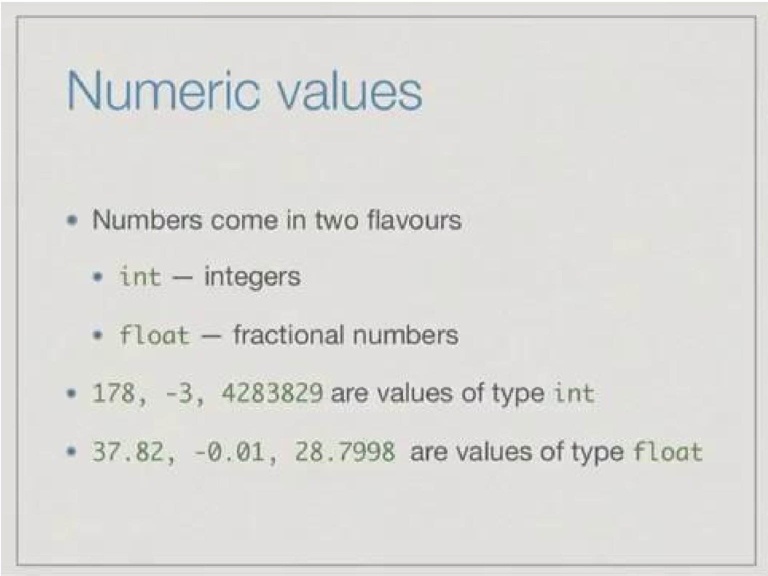



What is a statement ? The most basic statement in python is to assign a value to a name. So, we see examples and we have seen examples and here are some examples. So, in the first statement i is a name and it is assigned a value 5; in the second statement, j is a different name and it is assigned an expression 2 times i. So, in this expression, the value of i will be substituted for the expression i here. So, if i have not already been assigned a value before, python would not know what to substitute for i and it would be flagged as an error.

When you use a name of the right hand side as part of an expression, you must make sure that it already has a valid value. And as we saw, you can also have statements which merely update a value. So, when we say j is equal to j plus 5 it is not a mathematical statement, where the value of j is equal to the value of j plus 5. But rather that the old value of j which is on the right hand side is updated by adding 5 to it and then it gets replaced as a new value j. This is an assignment statement, this equality assigns the value computed form the right hand side given the current values of all the names if the name given on the left hand side, with the same name can appear on both sides.

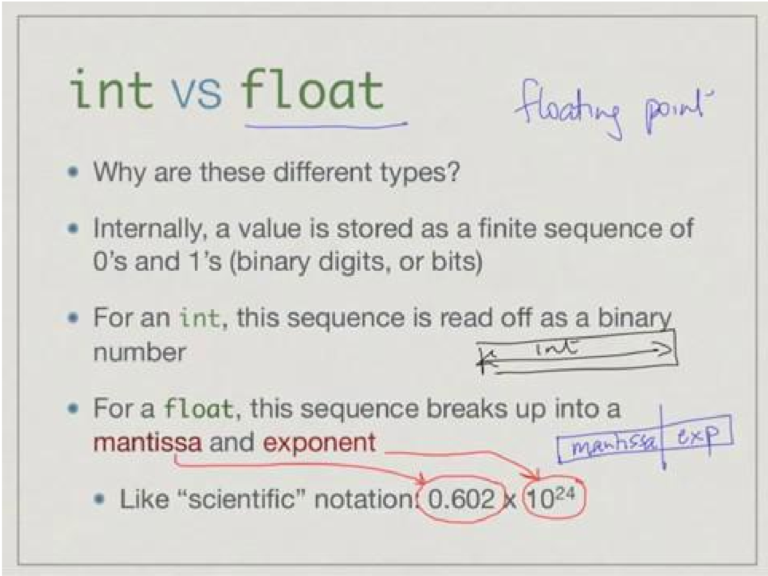
The left hand side is a name and the right hand side in general is an expression. And in the expression, you can do things which are legal, given the types of values in the expression. So, values have types; if you have numbers, you can perform arithmetic operations; if you have some others things, you can perform other operations. So, what operations are allowed depend on the values and this is given technically the name type. So, when we said type of values it is really specifying what kinds of operations are legally available on that class of values.

(Refer Slide Time: 04:01)




So the most basic type of value that one can think of are numbers. Now in python and in most programming languages numbers come in two distinct flavours as you can call them integers and numbers which have fractional parts. So, in python these two types are called int and float. So, int refers to numbers, which have no decimal part, which have no fractional part. So, these are whole numbers they could be negative. So, these are some examples of values of type int. On the other hand, if we have fractional parts then these are values of type float.

(Refer Slide Time: 04:43)




Normally in mathematics we can think of integers as being a special class of say real numbers. So, real numbers are arbitrary numbers with fractional parts integers are those real numbers which have no fractional. But in a programming language there is a real distinction between these two and that is, because of the way that these numbers are stored. So, when python has to remember values it has to represent this value in some internal form and this has to take a finite amount of space.

If you are writing down, say a manual addition sum you will write it down on a sheet of paper and depending on a sheet of paper and the size of your handwriting there is a physical limit to how large a number you can add on that given sheet of paper. In the same way any programming language, will fix in advance some size of how many digits it uses to store numbers and in particular as you know almost all programming languages will internally use a binary representation. So, we can assume that every number whether an integer or real number is stored as a finite sequence of zeroes and ones which represents its value.

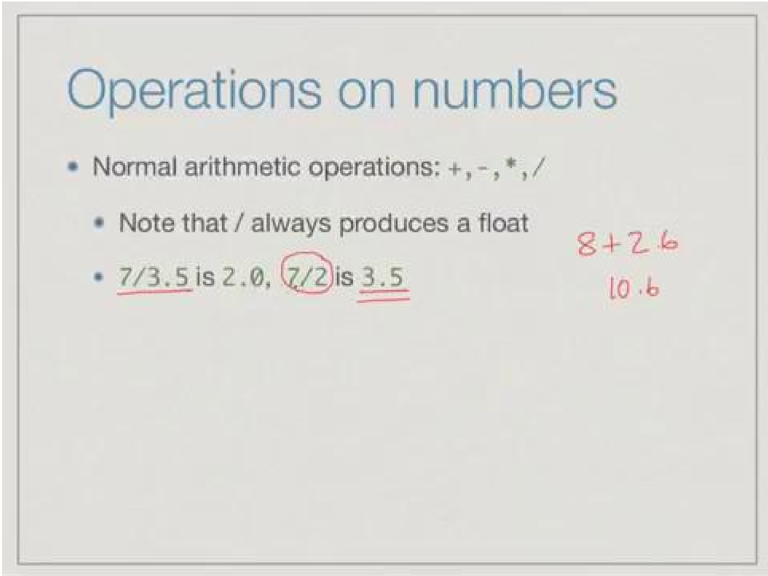
Now, if this happens to be an integer you can just treat that binary sequence as a binary number as you would have learnt in school. So, the digits represent powers of 2, usually there will be one extra binary digit 0 or 1 indicate whether it is plus or minus and they may be other more efficient ways of representing negative numbers, but in particular you can assume that integers are basically binary numbers.

They are just written as integers in binary notation. Now when we come to non integers then we have two issues one is we have to remember the value which is the number of digits which make up the fractional part and then we have to remember the scale. So, think of a number in scientific notation right, so, your normally have two parts when we use things in physics and chemistry for instance, we have the value itself that is what are the components of the value and we have how we must shift it with respect to the decimal point. So, this says move the decimal point 24 digits to the right.

So, this first part is called the mantissa right and this is called the exponent. So, when we have the number in memory if it is an int, then the entire string is just considered to be one value where as if we have block of digits which represents a float. Then we have some part of it, which is the mantissa, and the other part, which is the exponent.

The same sequence of binary digits if we think of it as an int has a different value and if we think of it as a float has a different value. So, why float you might ask. Float is an old term for computer science for floating point; it refers to the fact that this decimal point is not fixed. So, an integer can be thought of as a fixed decimal point at the end of the integer a floating point number is really a number where the decimal point can vary and how much it varies depends on the exponent. So there are basically fundamental differences in the way you represent integers and floating point numbers inside a computer and therefore, one has to be careful to distinguish between the two. So, what can we do with numbers?

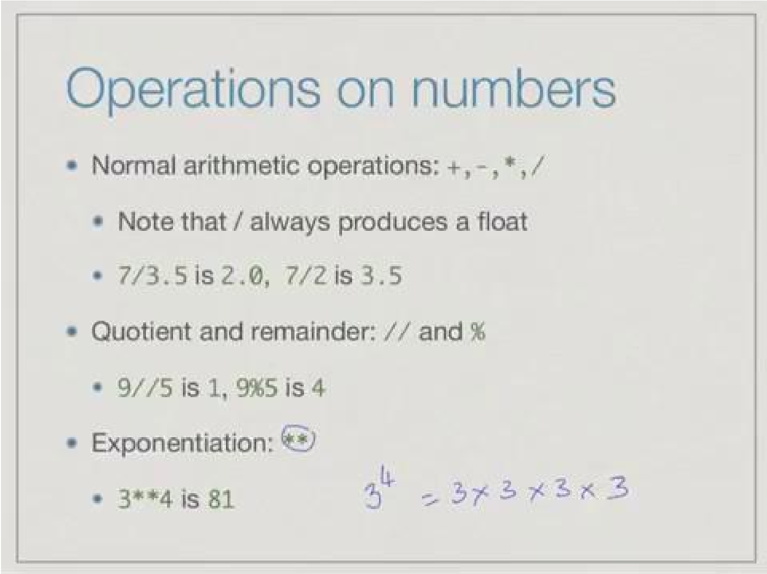
(Refer Slide Time: 07:59)




Well we have the normal arithmetic operations plus, minus, multiplication, which has a symbol star modern x and division, which has a symbol slash. Now notice that for the first three operations it is very clear if I have 2 ints and I multiply them or add them or subtract them I get an int. But I have 2 floats I will get a float, division, on the other hand will always produce a float if i say 7 and divided by 2 , for instance where both are ints I will get an answer 3 point 5.

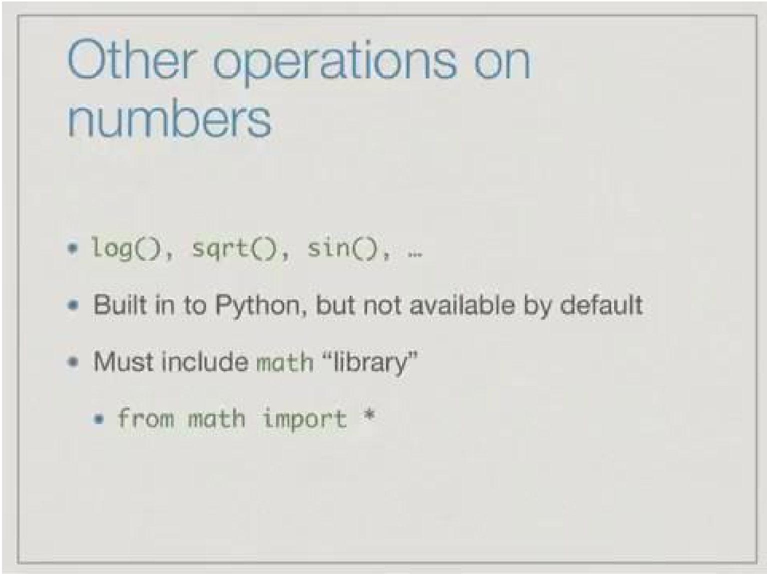
Now in general python will allow you to mix ints and floats, so i can write 8 plus 2 point 6 even though the left is an int and right is a float and it will correctly give me 10 point 6. In that sense python respects the fact that floats are a generalized form of int. So, we can always think of an int as being a float with a point 0 at the end. So, we can sort of upgrade an int to a float if you want to think of it that way and incorporate with an expression, but division always produces floats. So, 7 divided by 3 point 5 as an example of a mixed expression, where I have an int and float and this division results in 2 point 0 and 7 by two results in 3.5.

(Refer Slide Time: 09:15)



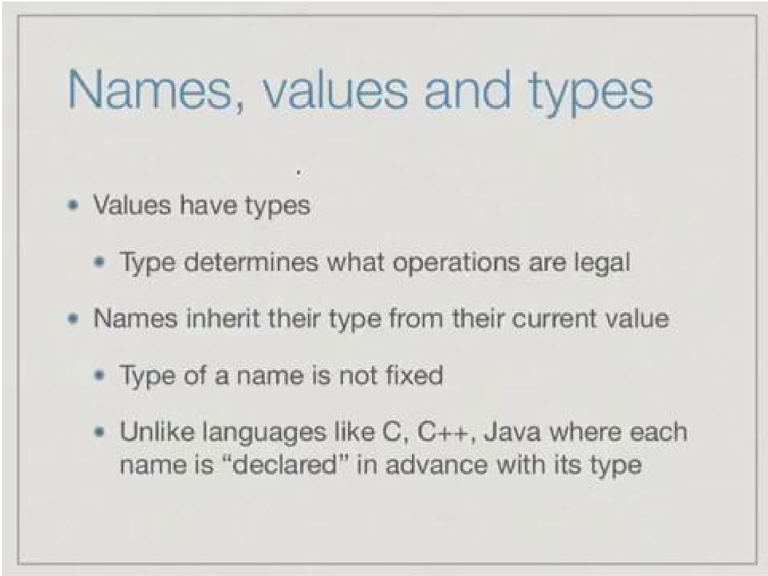

Now there are some operations where we want to preserve the integer nature of the operands. We have seen one repeatedly in gcd which is the modulus operator, the remainder operator. But the req corresponding operator that go through the reminder is the quotient operator. So, if I use a double slash it gives me the quotient. So, 9 double slash 5 says how many times 5 going to 9 exactly without a fraction and that is 1 because in a 5 times 1 is 5 and 5 times 2 is 10 which is bigger than 9 and the remainder is 4. So, 9 percent 5 will be 4. Another operation which is quite natural and common is to raise one number to another number and this is denoted by double star. 3 double star 4 is what we would write normally as 3 to the power 4 is 3 times, 3 times, 3 four times right and this is 81.

(Refer Slide Time: 10:12)




Now there are more advanced functions like log, square root, sin and all which are also built into python, but these are not loaded by default. If you start the python interpreter you have to include these explicitly. Remember we said that we can include functions from a file we write using this import statement. There is a built in set of functions for mathematical things which is called math. So, we must add from math import star; this can be done even within the python program it does not have to be done only at the interpreter. So, when we write a python program where we would like to use log, square root and sin and such like, then we should add the line from math import star before we use these functions.

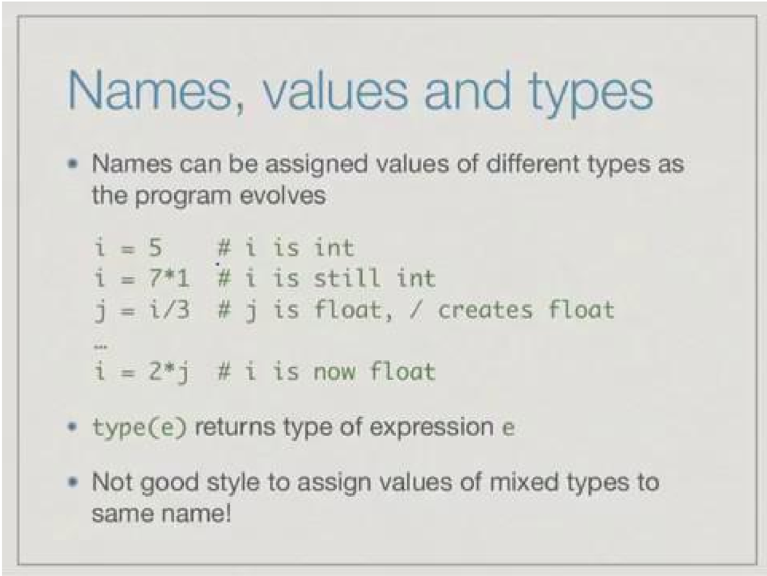
(Refer Slide Time: 10:58)




We have seen three concepts - names which are what we use to remember values, values which are the actual quantities which we assign to names and we said that there is a notion of a type. So, type determines what operations are legal given the values that we have. So, the main difference between python and other languages is that names themselves do not have any inherent type. I do not say in advance that the name i as an integer or the name x is a float. Names have only the type that they are currently assigned to by a value that they have.

The type of a name is not fixed. In a language like C or C++ or Java we announce our names in advance. We declare them and say in advance what type they have. So, if we see an i in an expression we know in advance that this i was declared to be of type int this x was declared to be of type float and so on. Now in python this is not the case.

(Refer Slide Time: 12:00)




So, let us illustrate this with an example. So, the name main feature of python is that a name can be assigned values of different types as the program evolves. So, if we start with an assignment i equals to 5 since 5 is an int i has a type int. Now if we take an expression, which produces an int such as 7 times 1, i remain an int. Now if we divide the value of i by 3. So, at this point if we had followed the sequence i is 7. So, 7 by 3 would be 2.33 and this would be a float.

Therefore, because the operation results in a float at this point j is assigned the value of type float. Now if we continue at some later stage we take i and assign it to the value 2 times j, since j was a float i now becomes float. In the interpreter there is a useful function called type. So, if you type the word type and put an expression and either a name or an expression in the bracket, it will tell you actually type of the expression.

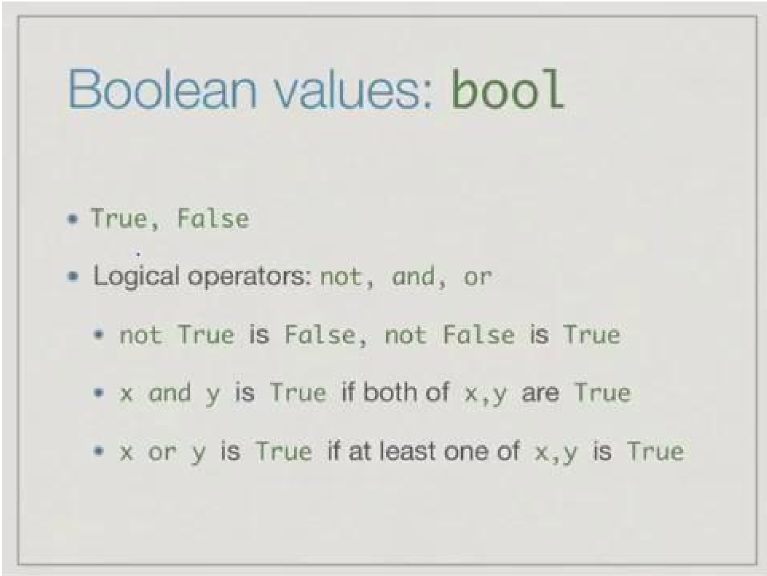
Now although python allows this feature of changing the type of value assigned to a name as the program evolves, this is not something that is recommended. Because if you see an i and sometimes its a float and sometimes its an int it is only confusing for you as a programmer and for the person trying to understand your code. The same way that we said before that we would like to organize our python code so that we define all functions before we execute statements, it is a good idea to fix in advance in your mind at least, what different names stand for and stick to a consistent way of using these either as ints or as floats.

(Refer Slide Time: 13:54)


Let us execute some code and check that what we have been saying actually happens. So, supposing we start the python interpreter and we say i is equal to 5, then if we use this command type i it tells us type of i. So, it returns it in the form which is not exactly transparent, but it says that i is of class int. So, you see the word int, if i say j is equal to 7 point 5 and i ask for the type of j then it will say j is of class float. So, the names int and floats are used internally to signify the types of these expressions. Now if I say i is equal to 2 times j as we suggested i has a value 15 point 0, because j was a float and therefore, the multiplication resulted in a float and indeed if we ask for the type of i at this point it says that i is now a float.

The point to keep in mind is that the name is themselves do not have fixed types they are not assigned types in advance. It depends on the value that is currently stored in that name according to the last expression that was assigned.

(Refer Slide Time: 15:05)



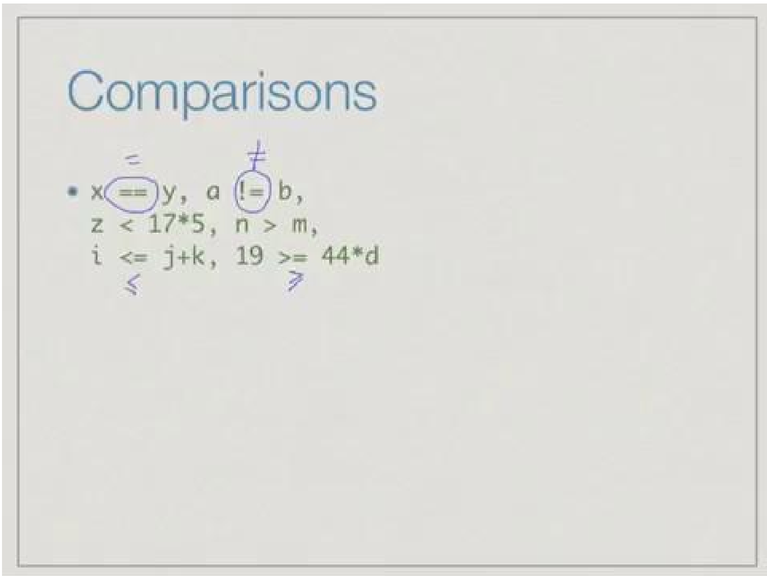

Another important class of values that we use implicitly in all our functions are Boolean values which designate truth or falseness. So, there are two constants or two basic values of this type which in python are called true with the capital “T” and false with the capital “F”. So, true is the value which tells something is true. So, when we remember we wrote conditions like if something happens if x is equal to y do something, x mod 7 is equal to something, to something in our gcd function. So, the output of such an expression where we compare something to another expression compare an expression on the left to an expression on the right is to determine whether this comparisons succeeds or fails when it succeeds it is true and when it fails it is false.

These are implicitly used to control the execution of our program. So, we need to have a way of recording these values and manipulate it. The basic values are true or false and typically there are three functions which operate on these values. So, not negates the value. So, true is the opposite of false. So, not applied to true will give us false not applied to false will give us true and follows the usually English meaning of and so, when we say that something is true and something else is true we mean that exactly both of them are true. So, x and y two values of Boolean type will be the expression x and y will be true provided at the moment x has a value true and y also has a value true. If either of them is not true then the output x and y is false.

“Or” again has an English meaning, but the meaning in computer science and logic is slightly different from what we mean. So, normally when we say or we mean 1 or the other. So, you might say either i will wake up in time or i will miss my bus. So, what you will mean is that one of these two will happen it is unlikely that you mean that you will wake up in time and you will miss your bus.

It is when we use or in English we usually mean either the first thing will happen or the second thing will happen, but not both, but in computer science and logic or is a so, called inclusive or not exclusive, its not exclusively one will happen or the other, but inclusive both may happen. So, x or y is true if at least one is true. So, one of them must be true, but it also possible when both are them true.

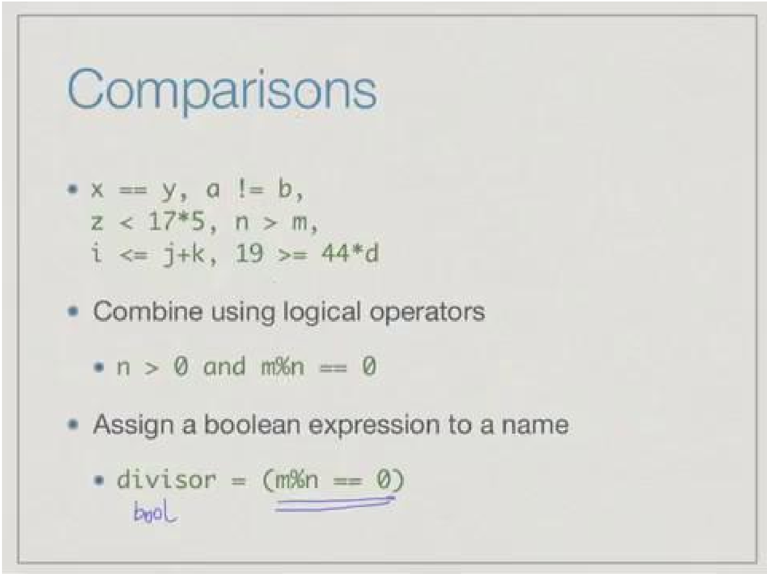
(Refer Slide Time: 17:28)




The most frequent way in which we generate Boolean values is through comparisons we have already seen the two of these. So, we have seen equal to - equal to. This is the actual equality of mathematics not the single equal to which is the assignment. So, if x equal to equal to y checks, whether the value of x is actually the same as the value y and if so, it returns the value true otherwise, it returns false.

And the corresponding inequality operator is exclamation mark followed. So, this is not equal to exclamation is equal to is a symbol for not equal to and this is the usual mathematical. And then of course you have for values which can be compared as smaller or larger you have less than, greater than this is less than equal to and this is greater than equal to. So, we have these 6 logic logical comparison operators' arithmetic comparison operators which yield a logical value true or false.

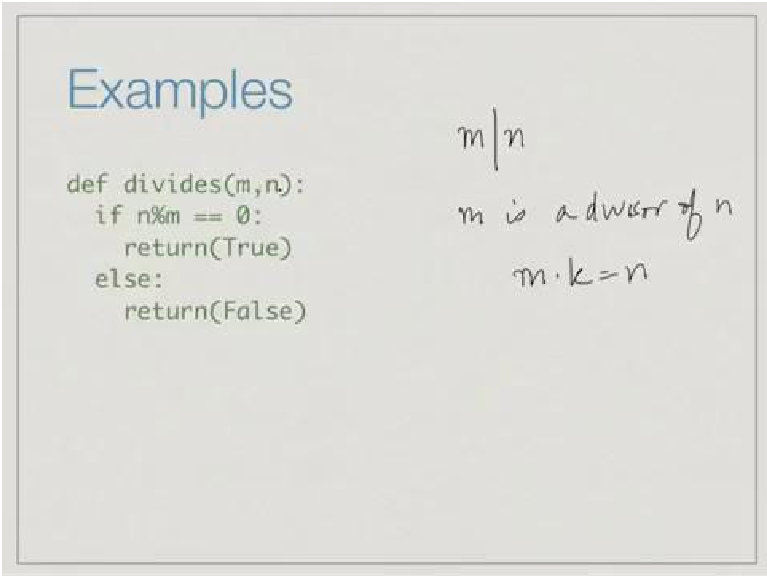
(Refer Slide Time: 18:24)




And the usual thing we will do is combine these. So, we might want to say that check if the reminder when divided by n is 0 provided n is 0 not 0. So, if we say n is greater than 0 and this it will require n to be number bigger than 0 and the reminder n divided by n to be equal to 0. So, this says n is a multiple of n and n is not 0. And we can take an expression of this file kind of comparison, which yields as we said a Boolean value, and take this Boolean value and assign it to a name.

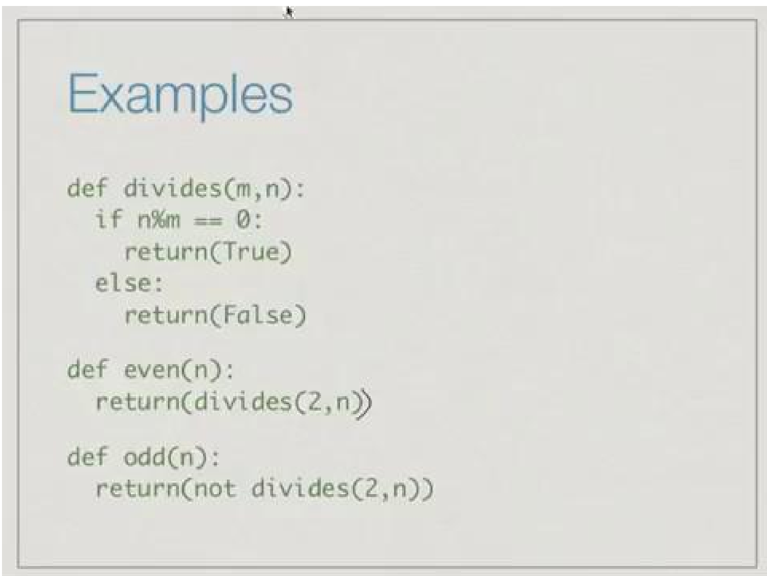
So, we can say that n is a divisor of m if the reminder of m divided by n is equal to 0. And we can say that the fact that it is a divisor its true provided this happens. So, divisor is now of type bool right and it has a value true or false depending or not whether or not n divides m evenly.

(Refer Slide Time: 19:22)




Let us look at an example of how we would use Boolean values. So, let us get back to the divides example. In mathematics we write m divides n to say that m is a divisor of n. So, this means that m times k is equal to n for some k. So, m divides n if the reminder of n divided by m is 0. If so you return true else you return false right. This is a very simple function it takes two arguments and checks if the first argument divides the second argument.

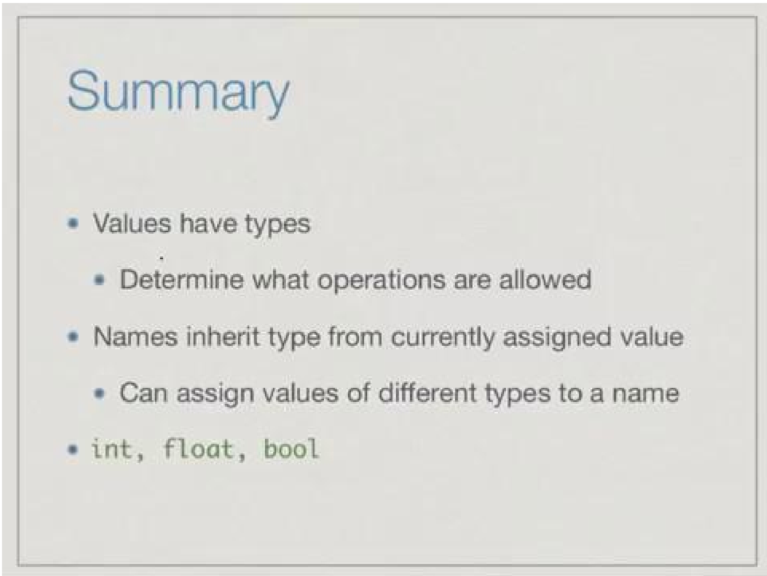
(Refer Slide Time: 20:00)




Now what we can do is define an another function called even whose value is derived from here. So, we check whether two is a divisor of this number. So, we check whether 2 divides n; if 2 divides n, then n is even, we return true; if 2 does not divide n, n is odd, we return false. So, similarly we could say define odd n else the negation of the previous case. So, if 2 divides n then n is not odd.

You take the answer about whether 2 divides n or not, and reverse it to get the answer odd. So, if 2 divides n, you negate it and say odd is false; if 2 does not divide n, you get false back and you negate and say odd is true. So, we just wanted to emphasise that Boolean values can be computed, assigned, passed around just like numerical values are.

(Refer Slide Time: 20:50)




To summarise what we have seen is that the basic type of statement is to assign a name to a value values have type and these determine what operations are allowed. So, we can use for instance arithmetic operations on numeric types, we can use logical operations like and, or, and not on Boolean types, but the important difference between python and traditional languages where we declare names in advance is that python does not fix types for names. So, we cannot say that 'i' has the type int forever; 'i' will have a type depending on what it is assigned. A name inherits the type from its currently assigned value and its type can change as a program evolves depending on what values have been assigned.

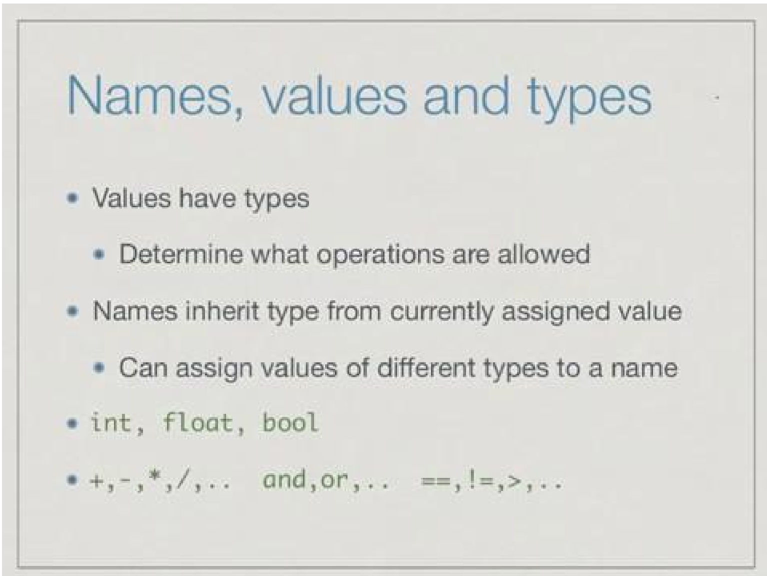
What we have seen in this particular lecture are 3 basic type int, float and bool. As we go along this week, we will see more types with interesting structures and interesting operations defined on them.

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

**Week - 02**
**Lecture - 02**
**Strings**

We have seen now that python uses names to remember values. Values are the actual quantities that we manipulate in our program, these are stored in names. Values have types, and essentially the type of a value determines what operations are allowed.

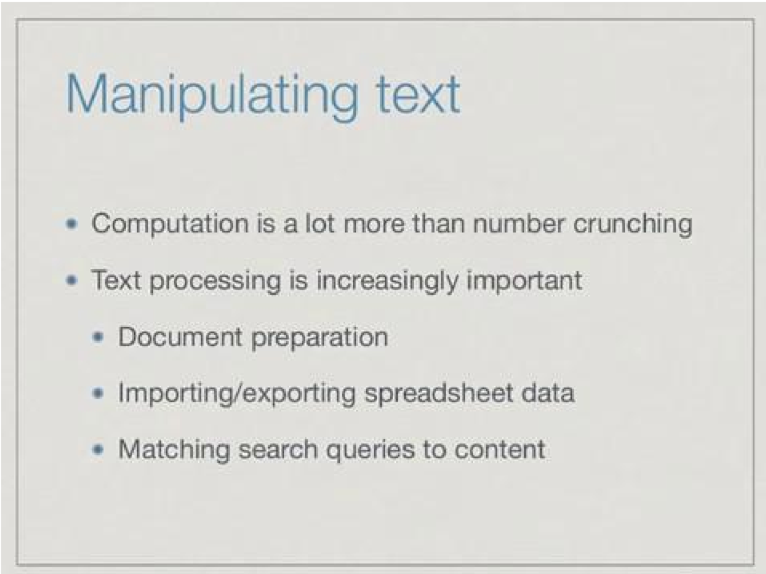
(Refer Slide Time: 00:02)




The types we have seen are the basic numeric types - int and float, and the logical type bool which takes values true or false. So, for the numeric types, we have arithmetic operations, we also have other operations which are more complicated. For the Boolean types we have and, or, not, which allows us to manipulate true and false values. And then we have these comparison operators equal to, greater than and so on, which allows us to check the relative values of two different quantities, and decide whether they are in some order with each other.

The important thing that we said was that in python the names themselves do not have a fixed type. So, we cannot say that i is of type int or x is of type float, rather it depends on what values assigned and in particular, if a name is used for the first time without assigning a value then python will complain. We do not have to announce names in advance like other programming languages, but whenever we first use a new name; its first use must be in an assignment statement on the left hand side. So, before we use a name in an expression on the right hand side it must be assigned a valid value.

(Refer Slide Time: 01:36)

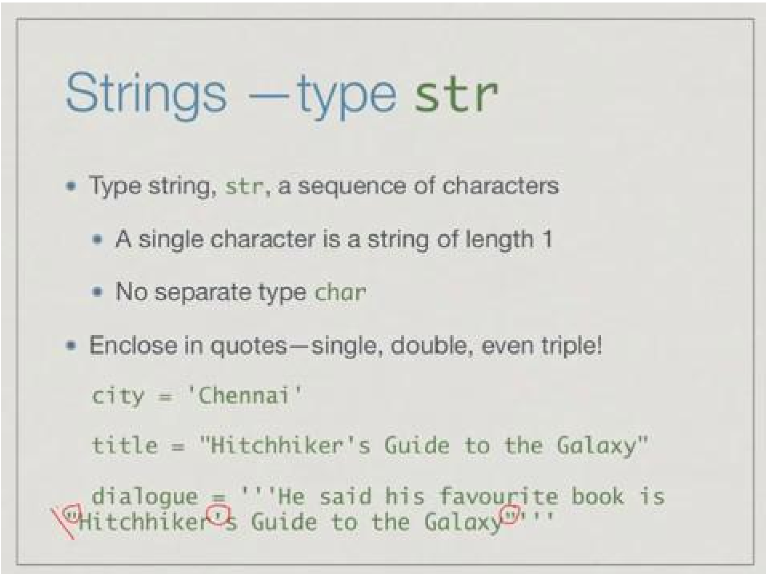



Numeric types by no means the only things that are of interest these days in computation. A lot of the computation we do is actually dealing with text. So, whenever we prepare a document, for example, using a word processor or some other things for presentation, then we are actually manipulating text; so we are moving text around, searching for something to replace and so on. Also when we are manipulating data itself, very often data comes from multiple sources.

We might have tables of values which are typed in by somebody or generated by a device and we have to import them in a spreadsheet. And then if we want to manipulate them by using another program, we might want to export them from a spreadsheet this is typically done using text files in which the columns of the spreadsheet are stored in a systematic way separated by say commas. So, this also involves text processing.

And finally, most of us spend a time using a computer actually working with the internet. One of the most common things we do when we use the internet is to type queries and look for matching documents or other resources on the internet. So, most of this search query processing currently is done using text. It matches the text in the queries that we give with some information about the documents also implicitly in text and decides which documents are most relevant to our query. So, text processing is an important part of computation in general. And the ease in which you can manipulate text in python is one of the reasons why it has become a very popular language to program many things including internet applications.

(Refer Slide Time: 03:18)



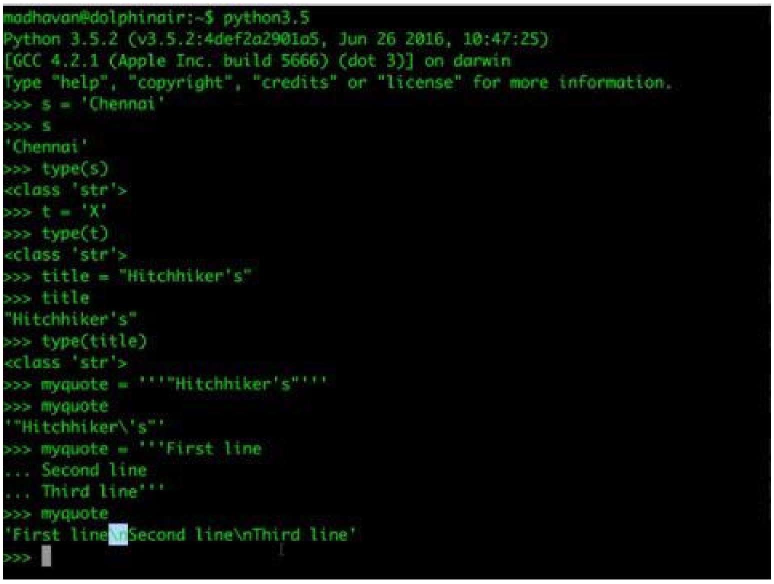

Python uses the type string for text, which internally is called str. So, we will use the word string instead of str, because it is easier to say. So, a string is basically sequence of characters. Unlike other programming languages, python does not have a specific character type to distinguish a single character from a string of length 1. So, there is only one type for text in python, which is string, and a single character is indistinguishable from a string of length 1. So, there are not two types of things; it is not that we have single characters and then string is a sequence of characters, a string is sequence of symbols and one symbol is just a sequence of length 1.

The values of this type are written as we would normally do in English using quotes. We use quotation marks to demarcate the beginning and at the end of a string when we want to write down an explicit value. So, we can use any type of quote, so a single quote would denote in this case the name city is assigned the string ‘Chennai’. Note that when we write symbols like this capital C is different from small c and so on. So, we have seen exactly seen the symbols within these two quotes as the value assigned to the string to the name city.

Now we can also use double quotes; and one reason to use double quotes is if you actually need to use a single quote as part of the string. This is one way to do it; and the other way to do it is actually to write a back slash. If you write a back slash and s quote in the middle of the string, it means that this quote is to be taken as a symbol and not at the end of the string, but a much simpler way to include special things like quotes inside other quotes is to change the quotation. So, a single quote can include double quotes, and the double quote can include single quote without any confusion. So, this says that the name title is assigned a value “Hitchhiker’s Guide to the Galaxy”.

Now, what if you wanted to combine both double quotes and single quotes in a string? So, python has a very convenient thing called a triple quote. So, you can open three single quotes, and then you can write whatever you want with multiple double quotes and single quotes. So, if you want to say ‘‘‘He said his favorite book is within quotes “Hitchhiker’s Guide to the Galaxy” ’”. Then this value string has both double quotes inside it and it also has a single quote inside it. So, we cannot enclose it in double quotes and we cannot enclose it in single quotes, because either of them will be ambiguous unless we use this back slash as I said before. So, if we do not want to use back slash, you can use a triple quote.

(Refer Slide Time: 06:06)

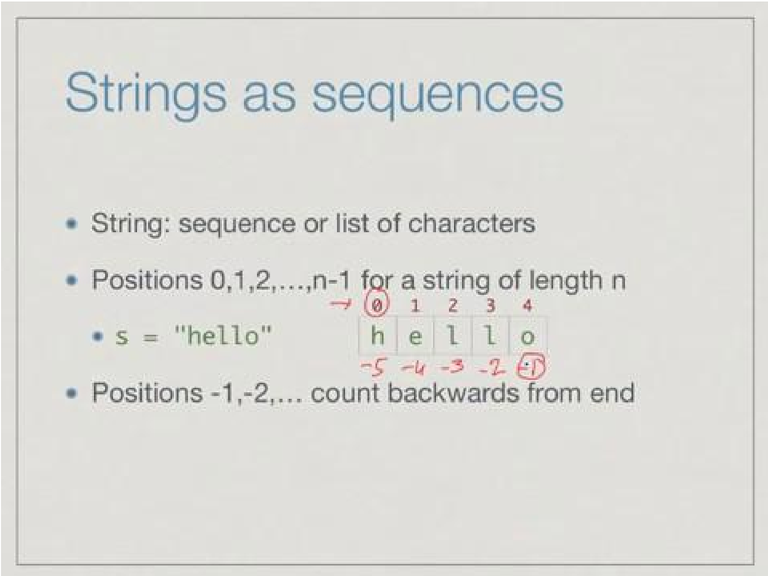



Let see how this works in python interpreter. So, we can say s equal to ‘Chennai’ and now been asked the value of this and we see that it is reported with single quote. If we ask for the type of s, it says that s is class of str. So, this tells us that internally python realizes that s is a string. If we say t is equal to say just the letter x, then the type of t is also a string. So, there is no distinction between single character and multiple characters. Now if we say let us just shorten it say title is equal to “Hitchhiker’s” then if you ask for the value of title, it shows it to you with double quotes outside and a single quote outside. So, this indicates that this is a single string and again the type of title is str.

And finally, if I say myquote is equal to and I use three quotes and I use ‘‘‘ “Hitchhiker’s” ’’’. So, I have “Hitchhiker’s” in double quotes and Hitchhiker’s itself contains a single quote. And I use triple quotes around it then my quote is correctly shown. Now notice that when it displays my quote, it does not show triple quotes. It includes puts another single quote outside and it shows this internal single quote has been highlighted with the back slash. So, back slash single quote is python's way and many programming languages’ way of saying that the next character should not be treated as what it stands for, but as it is. So, just take the next single quote as a single quote, do not treat as the end of the quotation.

The other thing that you can do with single quote is to actually write multiple lines. So, I do this first line, and then second line, and then third line, and then close the quote then my quote is shown as first line with back slash n. So, back slash again is a special character which indicates a new line; then second line, then new line, and then third line. We said before that python is very useful for manipulating text and one other thing that you would like to do is actually read and say a paragraph of text or multiple lines of a document and not have to worry about the fact that these are multiple lines just store it as a text value as a string. This is very much possible in python you can embed multiple lines of text into a single value.

(Refer Slide Time: 09:00)

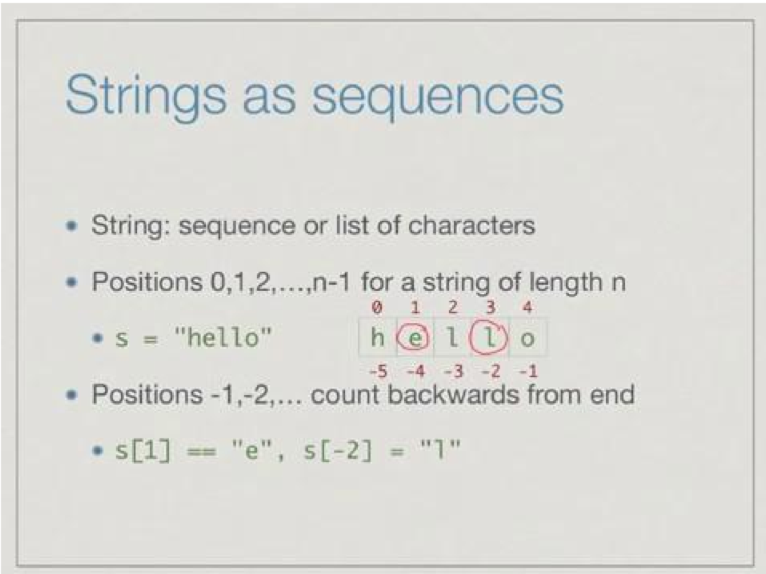



As we said the string is a sequence or a list of characters. So, how do we get to individual characters in this list? Well, these characters have positions and in python positions in a string start with 0. So, if I have n characters in a string, the positions are named 0 to n minus 1. So, supposing we have a string hello, it has 5 characters. So, the positions in the string will be called 0, 1, 2, 3 and 4; so this is how we label positions.

And another convenience in python is that we can actually label it backwards. We can say that this is position minus 1; very often you want to say take the last character of a string and do something. So, instead of having to remember the length and then go to the end, it is convenient to say take the last character. So, take the minus 1th character. So, we actually saw this and we did the gcd, we talked about the last element of the list say the list of common characters, and we said the minus 1th element n the list is the last element.

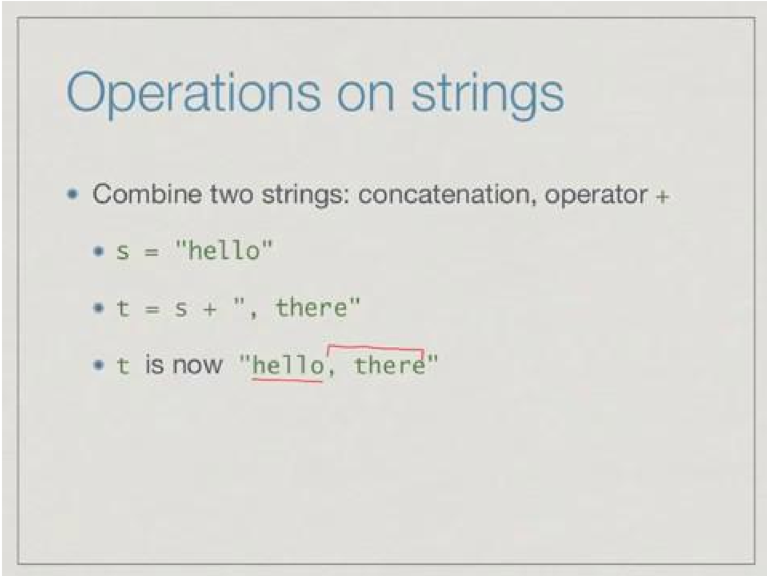
This numbering scheme that we use for list informally in the gcd example without formally explaining, it is actually the same numbering scheme that is used for positions in the string. We have minus 1, minus 2, minus 3, minus 4, minus 5, so the important thing to remember is that going forward, you start at 0, and coming backward you start at minus 1, because obviously, minus 0 is same as 0. So, if we use minus 0 for the right most thing there would be terrible confusion as to whether we are talking about the first value or the last value. So, the forward position start from 0 from the beginning and the reverse position start from minus 1 from the last element.

(Refer Slide Time: 10:37)




Once we have this then we can see that we use this square bracket notation to extract individual positions. So, s 1, so that is the character at position 1 is an e and if I walk backwards then s minus 2 is an l.

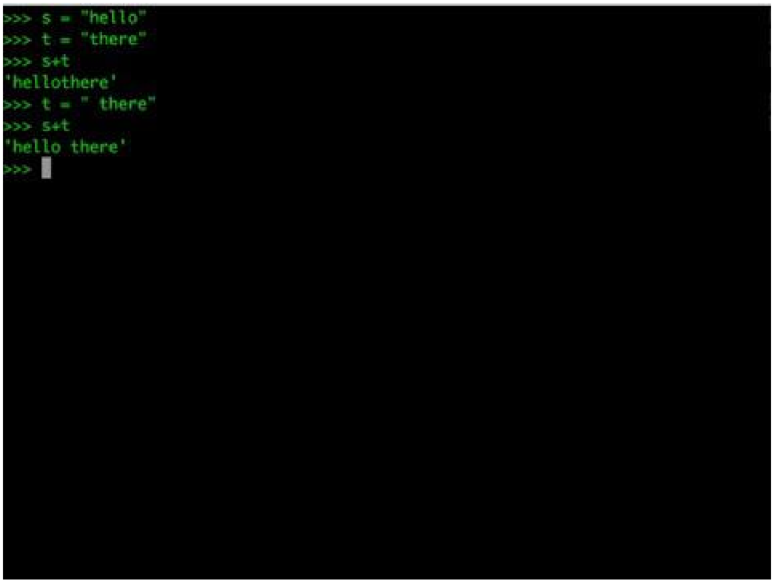
(Refer Slide Time: 10:59)




One of the most basic things one can do with strings is to put them together; to combine two values into a larger string and this is called concatenation; putting them one after the other. And the operator that is used for this is plus. So, plus, we saw for numeric values add them; for strings the same symbol plus does not add strings; obviously, it does not make sense to add strings, but it juxtaposes them, puts them one after the other.

So, if we have a string hello as we did before, and we take this, and we take a new string and we add it to s. Then we get a string t, whose value is the part that was in hello plus the part that was added. So, plus is just the simple operator which takes two strings and sticks them side by side.

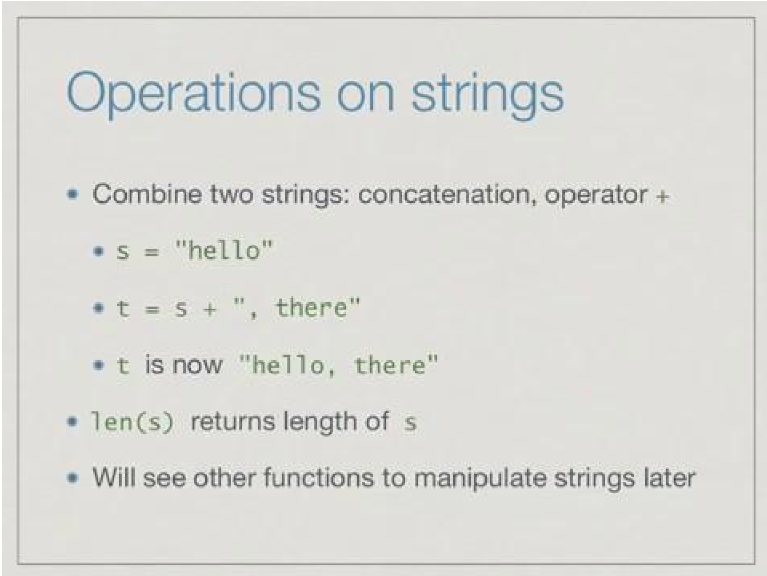
(Refer Slide Time: 11:52)




Let us look at an example in the interpreter. Just to emphasize one point; supposing I said s was hello and t was there, then s plus t would be the value hello there. Now notice that there is no space. So, plus literally puts s followed by t, it does not introduce punctuation, any separation, any space and this is as you would like it. If you want to put a comma or a space you must do that, so if you say t instead of that was space there t is the string consisting of blank space followed by there, now if I say s plus t, I get a space between hello and there.

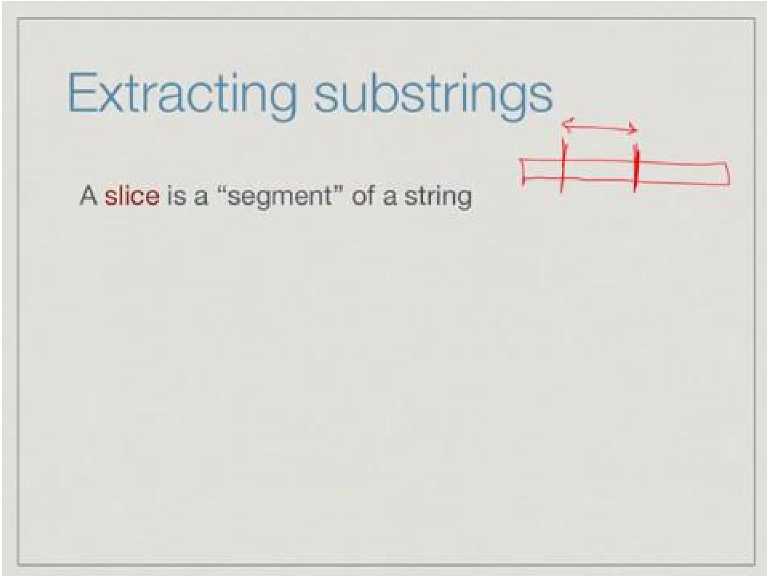
This is important to note that plus directly puts things together it does not add any punctuation or any separation between the two values. So, it is as though you have one new string which is composed of many old strings whose boundaries disappear completely.

(Refer Slide Time: 12:47)



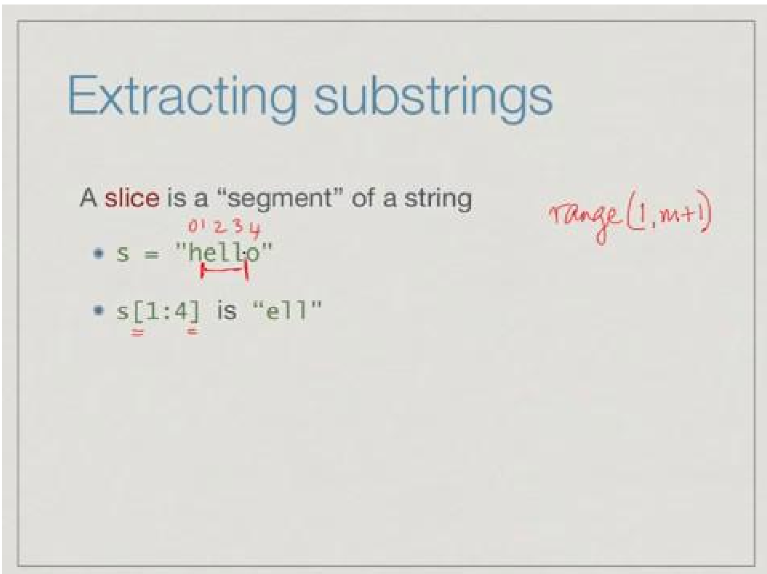

We can get length of the string using the function len. So, len(s) returns the length of s. So, this is the number of characters. So, remember that if the number of characters is n then the positions are 0 to n minus 1. So, the length of the string s here would be 5, the length of the string t here would be 5 plus 7 – 12. There are many other interesting functions that one can use to manipulate strings, you can search and replace things, you can find the first occurrence of something and so on, and we will see some of these later on, when we get into strings and text processing and reading data from files in more details.

(Refer Slide Time: 13:26)




A very common thing that we want to do with strings is to extract the part of a string. We might want to extract the beginning, the first word and things like that. The most simple way to do this in python is to take what is called a slice. Slice is a segment, a segment means I take a long string which I can think of as a list of character and I want the portion from some starting point to some ending point.

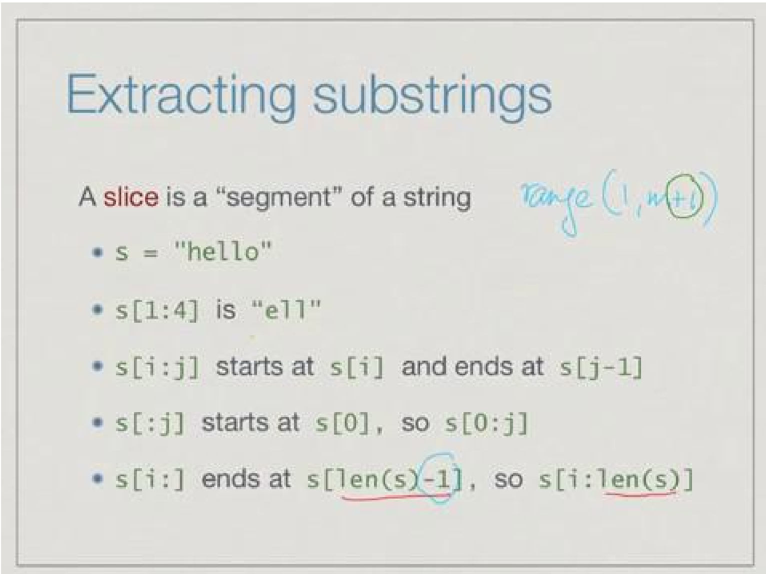
(Refer Slide Time: 13:55)



This is what python calls a slice. So, if we say s is hello as before, then for a slice we give this starting point and the ending point separated by colon. So, we use this square bracket notation exactly as though we were extracting part of a string, but the part that we are extracting is not the single position, but a range of positions from 1 to 4.

Now in python, we saw that we had this range function which we wrote last time, it said things like, if I want the numbers from 1 to m, I must write 1 to m plus 1, because the range function in python stops one position short of the last element of the range. So, in the same way, a slice stops one position short of the last index in the slice. So, if I do this then remember that hello has position 0, 1, 2, 3, 4, so the slice from 1 to 4 starts at 1 goes to 2, goes to 3, but does not go to 4, so it is only from e to l - the second l.

(Refer Slide Time: 14:59)

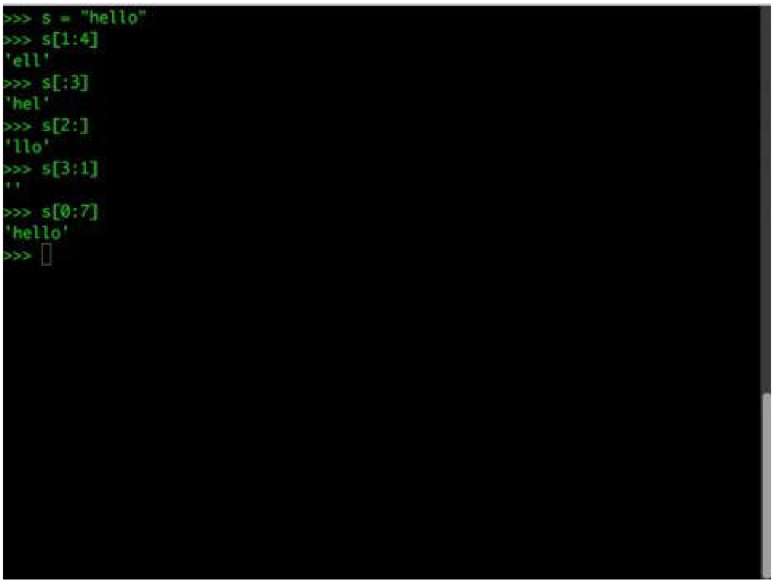



In general, if I write s i colon j then it starts at s i and ends at s j minus 1. There are some shortcuts which are easy to remember and use; very often you want to take the first n characters in the string, then you could omit the 0, and just say start implicitly from 0, so just leave it out, so just start say colon and j. So this will give us all position 0 1 up to j minus 1. So, if I leave out first position, it is implicitly starting from 0.

Similarly, if I leave out the last position it runs to the end of the string. So, if I want everything from i onwards then I can say s i colon and this will go up to the position length of s minus 1, but if I write explicitly as a slice, I will only write length of s. So, essentially this is the main reason that python has this convention that whenever I write something like a range of 1 to m plus 1 then I have this extra plus 1 here. So, the main reason for this plus 1 here is to avoid having to write minus 1.

If I had to include the last character and if I start numbering at 0, then every time I wanted to go to the end of the string I would have to say length of s minus 1. It is much more convenient to just say length of s, and implicitly assume that it knows that it should not go to length of s, but length of s minus 1. So, this whole confusion if you would like to call that in python about that fact that all ranges end one short of the right hand side of the range, stems from the fact that you very often want to run from something to the length of it in a list or a sequence or a string and when you say that you do not want have keep remembering to say minus 1.

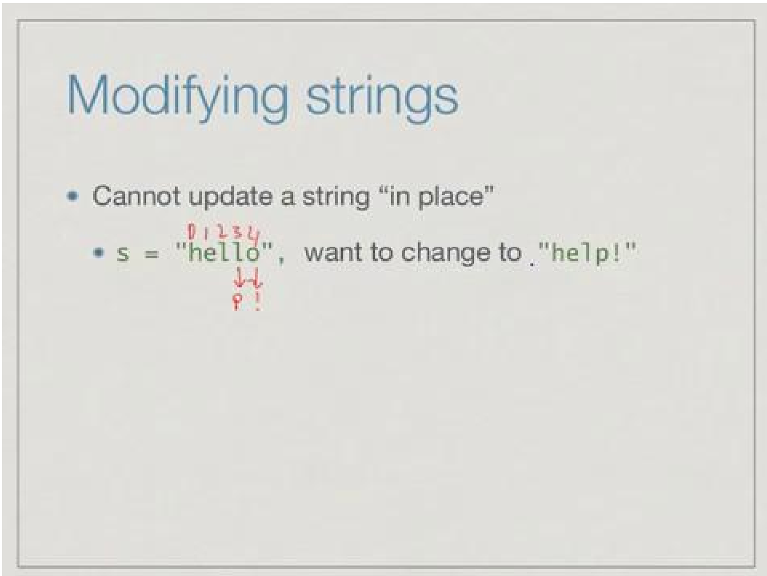
(Refer Slide Time: 16:45)




Let us play with the second in the python interpreter. So, if I say s is equal to hello then we saw that if I do 1 to 4, I get 'ell'. If I say colon 3 then I get 'hel' that is 0 1 2. If I say 2 colon, I get 'llo' that is 2 3 4. What if I say 3 2 1, so this says: start at position 3 and go up to position 1 minus 1 which is 0. So, python does not give you an error, it takes all these invalid ranges, anything where for example, the starting point to the ending point does not define a valid range, and it says this is the empty string.

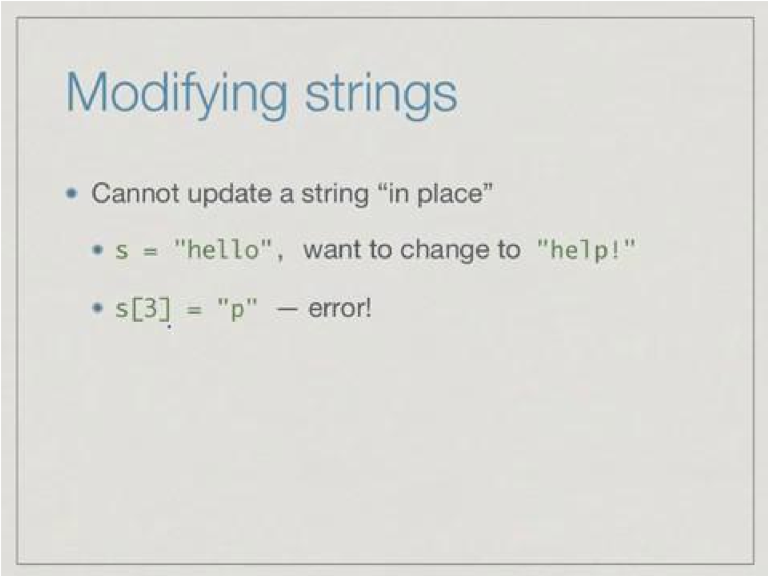
On the other hand, if I say something like go from 0 to 7, where there is no 7th position in the string, here python will not give an error instead, it will just go up to the last position which actually exists in the string below 7. So, in general these range values are treated in a sensible way, if you give values which do not make sense. As far as possible python tries to do something sensible with the slice definition.

(Refer Slide Time: 17:57)



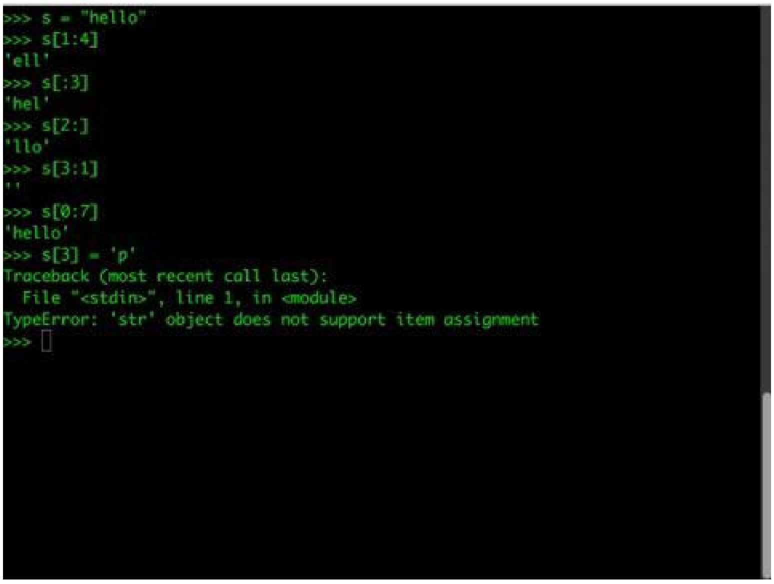

Though we have access to individual positions or individual slices or sequences within a string, we cannot take a part of a string and change it as it stands. So, we cannot update a string in place. Suppose, we want to take our string “hello” and change it to the string “help!” it would be nice if we could take the third and the fourth position. So, remember 0, 1, 2, 3, 4, 5, so 0, 1, 2, 3, 4, so it would be nice if we could say make this into a p and make this into an exclamation mark, so that I could get help instead of hello.

(Refer Slide Time: 18:36)



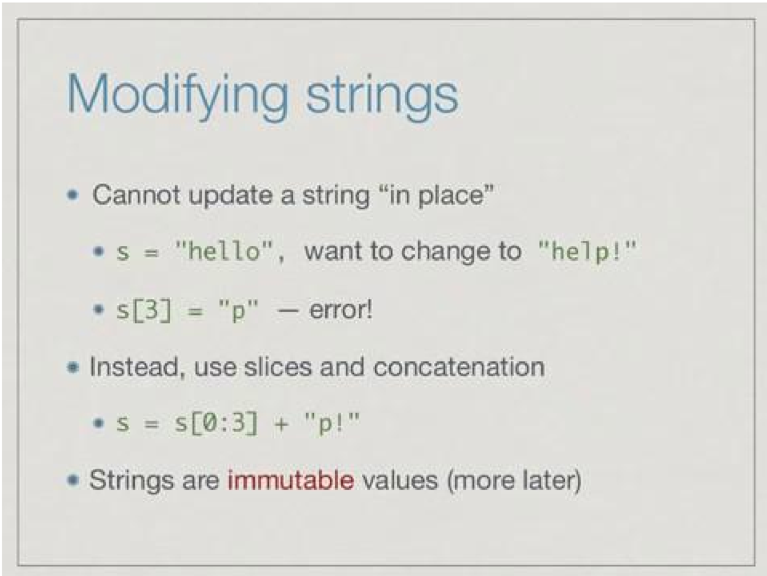

We would want to write something like change s 3, assign the value s 3 to be the string p. Now, unfortunately python does not allow this. So, you cannot update a string in place by changing its part. In fact, if you try this, you will actually get an error message, let us see.

(Refer Slide Time: 18:54)



Here we have the string hello defined in four, and if I now try to say s 3 is equal to p, then it says this does not support item assignment, which is what we are trying to say you cannot change parts of a string as it stands.

(Refer Slide Time: 19:12)

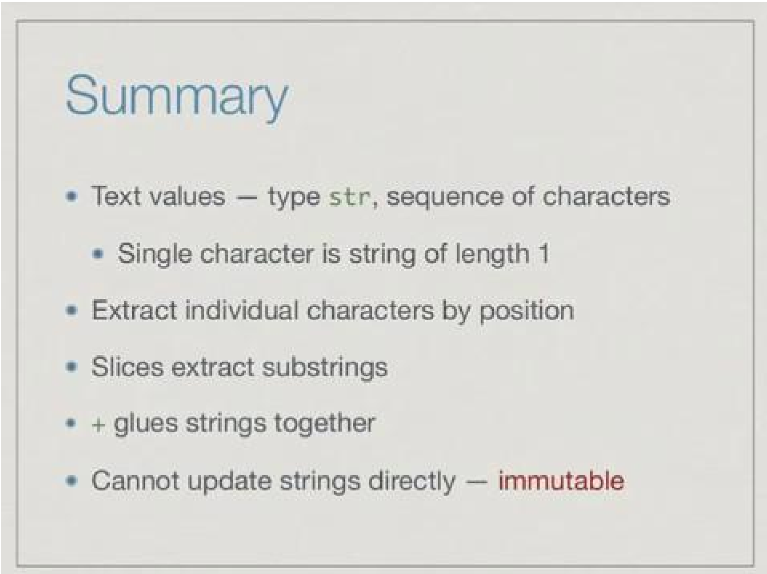



Instead of doing this, instead of trying to take a string and change the part of it as its stands what you need to do is actually construct a new string effectively using the notion of slices and concatenation. Here what we want to do is we want to take the first part of the string as it is. These are the first three characters, and then we want to change this to p exclamation mark. So, what we can say is update s by taking 0, 1, 2 which is slice 0 to 3 and concatenating it with the new string p exclamation mark. So, this is how you modify strings in python, but important thing is this is a new s we are not claiming that this s is same as old s.

There we build a new string from the old string and perhaps store it back in the same name, it is partly like when we say j is equal to j plus 5, we are actually saying that we have created a new value for j and stored it back in j.

Here again we are creating a new string and putting it back, but we are not modifying it. Now this distinction between modifying and creating a new value may not seem very important at this moment, but it will become important as we go along. So, strings are what are called immutable values, you cannot change them without actually creating a fresh value; whereas, lists as we will see which are more general type of sequence can be changed in place you can take one part of a list and then replace it by something else. So, we will see more about this later, this is a fairly important concept. Remember for now that strings cannot be changed in place.

(Refer Slide Time: 20:42)




To summarize what we have seen is that text values are important for computation, and python has the types - string or str, which is a sequence of characters to denote text values. And there is no distinction for a separate type for a single character; there is no single character type in python, a single character is just a string of a length 1.

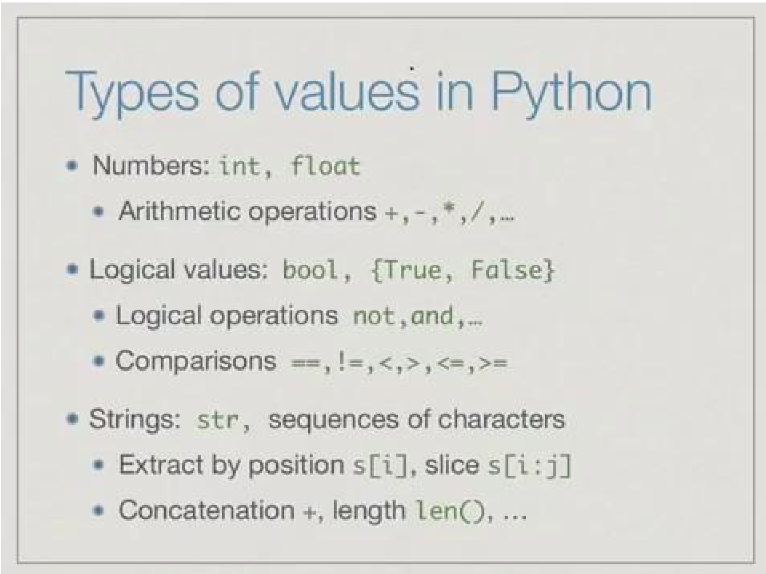
We can extract individual characters by index positions, we can use slices to extract sub strings, and we can glue strings together using the concatenation operator plus, but strings are immutable. We cannot take a value assigned to a string name and update it in place. We can create a new value by manipulating it using slices and concatenation, but we cannot directly update it, because strings are immutable.

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

**Week - 02**
**Lecture - 03**
**Lists**

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

(Refer Slide Time: 00:02)

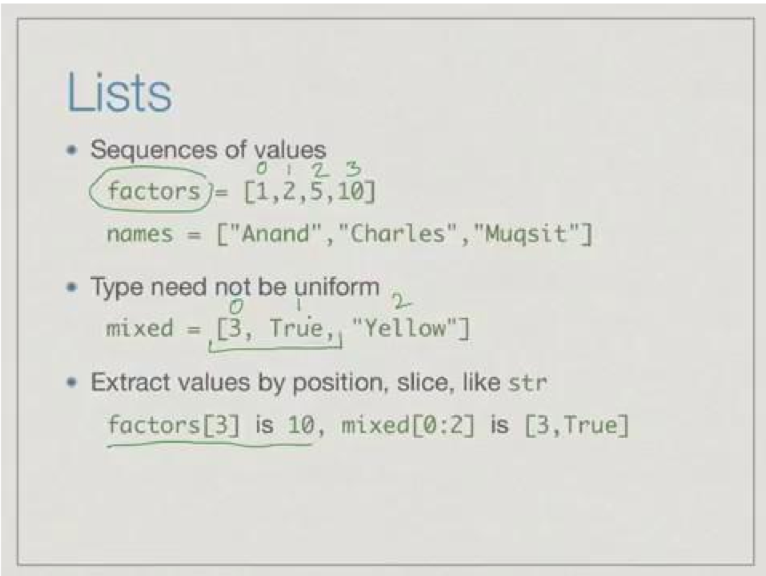



You began with the numeric types, which divided into two categories int and float. So, int represented whole numbers or integers, and float represented values which have a decimal point. And for these, we had arithmetic operations such as plus, minus, times, divide and also other functions which we can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

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

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

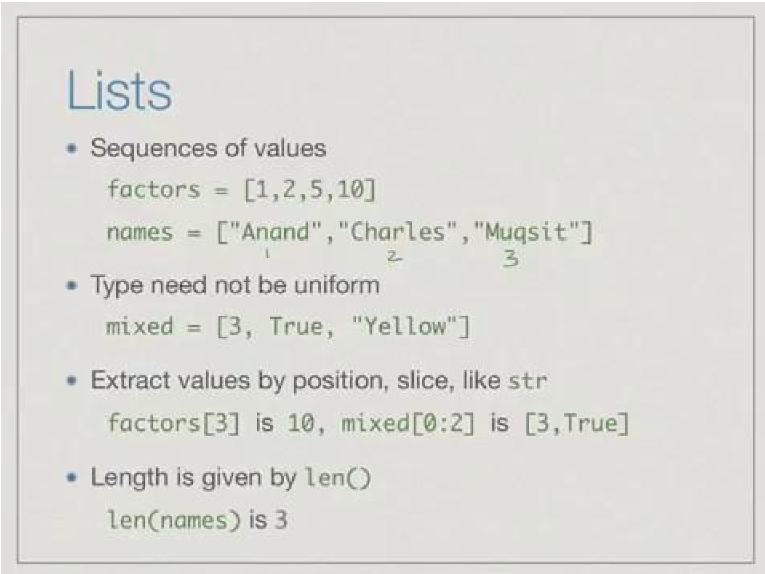
(Refer Slide Time: 01:59)




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

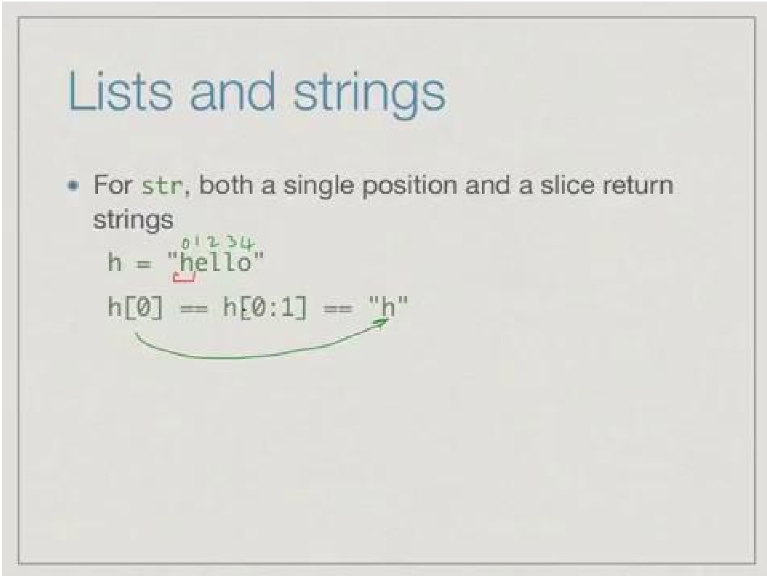
A list is a sequence in the same way as a string is and it has positions 0 to n minus 1 where n is the length of the list. So, we can now extract values at a given position or we can extract slices. In this example if we take the list factors and we look at the third position remember that the positions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

(Refer Slide Time: 03:23)




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

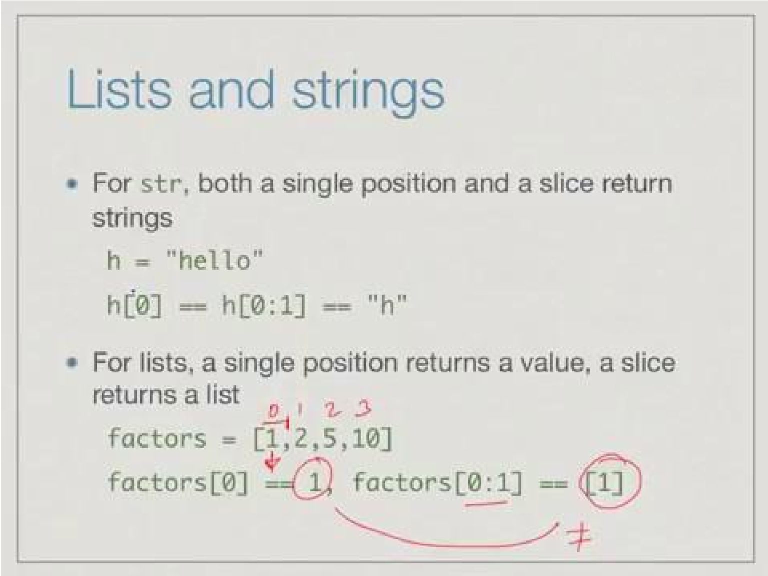
(Refer Slide Time: 03:43)




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

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

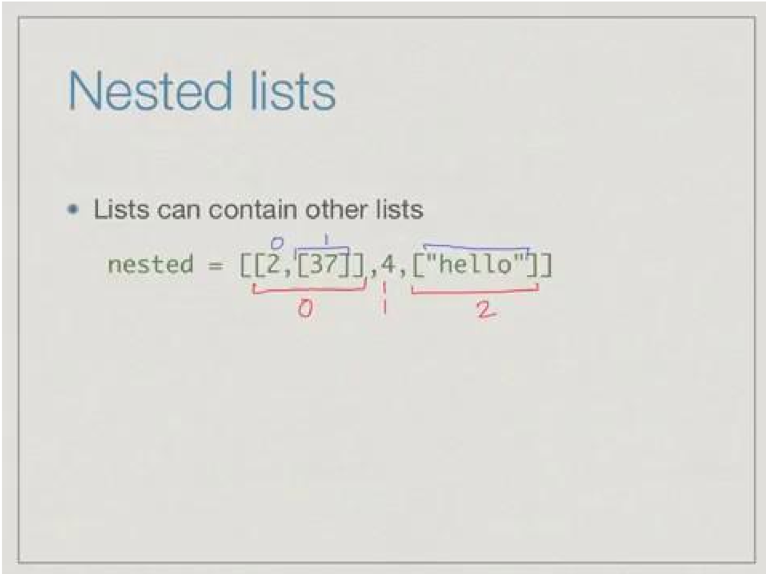
(Refer Slide Time: 04:39)




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

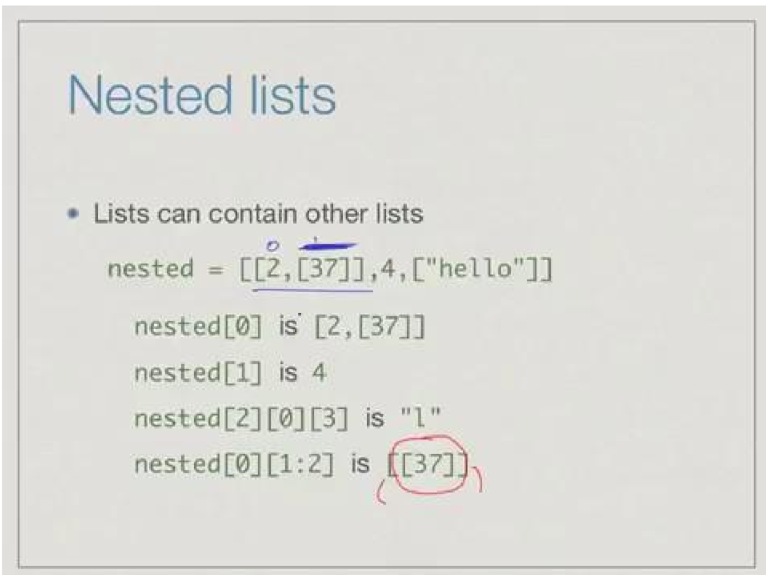
Just remember this that in a string we cannot distinguish between a single value at a position and a slice of length one. They give us exactly the same type of value and the same value itself. Whereas, in a list a slice of length one is a list whereas, a value at a position is a single value at that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

(Refer Slide Time: 05:39)




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

(Refer Slide Time: 06:33)

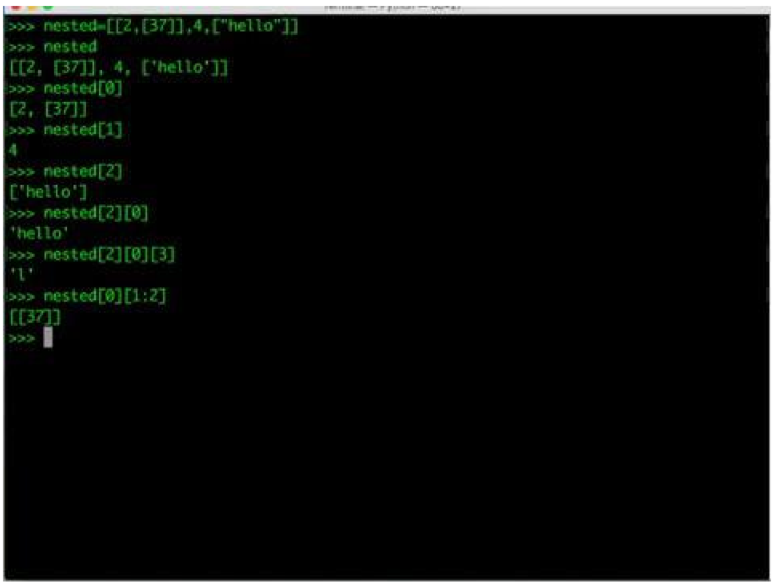


If we look at this example then we can see that if we want to look at the 0th position in nested then as we said we get this value and this value consist of a list itself containing 2 and the list containing 37. On the other hand, if we ask for the first position number 1 then we get the value 4. And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

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

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

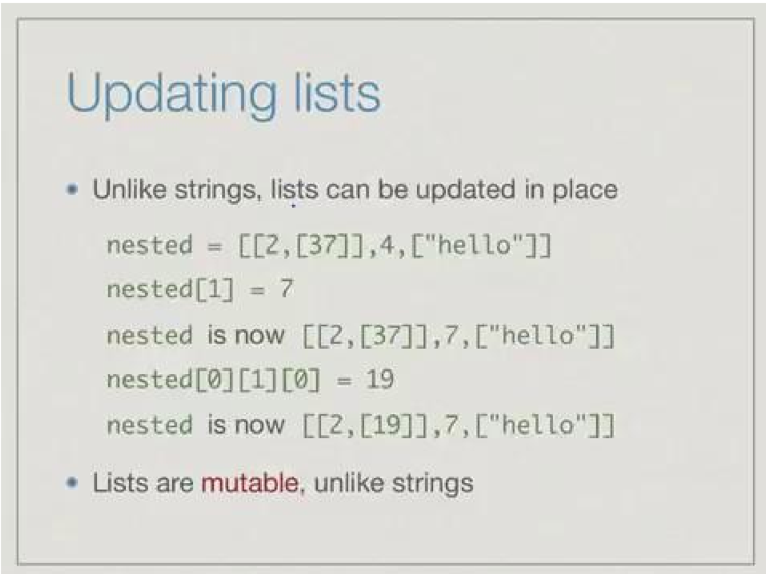
(Refer Slide Time: 08:18)



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

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

(Refer Slide Time: 09:25)

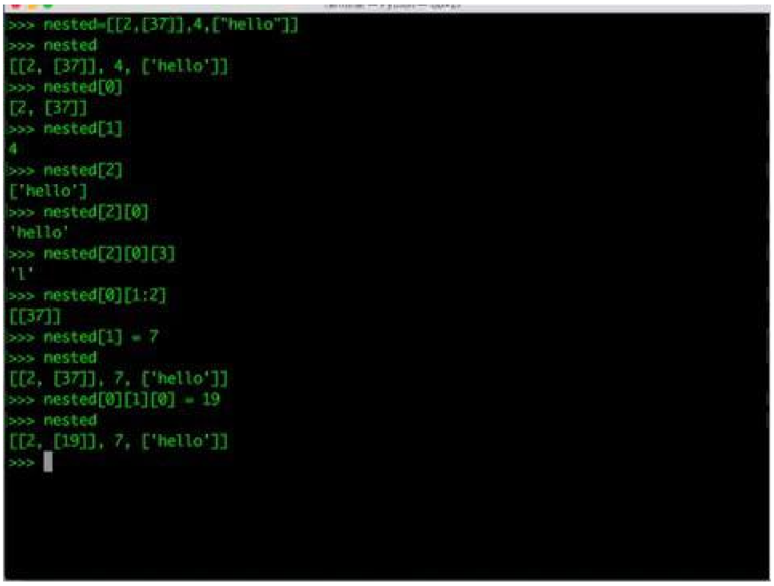



One fundamental difference between list and string or indeed any of the values we have seen before is that a list can be updated in place. So, if we have a list nested as we had before and we say nested of 1 is 7. Remember when we try to change a position in a string we got an error. We cannot change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

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

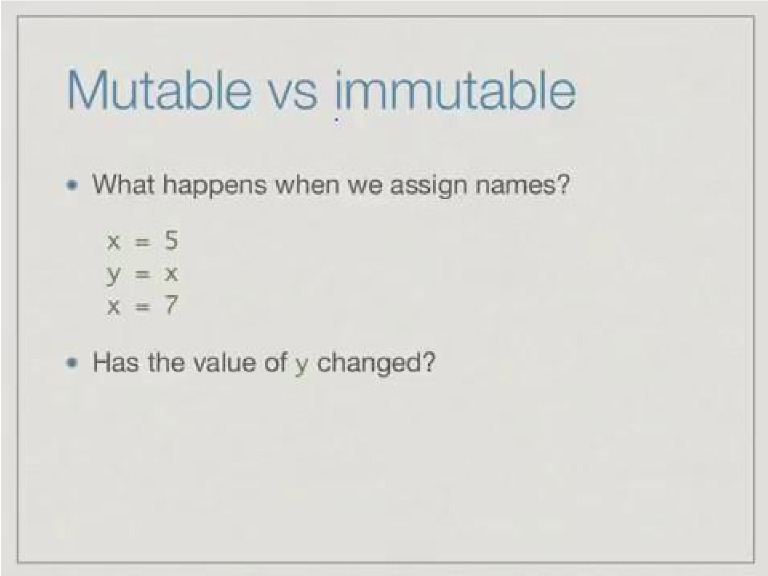
A list can be transformed in place we can take a list and change its structure unlike a string if we try to change a string we have to actually construct a new string and re assign the name, but in a list with the same name we can update parts of it without affecting the other parts.

(Refer Slide Time: 11:03)




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

(Refer Slide Time: 11:45)



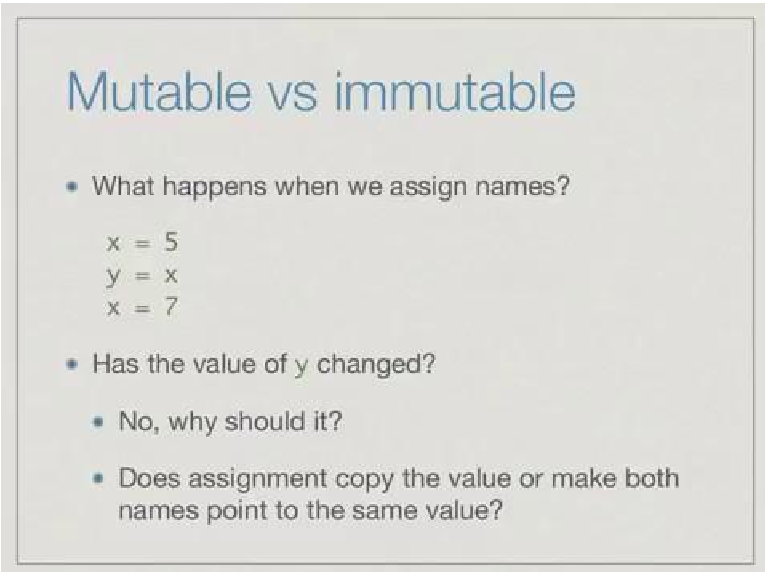

It is important to understand the distinction between mutable and immutable values, because this plays an important role in assignment. And as you will later see, it also plays a major role in what happens when we pass values to functions as arguments. Let us look at what happens when we assign names.

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

(Refer Slide Time: 12:38)

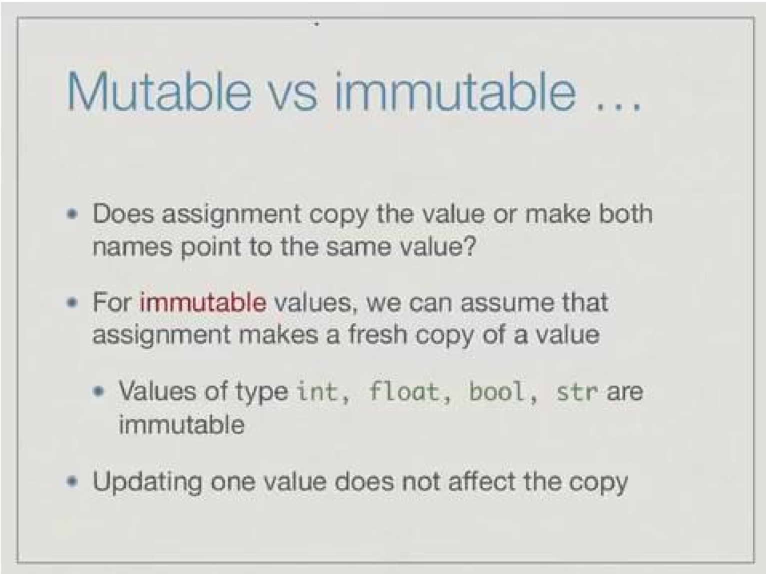

Let us start with x equal to 5, y equal to x. So, if we ask for the value of y at this point it is 5 as we expect. Now we change x to seven the question is it is y 5 or y 7 and indeed y is still 5 and this is perfectly natural as far as our understanding goes that what we did, when we set the value of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

(Refer Slide Time: 13:07)




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

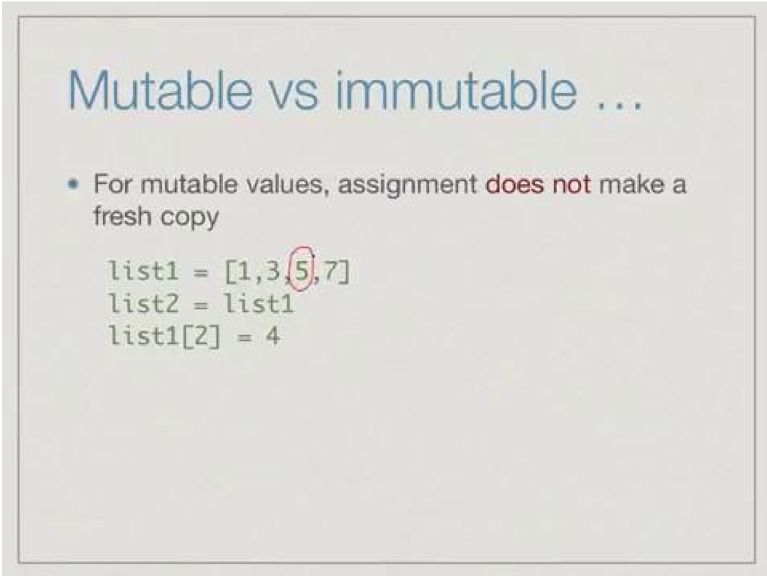
(Refer Slide Time: 13:42)




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

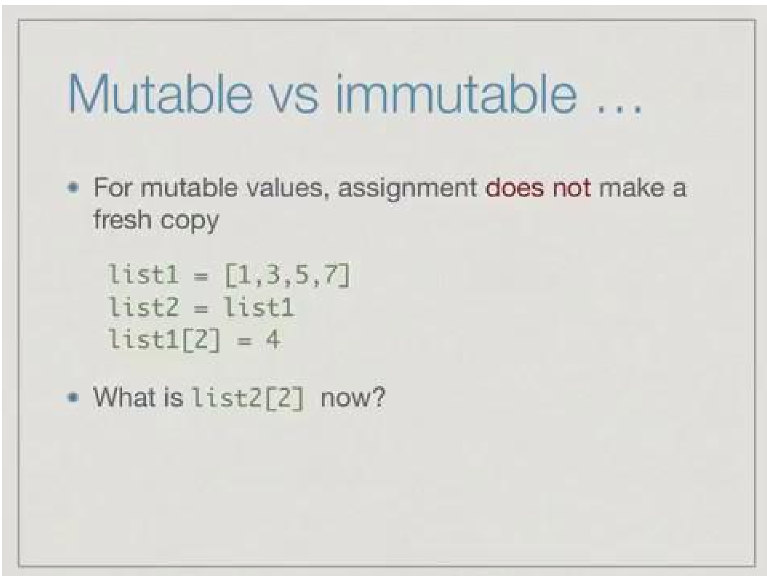
This applies to all the types we have seen before today’s lecture namely int float bool and string these are all immutable. If we do the kind of assignment we did before where we assign something to x then make y is the same value was x and then update x, y will not change. Updating one value does not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

(Refer Slide Time: 14:30)



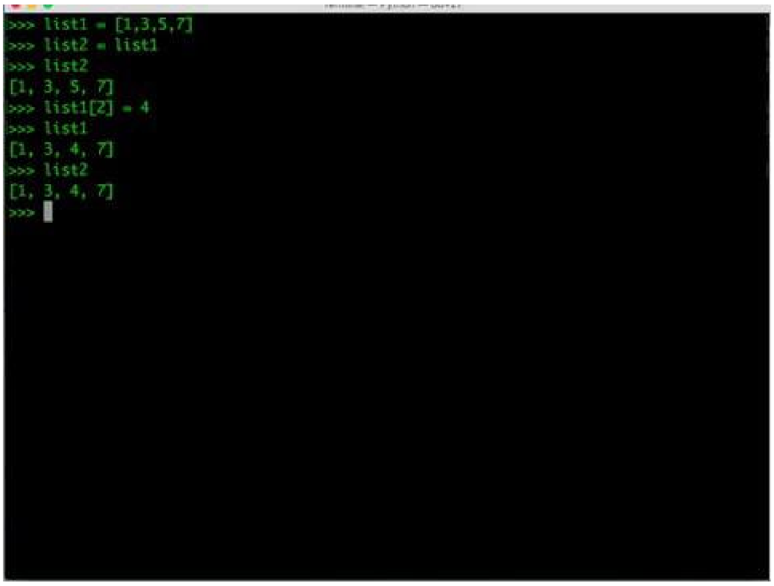

However as we are pointed out lists are difference beast from strings and list are mutable. It turns out that for mutable values assignment does not make a fresh copy. Let us look at the following example we first assign say the list 1, 3, 5,7 to the name list1, then we say that list2 is the same as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

(Refer Slide Time: 15:08)



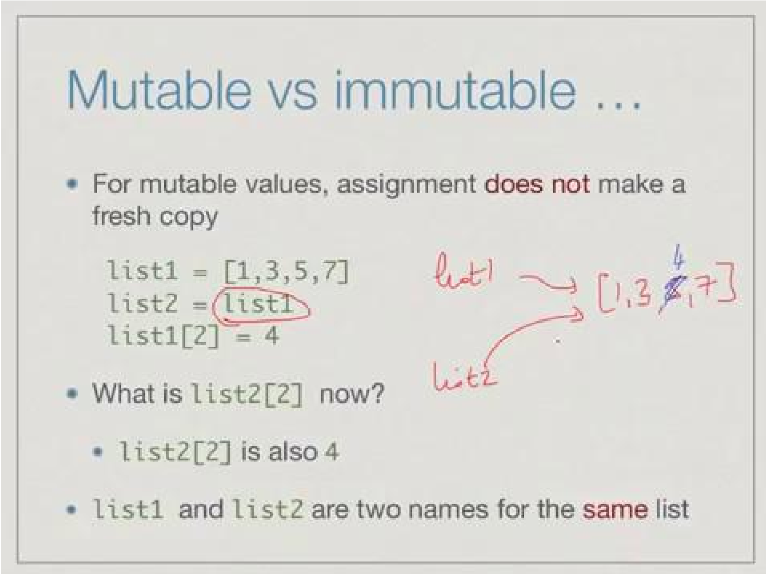
The question is what has happened to list2, is list2 the same as before namely 1, 3, 5, 7 or as list2 also become 1, 3, 4, 7 like list1. So, lets us see what happens in the interpreter.

(Refer Slide Time: 15:25)




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

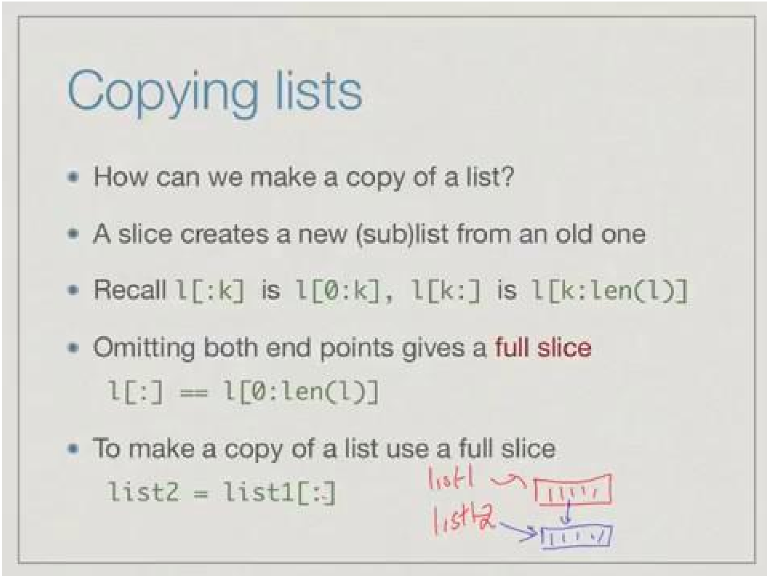
(Refer Slide Time: 16:05)




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

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

(Refer Slide Time: 17:10)




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

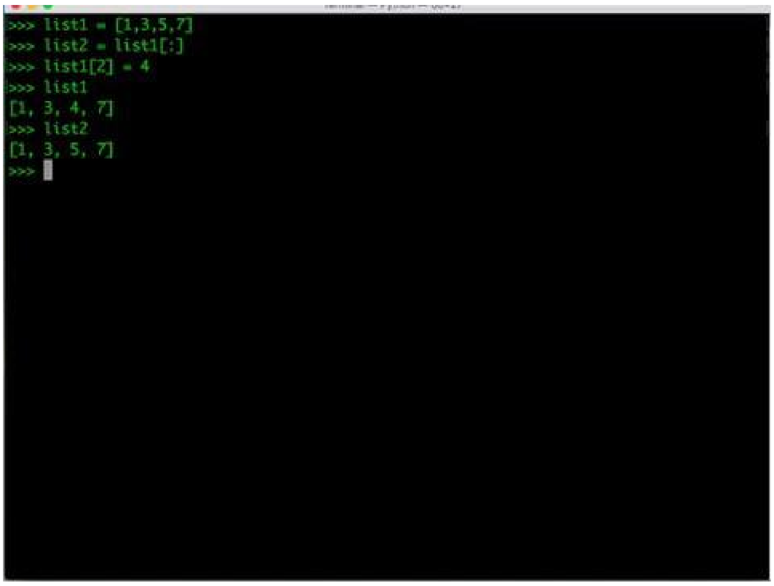
We also saw that when we looked at strings that we can leave out the first position or the last position when specifying a slice. If we leave out the first position as this then we will implicitly say that the first position is 0, so we start at the beginning. Similarly, if we leave out the last position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

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

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

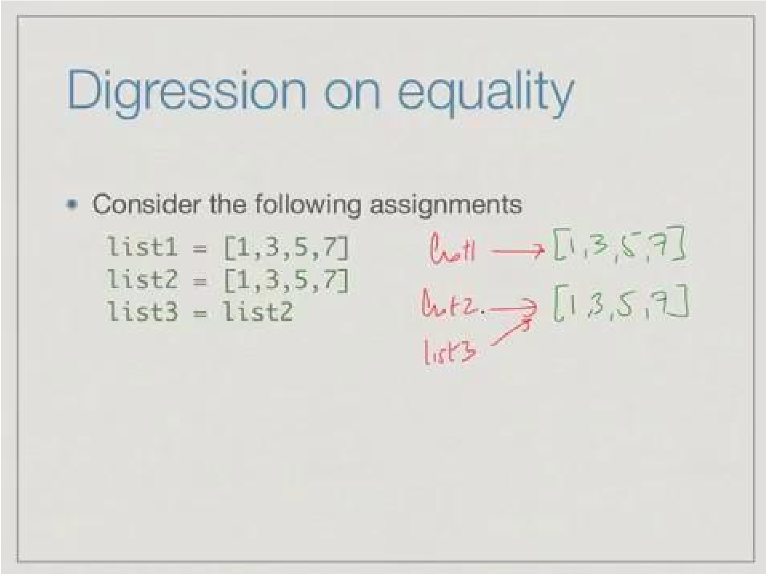
Remember if I do not have this then I will get list1 and list2 pointing to the same actual list. There will be only 1 list of values and will point to the same. But if I do there have this then the picture changes then what happens is that the slice operation produces a new list which has exactly the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

(Refer Slide Time: 19:45)




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

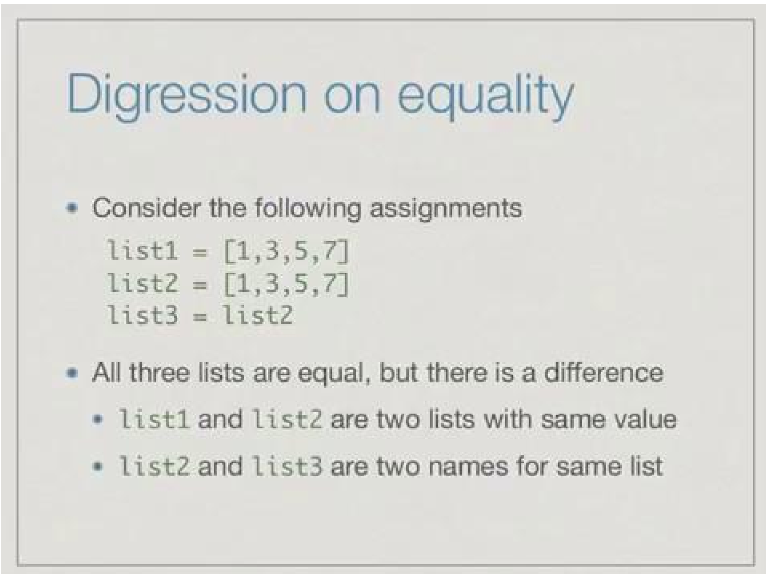
(Refer Slide Time: 20:19)




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

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

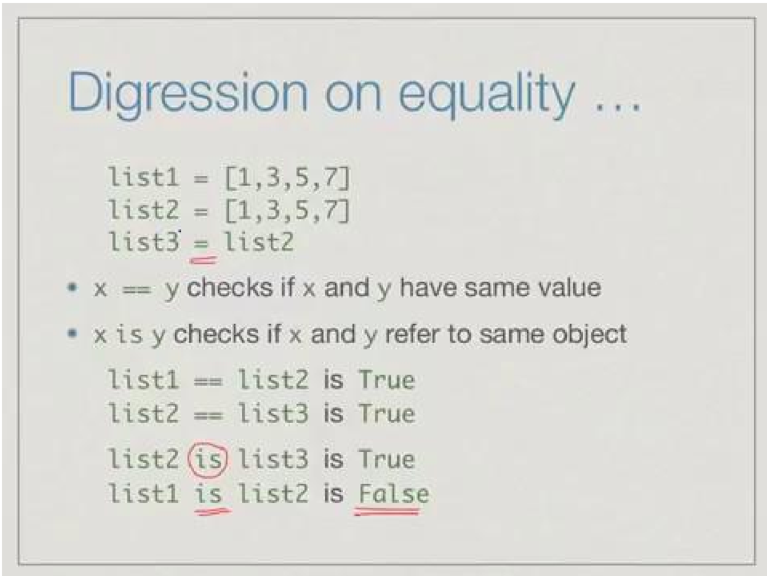
(Refer Slide Time: 21:10)



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

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

(Refer Slide Time: 21:57)



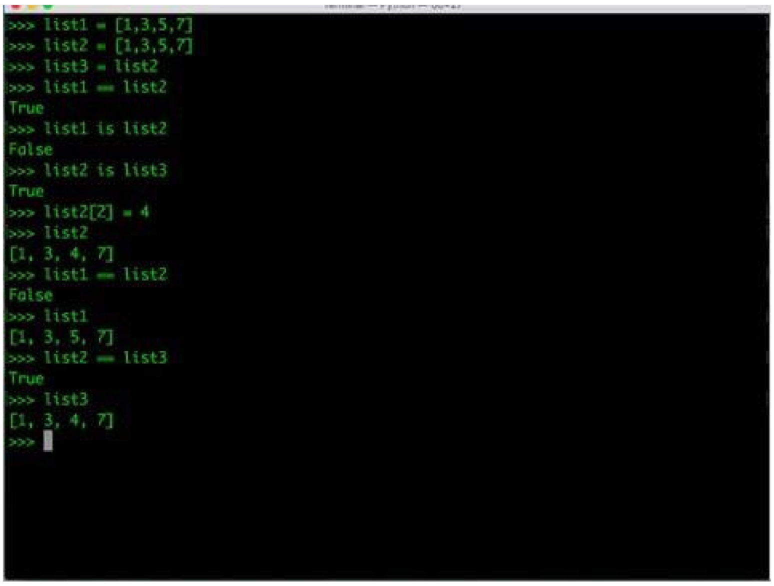

Python has we saw this operation equal to equal to, which is equivalent or the mathematically equality which checks if x and y as names have the same value. So, this will capture the fact that list1 is equal to list2 even though they are two different lists they happen to have the same value.

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

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

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

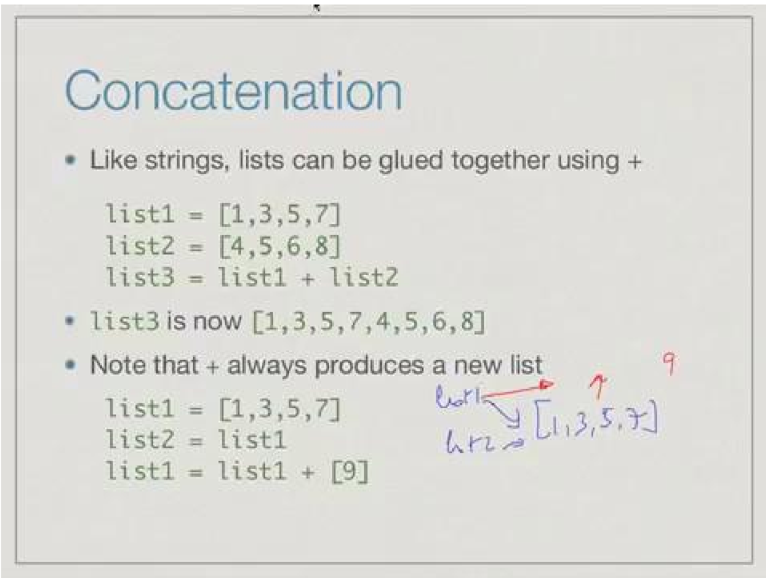
(Refer Slide Time: 23:41)




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

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

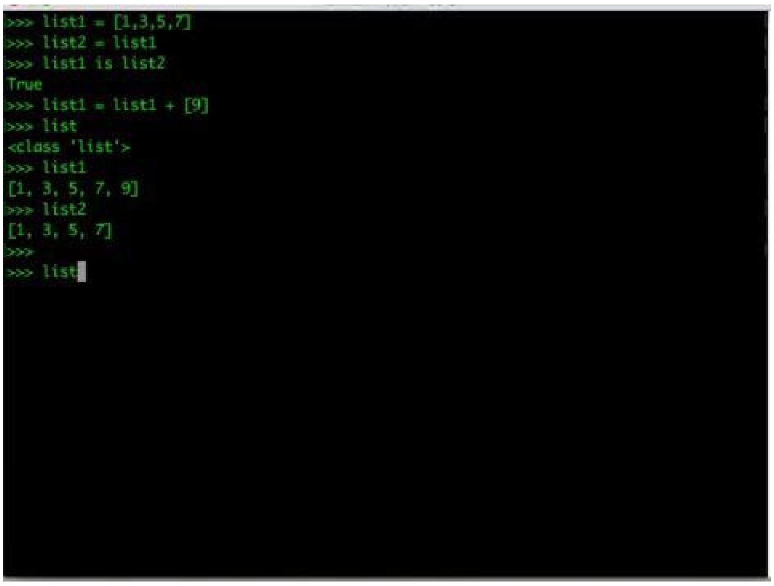
(Refer Slide Time: 24:54)




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

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

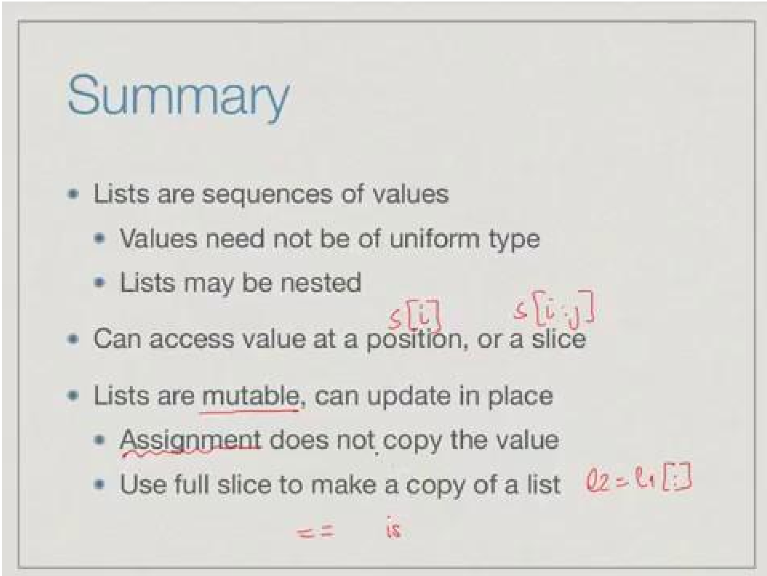
(Refer Slide Time: 26:00)




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

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

(Refer Slide Time: 26:49)




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

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

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

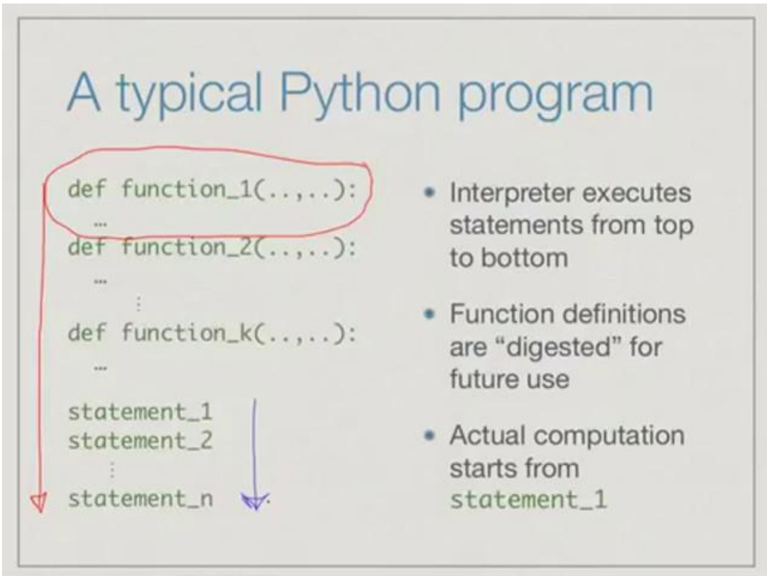
If we really want to make a copy, we use a full slice. So, we say l2 is equal to l1 colon with nothing before or after, this is implicitly from 0 to the length of l1, and this gives us a fresh list with exact same contents as l1. And finally, we saw that we can use equality and is as two different operators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

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

**Week - 02**
**Lecture - 04**
**Control Flow**

In the past few lectures, we have looked at values of different types in python, starting with the numbers and then moving onto the Booleans and then sequences such as strings and lists.

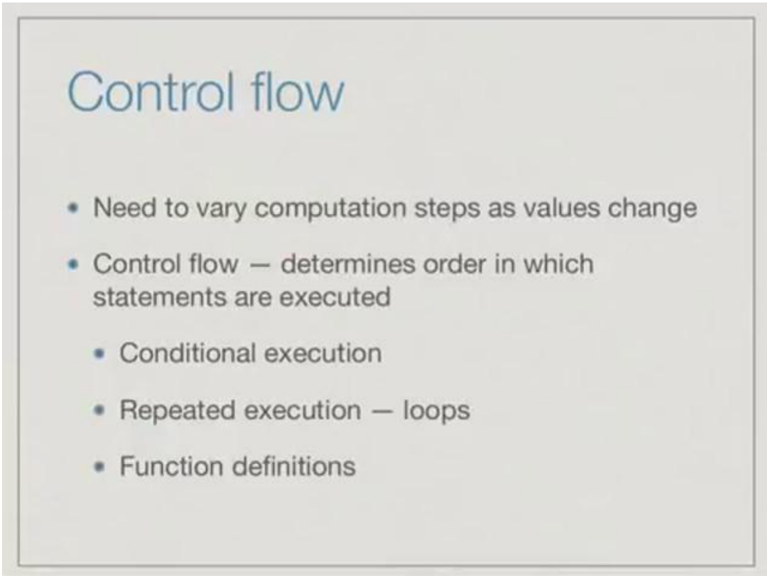
(Refer Slide Time: 00:02)




Now let us go back to the order in which statements are executed. Remember we said that a typical python program will have a collection of function definitions followed by a bunch of statements. In general, the python interpreter will read whatever we give it from top to bottom. So, when it sees the definition of a function, it will digest it but not execute it.

We will look at function definitions in more detail very soon. And when we now come to something which is not a definition then python will try to execute it, this in turn could involve invoking a function in which case the statements which define the function will be executed and so on. However, if we have this kind of a rigid straight-line execution of our program then we can only do limited things. This means we have an inflexible sequence of steps that we always follow regardless of what we want to do.

(Refer Slide Time: 01:13)



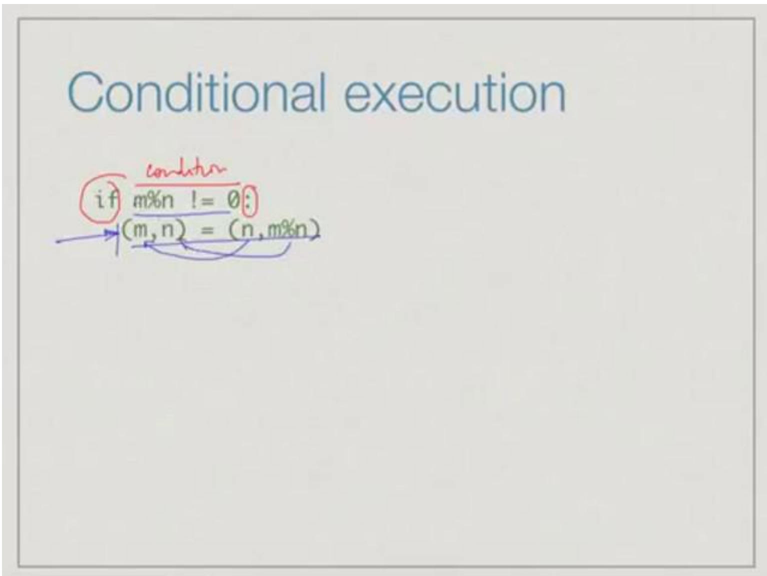

Now in many situations, many realistic situations, we need to vary the next step according to what has happened so far, according to the current values that we see. Let us look at a real life example. Supposing, you are packing your things to leave for the bus stop in the morning, or whether or not you take an umbrella with you will depend on whether you think it is going to rain that day. If you carry the umbrella all the time then your bag becomes heavy, if you do not carry the umbrella ever, then you risk the chance of being wet.

So, you would stop at this point and see, in whatever way by reading the weather forecast or looking out of the window is it likely to rain today? If it is likely to rain, ensure that the umbrellas in your bag, put it if it is not there, or leave it if it is already there. If it is not likely to rain, ensure the umbrella is not in the bag, if it is not there it is fine otherwise, take it out. So, this kind of execution which varies the order in which statements are executed is referred to in programming languages as control flow.

There are three fundamental things all of which we have see informally in the gcd case. One is what we just describe the conditional execution. The other is when we want to repeat something a fixed number of times and this number of times is known in advance. We want to carry 10 boxes from this room to that room, so 10 times we carry one box at a time from here to there. On the other hand, sometime we may want to repeat something where the number of repetitions is not known in advance.

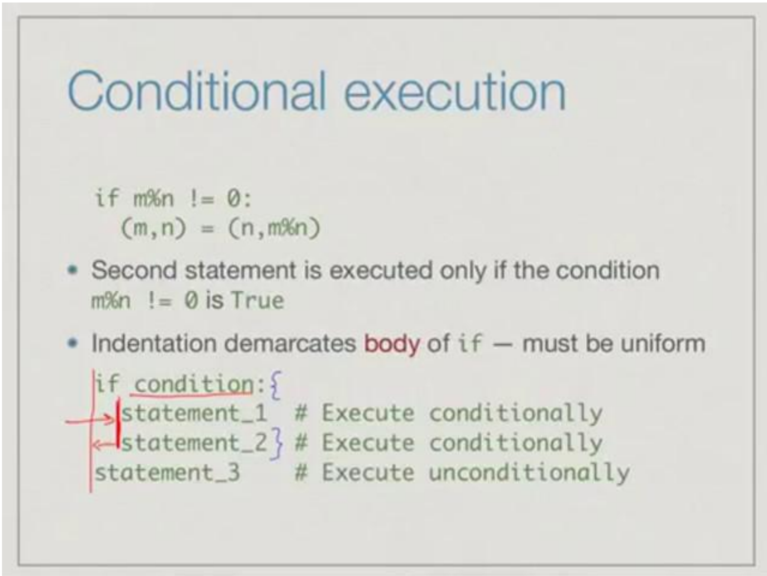
Suppose, we put sugar in our tea cup and we want to stir it till the sugar dissolves. So, what we will do, we stir it once check, if there is still sugar stir it again, check it there is still sugar and so on, and as the sugar dissolves finally after one round we will find there is no sugar at the bottom of the cup and we will stop stirring. Here, we will repeat the stirring action a number of times, but we will not know in advance whether we have to stir it twice or five times, we will stir until the sugar dissolves.

(Refer Slide Time: 03:17)




Let us begin with conditional execution. Conditional execution in Python is written as we saw using the 'if statement'. We have 'if', then we have a conditional expression which returns a value true or false typically a comparison, but it could also be invoking a function which returns true or false for example, and we have this colon which indicates the end of the condition. And then the statement to be executed conditionally is indented, so it is set off from the left so that Python knows that this statement is governed by this condition. We make this simultaneous assignment of m taking the old value of n, and n taking the value of m divided by n, only if m divided by n currently is not 0.

(Refer Slide Time: 04:03)




The second statement is executed conditionally, only if the condition evaluates to true and indentation demarcates the body of the 'if'. The body refers to those statements which are governed by the condition that we have just checked. So, let us look at a small kind of illustration of this. Suppose, we have code which looks as follows; we have if condition then we have two indented statements - statement 1 and statement 2, and then we have a third statement which is not indented.

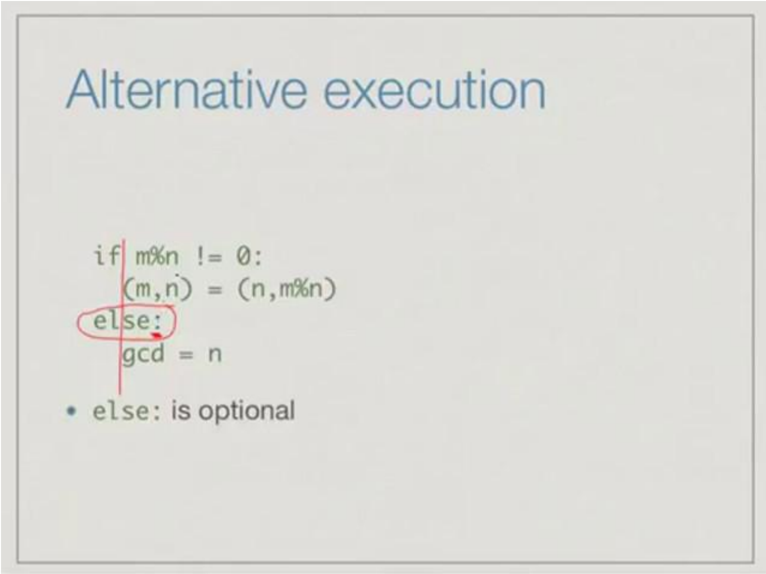
The indentation tells Python that these two statements are conditionally executed depending on the value of this condition. However, statement 3 is not governed by this condition, because it is pushed out to the same level as the if. So, by governing by describing where your text lies, you can decide the beginning and the end of what is governed by this condition.

In a conventional programming language, you would have some kind of punctuation typically something like a brace to indicate the beginning and the end of the block, which is governed by the condition. One of the nice things about Python which makes it easier to learn and to use and read is the dispensation with many of these punctuations and syntactic things which make programming languages difficult to understand. So, when you are trying to learn a programming language, you would like to start programming and not spend a lot of time understanding where to put colons, semicolons, open braces and close braces and so on.

Python tries to minimize this and that makes it an attractive language both to learn and to write code in if you are doing certain kinds of things. Python will not have this and then we will see a much cleaner program as a result. One thing we have emphasized before and, I will say it again is that this indentation has to be uniform; in other words, it must be the same number of spaces. The most dangerous thing you can do is to use a mixture of tabs and spaces, when you press the tab on your keyboard it inserts some number of spaces which might look equal to you when you see it on the screen, but Python does not confuse tabs and spaces.

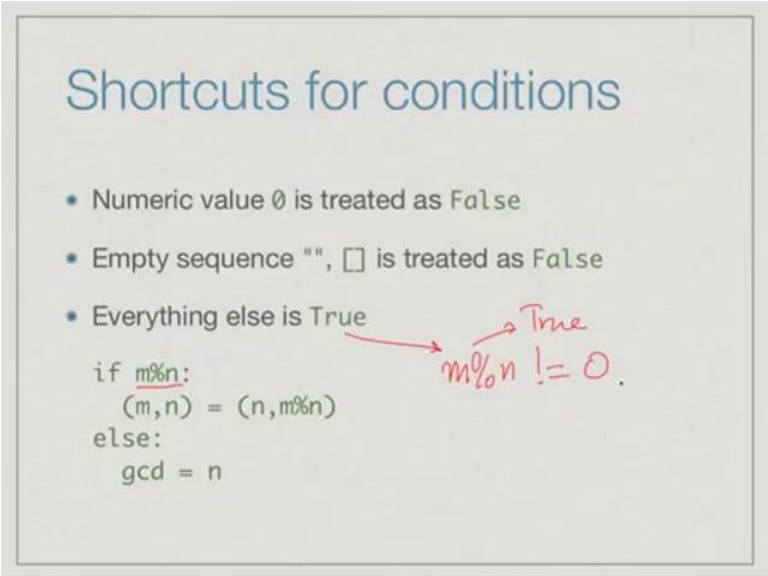
So, one tab is not going to be equal to three spaces or four spaces or whatever you see on the screen; and the more worried thing is the python will give you some kind of error message which is not very easy to understand. So, it is quite useful to not get into the situation by remembering to always use exactly some uniform strategy for example, two spaces to indent whenever you have such a nested block.

(Refer Slide Time: 06:37)




Quite often, we have two alternatives; one to be taken if the condition is true, and the other to be taken if the condition is not true. If it is likely to rain ensure the umbrella is in the bag, else ensure the umbrella is not in the bag. In this case for example, if the remainder is not 0 continue with new values for m and n if the remainder is 0 then we have found the gcd namely the smaller with two values. This is indicated using the such special word else which is like the English else again with the colon and again, we have nesting to indicate what goes into the else and the if and the else should be nested at the same level. The else is optional, so we could have the 'if' without the else.

(Refer Slide Time: 07:21)

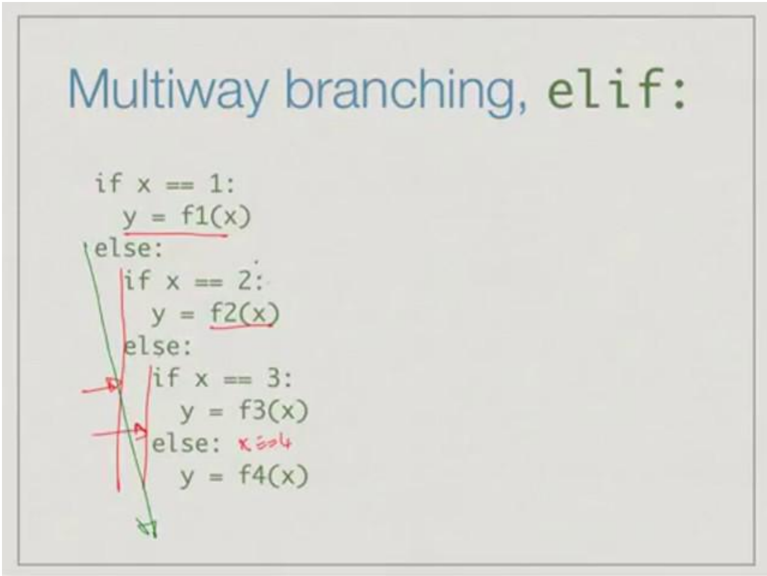



Technically speaking, the condition that we put into an 'if statement' must be an expression that explicitly returns Boolean value true or false. But like many other programming languages, Python relaxes this a little bit and allows different types of expressions which have different values like the types we have seen so far like numbers and list to also be interpreted it as true or false. In particular, any number, any expression, which returns a number 0, any numeric expression of value 0 is treated as false.

Similarly, any empty sequence such as the empty string or the empty list is also treated as false. And anything which is not in this case, so if I have a nonzero value as a number or if I have a nonempty string a string with seven characters or a nonempty list a list with three values then all of these would be interpreted as true if I just stick them into a condition. So, this can simplify our expressions and our code.

Instead of saying m percent n as we said before m percent n not equal to 0. Remember if it is not equal to 0, then it is true. If this condition holds is the same as asking whether m percent n is a nonzero value, and if it is nonzero value we want to replace m and n, so we can just write if m percent n. So this is a shortcut, now use it with care sometimes if you are used to it and if you are familiar with what is going on, this can simplify the way you write things, but if you are not familiar with what you are going on, you can make mistakes. So, if you are in doubt, write the full comparison; if you are not in doubt or if it is very obvious, then go with the shortcut.

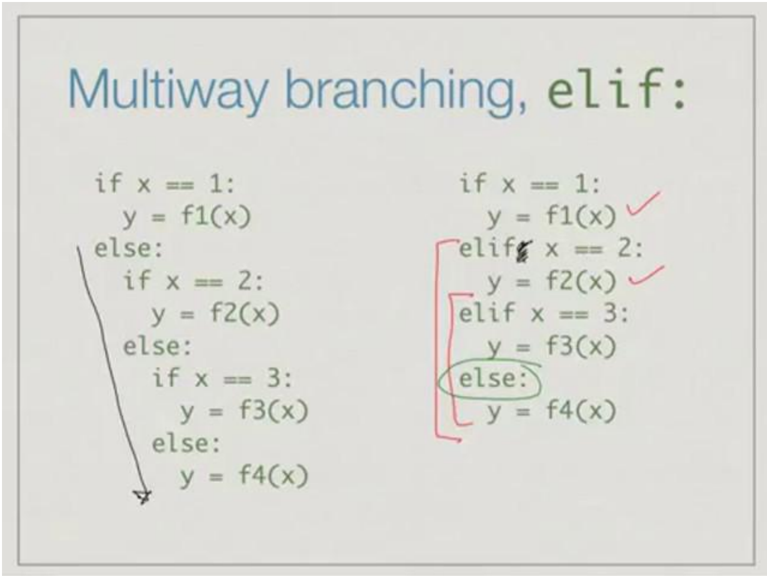
(Refer Slide Time: 09:12)




Here is a very common situation that occurs; sometimes we do not want to check between one of two conditions, but one of many conditions. So, supposing we have a value x, name x which can take a value 1, 2, 3 or 4 and depending on whether it is 1, 2, 3 or 4, you want to do four different things with which we called f1, f2, f3 and f4. So, if we simulate this four way choice, using a if and else then we have to make some comparison first. Supposing, we first check if x is equal to 1, then we invoke f1. If x is not 1, then it is one of the others, so all of this goes into an else and everything gets nested. Then we check in this case, if x is equal to 2 then we do f2 otherwise, we have 3 or 4, so now all of this is nested once again.

And finally, we have x is equal to 3 or not 3, so this is x is equal to 4 and then we are done, but the main problem with this is that first of all this code is getting indented. So, as you go into this nested if structure to simulate this multi-way branch that we have essentially a four way branch x could take one of four values, where each of these four values you want to do four different things. If we simulate it by taking 4, 3, 2-way decisions, we check 1 or not 1, then we check 2 or not 2, and then we check 3 or not 3 then we have this ugly nesting and secondly, we have this else followed immediately by an if.

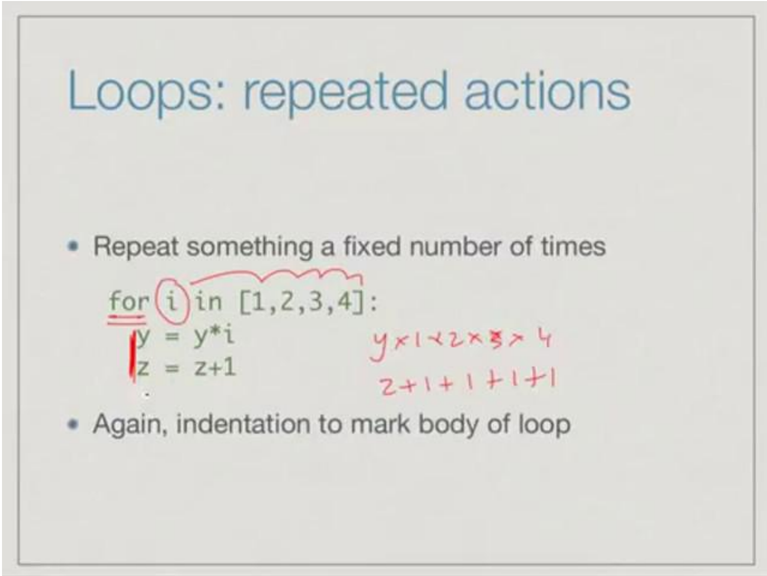
(Refer Slide Time: 10:44)




Python has a shortcut for this which is to combine the else and the if into a second check elif. So, this on the right is exactly the same as the left as for as python is concerned. It checks the condition x is equal to 1, if the x is equal to 1 then it will invoke f1, otherwise, it will invoke the rest. Now the rest, we have just collapse the else and if to directly check a new condition; if this condition holds then this works; otherwise, it will check the rest and so on. So, we can replace nested else if by elif in this way, and it makes the code more readable because it tells us that we are actually doing a multi-way check.

And notice that the last word in elif is again an else. So, if you have say seven different options, and you are only doing special things so 1, 2, 3, and 4, 5, 6, 7 are all the same then we can use else; we do not have explicitly enumerate all the other option. So, we have a number of explicit conditions that we check. So, by the way this and notice the type or they should not be a ok. So, we have a number of explicit conditions we check with if and a sequence or elifs, and finally, we have an else, the else again is optional like it is with the normal if, but the main thing is it avoids this long indentation sequence which makes the code very difficult to read later on.

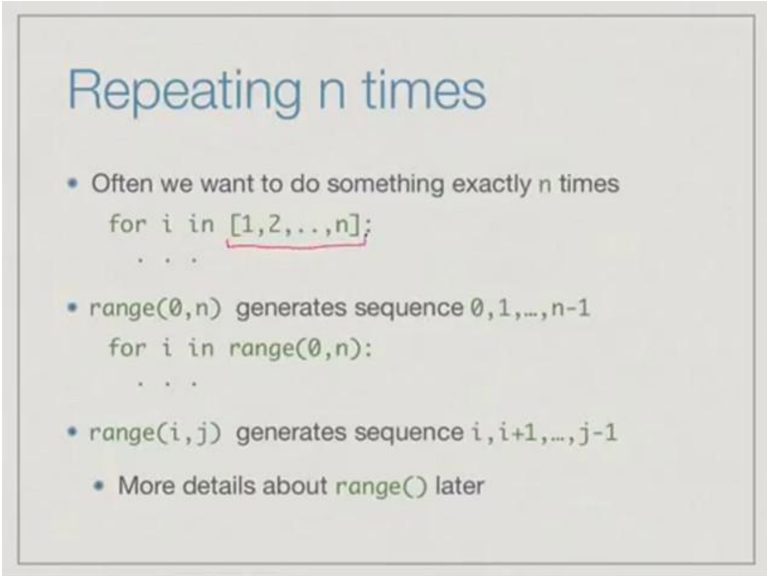
(Refer Slide Time: 12:07)




The first type of control flow which we have seen is conditional execution and the other type is loops. In a loop, we want to do something repeated number of times. So, the most basic requirement is do something a fix number of times. For instance, we have the statement for which is the keyword in python. So, what python says is take a new name and let it run through a sequence of values, and for each value in this sequence executes this once right. So, this is in sequence, it is multiplying y by 1 then the result by 2 then the result by 3 and so on.

The end of this will have y times 1 times, 2 times, 3 times 4, and we will have z plus 1 plus 1 plus 1 plus 1 - four times because each time we are adding 1 to z. So, this should be outcome of this loop. The main thing is exactly like if, we have indentation to mark the body of a loop.

(Refer Slide Time: 13:05)

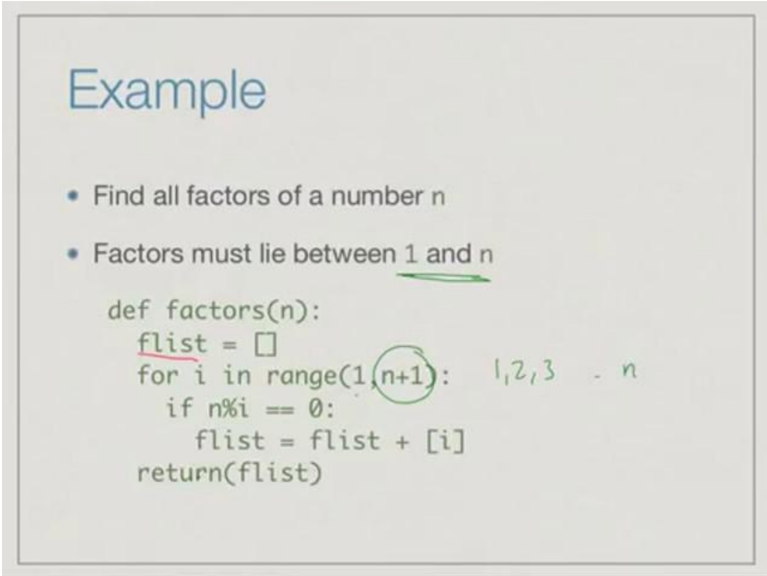



The most common case for repeating things is to do something exactly n times. So, instead of writing out a list 1 to n, what we saw is that we can generate something equivalent to this list by using this built in function range. The range 0, 1, we said it starts at 0 and it generates the sequence of the form 0, 1, 2 stopping at n minus 1. This is similar to the similar positional convention in Python which says that the positions in a list go from 0 to the length of the list minus 1. Range also does not go from 0 to n, but 0 to n minus 1.

So, instead of writing for i in a list, we can write for i in a range, and this is from the point of view of Python the same thing, either we can let this new name range over an explicit list or an implicit sequence given by the range function.

In general range i, j, like a slice i to j, starts at i and goes up to j minus 1. We can also have range functions which count down and we can have range functions which skip a value we can do every alternate value and so on. We will see these variations on range a little later, but for now just note that we can either have for statement which explicitly goes through a list of values. So, we can give a list and ask for to go through each value in that list or we can generate a sequence of n values by using the range function.

(Refer Slide Time: 14:35)

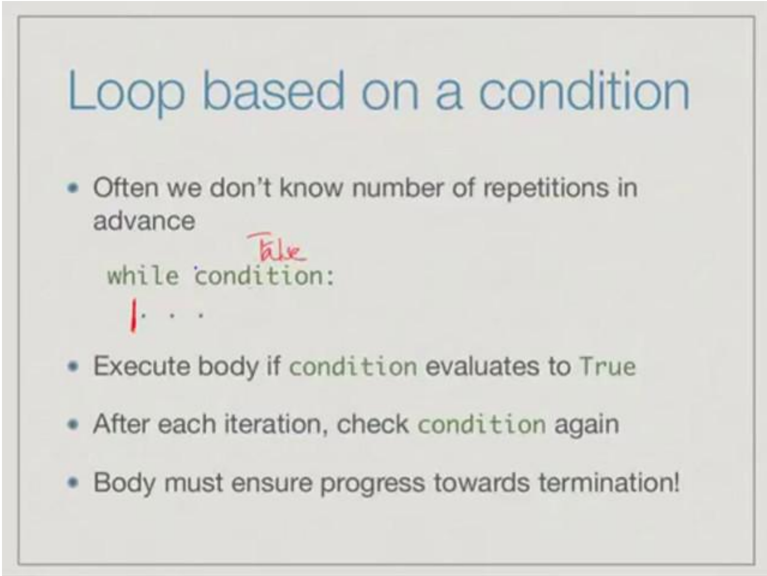



Let us look at a simple example of this. Suppose, we want to find all the factors of a number n, all numbers that divide n without a remainder. As we recalled when we did gcd, all the divisors or factors of a number must lie between 1 and n; we cannot have any number bigger than n, which is a factor of n. One simple way to check the list of factors is to try every number from 1 to n, and see if it divides, so here is a very simple function for it. So, we define a function called factors of n which is going to give back a list of all the factors.

Internally we use a name flist to the record this list. The flist, the next list of factors is initially empty. And now keeping in mind that all the factors lie between 1 and n, we generate in sequence all the numbers from 1 to n, and remember this requires the upper bound of the range to be n plus 1, because the range function stops one below the number which is the right hand side. So, this will generate a sequence 1, 2, 3 up to n. And what we check is if there is no remainder when n is divided by i then we add i to this list.

And remember that plus for a list and for a sequence is concatenation; it actually adds a value to a list, and this allows us to return a new list. The end of this, we have computed all the factors of n and return them as a list.

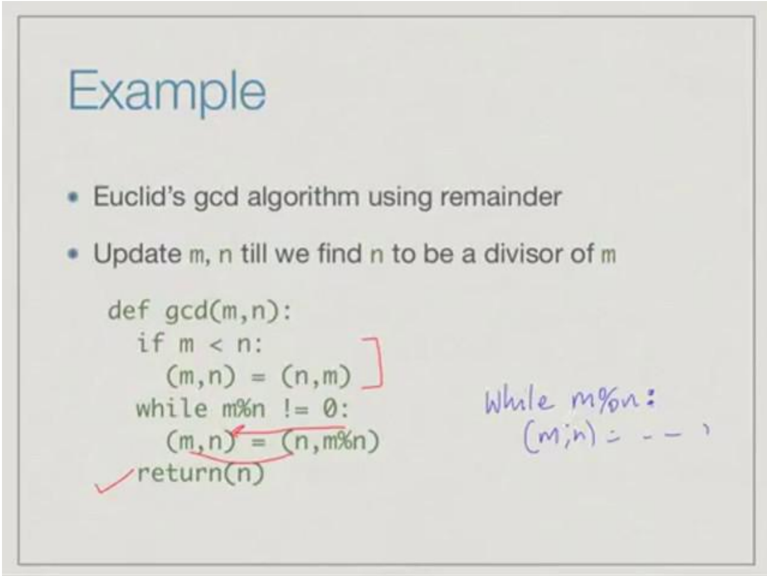
(Refer Slide Time: 16:07)




But as we said at the beginning of today's lectures, sometimes we want to do something some number of times without knowing the number of times in advance like stirring the sugar in the cup. We saw an example of this with the gcd. So what we did was, we have another statement like for called while. While executes something so long as a condition holds, so we execute this body so, long as this condition evaluates to true, and then when we have finished executing this we come back and check again whether this condition is true or not.

So the danger is that every time we come back the condition will be true and this thing will never end in the for case we know in advance that it will execute exactly as many times is length of the sequence that we started. So, when we start a for loop we give a fix sequence that fix sequence has a fix length and so we must terminate the loop in that many executions. In a while we come back we check the condition again, but there is a every possibility that the condition will never become true. We have to ensure when we write a while loop that somehow this sequence of statements to execute inside the while must eventually make this thing to be false, because if this thing never become false, condition will never, the loop will never terminate.

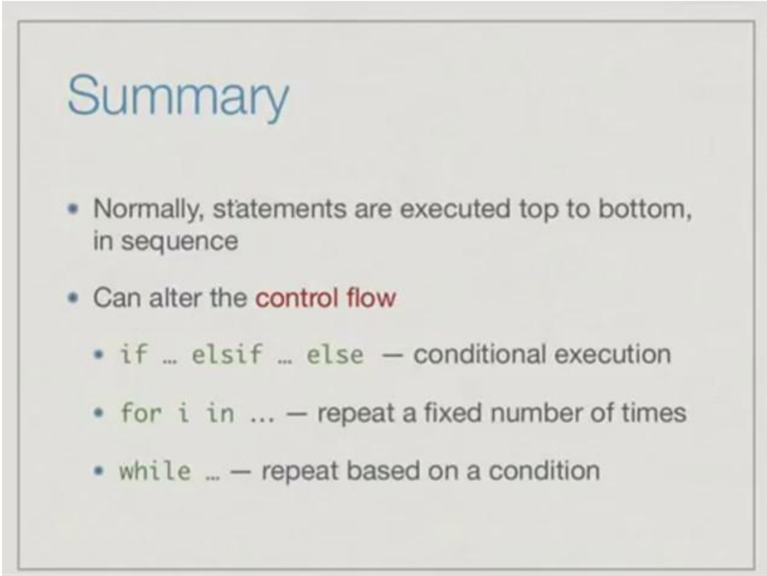
(Refer Slide Time: 17:29)




The example we saw with gcd was the variation of what we wrote so far. We first of course check and swap, so that the bigger number is first and now we check whether the bigger number is divisible by the smaller. So, so long as this is not the case, we exchange values for m and n. We make m point to the smaller number n, and we make n point to the remainder, which will be still smaller; and eventually this thing will become false, we will get the situation where n divides m and the remainder is 0 and at that point we have found the gcd.

This is a simple example, remember that by our earlier convention we could also write this as while m percent n make this thing. So, we do not need this explicit not equal to 0, because the value m percent n if it is not 0 is treated as true and the loop will go one more time, but now in some situations it may or may not be so easy read this. It might be useful to just say explicitly not equal to 0, just to illustrate to yourself and to the person reading your code what is going on.

(Refer Slide Time: 18:41)



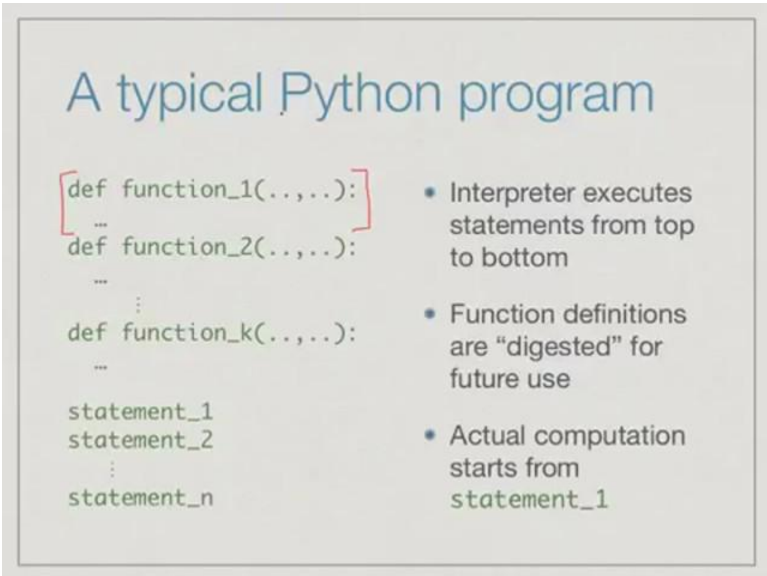

To summarize, a Python interpreter normally executes code from top to bottom in sequence. We can alter the control flow in three ways we have seen. One is using the 'if statement', which conditionally executes and this extends to the elif which allows us to do a multi-way branch. Then there is for statement which allows us to repeat something a fixed number of times that providing a list or a sequence of values from range. And then we have a repetition which is based on a condition using a while statement where we put a condition and then the body is executed each time the condition is checked again so long this is true the body keeps repeating.

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

**Week - 02**
**Lecture - 05**
**Functions**

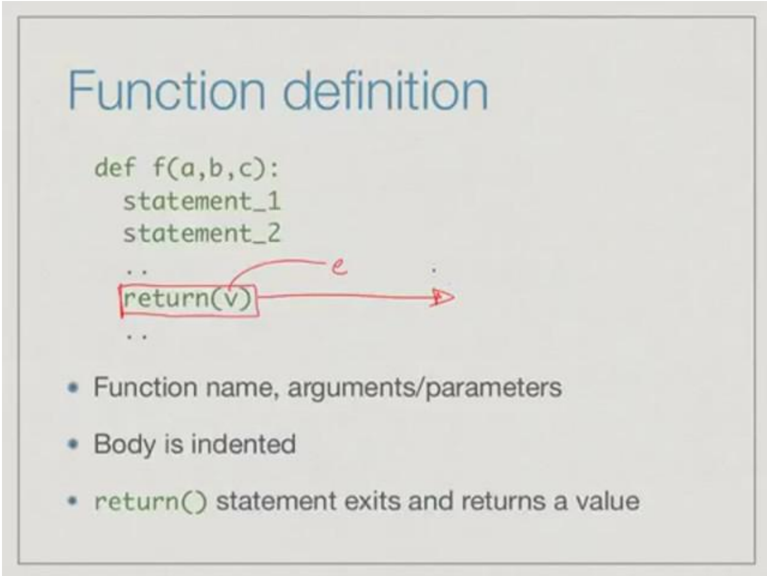
We have seen how to alter the flow of a program by using if, for and while. We can have conditional execution, we can have repeated execution.

(Refer Slide Time: 00:01)




The last ingredient in our typical Python program is a function. What is a function? A function is a group of statements which performs a given task. So of course, we could write the function code as part of the main program, but by isolating it we can logically separate out units of work and very often these functions are called repeatedly with different arguments. So, they constitute a unit of computation which can be used repeatedly from time to time.

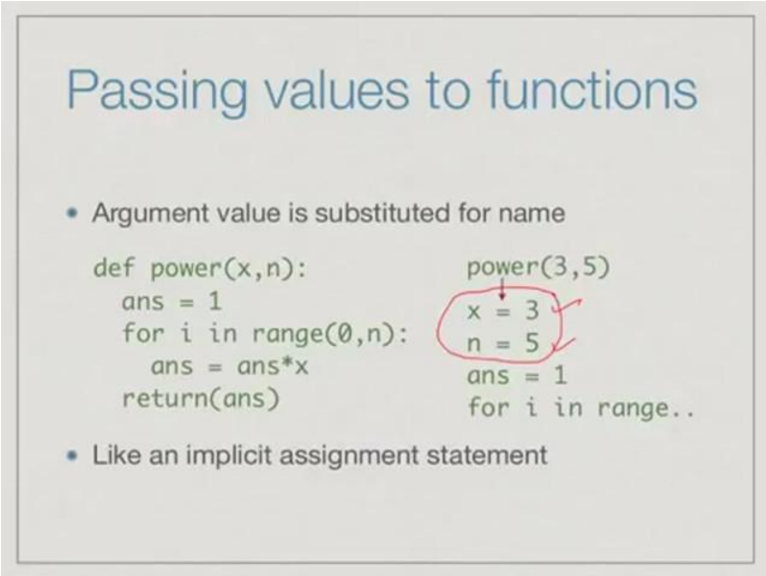
(Refer Slide Time: 00:43)




We define functions using the def statement as we have seen informally. So the definition defines the name of function, in this case we have just called it f usually we would give it more meaningful names. Then it says that this function takes three values as inputs, so these are called parameters or arguments. So, the first one is called a, the second is called b, and third is called c, and within the body of the program of the function a, b and c will refer to the values which are passed to this function for a given call. Within a function we might have a statement like this called return.

The body of the function is indented like we had for if, while and for, and the return statement if it is encountered, it says that at this point the execution of the function will end and you will get back to where you called function from returning the value in the name v. This could be any expression; we could just have return of a constant or return of v plus one or whatever.

(Refer Slide Time: 01:50)

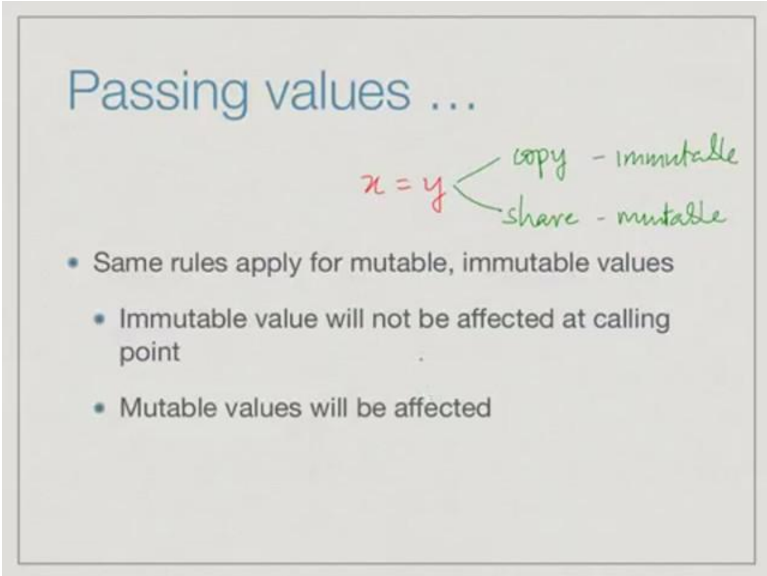



When we call a function we have to pass values for the arguments, and this is actually done exactly the same way as assigning that value to a name. Suppose, we have function like this which takes x and raises it to the power n. Let us just look at the function just to understand what the code is doing. We assume that the value of the answer is 1. And now for as many i as there are in the range 0 to n minus 1 we multiply x into answer so we get effectively x times, x times, x n times. Each time we go through this loop we multiply one more x and finally we return the answer that we have got.

Now the way we would use this function in our code is to write an expression of the form, say power 3, 5, so obviously, what this means is that 3 should be used for x and 5 should be used for n and we would then run this code with the values x equal to 3 and n equal to 5.

Actually, you can imagine that when we run this code, it is as though we have this code inserted into our program at this point preceded by this assignment. So, this assignment basically says set the value of the name x to the value passed by this namely 3, set n to 5. This assignment is what takes place effectively when you call a function. And since it is an assignment, this behaves very much like assignment in the regular case.

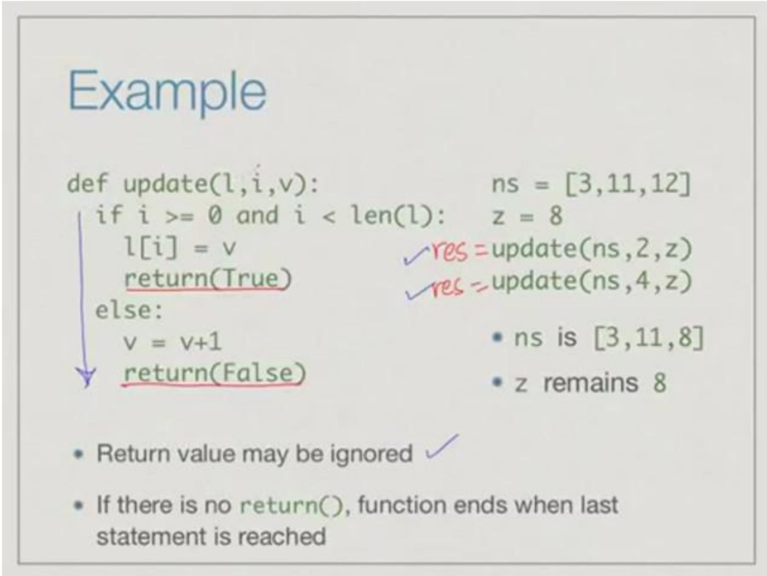
(Refer Slide Time: 03:32)




In particular the same rules apply for mutable and immutable values. Remember we said that when we write something like x equal to y, if it is immutable that is the value in y cannot be change in place then we copy the value and we get a fresh copy in x, so the value in x and the value in y are disjoint. So this is if it is immutable. And if it is mutable, we said we do not copy, we share the value; that is, both names will point to the same copy of the value, so change in one will also make a change in the other; that happens with mutable things like lists.

Immutable values will not be affected at the calling point in our case and mutable values will be affected. It is as though we are making an assignment of the expression or the name in the calling function, calling point to the name in the function. So, if the function modifies that name, the value of that name; if it is immutable value, nothing will happen here, if it is a mutable value something will happen.

(Refer Slide Time: 04:36)




So, here is a simple function just to illustrate this point. The aim of this function is to update a list. So, I give you a list which is called in this function l and I give you a position which is i and what I want to do is I want to replace whatever is there by a new value v. So I get three arguments, l is a list, and then i is the index of the position, and finally v is the value to be replaced.

So, what do we do? First check that the index is a valid index. We check that it lies between 0 and l minus 1. So it is greater than equal to 0 and it is strictly lesser the length of l. If so, what we do is just replace l of i by the value v which we have got and we return true to indicate that the update succeeded. Now if i is not in this range then we cannot do an update. So, what we will do is effectively return false. This is just say that the update did not work and then the person, the part of code which is calling this can understand that something went wrong and presumably what went wrong is the index was not in the valid range.

But just to illustrate what happens with immutable values, in this case we are also updating for no good reason the value v to be v plus 1. So, remember that v is being passes a value to be put in here and we are assuming normally that v would be a immutable value. Let us assume we call it now, so what we use do is we set up a list of numbers 3, 11, 12 and then we want to replace this 12 say by 8. So, just for the sake of argument we first set up a new name z called 8 and we say update the list n s at position 2, so remember the positions are 0, 1, 2. So, update the list in position 2 by the value of z.

And then we say update the same list at position 4 by the value of z. Now as we saw if the values 4 right then this if will fail, so it will instead go here, this won't work, so it will go here. And what will happen inside the code is that v will be incremented, now v has been copied from z. The question is what happens to z? So, as you would expect after executing these four statements, because of this update succeeding the value of z is copied into the list that position 2 and so we get the value 8 instead of the value 12 that we started with. On the other hand, if we execute this statement, then because this is an immutable value the change in v inside the function does not affect z at all. Although v has been incremented from 8 to 9, z remains 8.

This is just to illustrate that if we pass a parameter, through a parameter a value that is mutable it can get updated in function and this is sometimes called a side effect. So the function affects the value in the other program, so this is called a side effect. A side effect can happen if the value is mutable, but if the value is immutable then the value does not change no matter what we do inside the program.

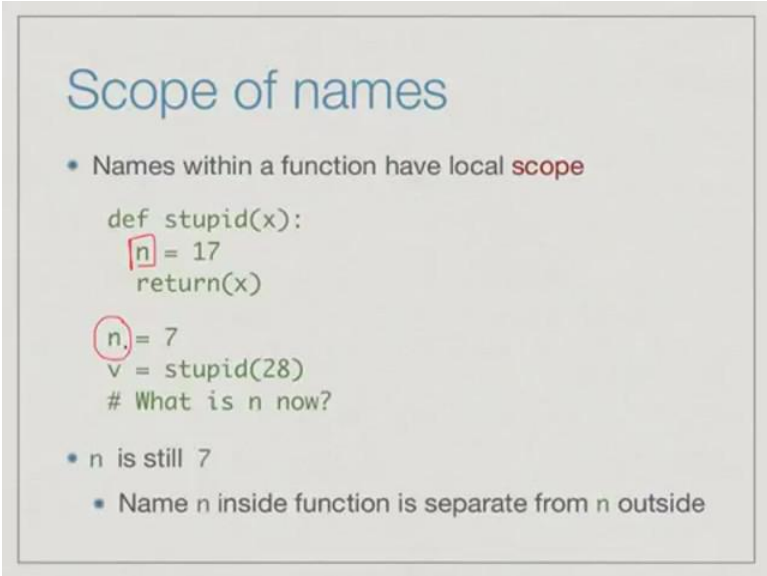
Now, there are couple of other points to note about this function just to illustrate; one is that we have here two return statements: return true or return false. The idea is that they indicate to the calling function whether or not the update succeeded. So ideally you should have said something like result is equal to update, and then check after the update where the result is true or false.

Remember update will update the list or not update the list depending on whether the index is valid and it will return true or false depending on whether they update succeeded. So, by examining the value of whatever is returned we can check whether the update we intended worked or not. This is something which we would expect but we have not done it, so this is just to illustrate that there may be a return value but may be the idea is a function will actually update some mutable value so we do not care what it returns all the work is done inside the function.

Even though there is a return value you are not obliged to use it, you can just call a function as a separate statement as we have done here it does not have to be part of an assignment. The other thing is that because of this there may be functions which do not return anything useful at all. A typical example would be a function which just displays a message like there was an error or it displays some other indicative things for you to understand what your code is doing. Now such a function just as to display something, it does not have to compute or return anything. So, there may be no return function. So, by default what happens is that a function executes like everything else from top to bottom when it is involved.

And now if you encounter a return statement at that point the function stops executing and you go back. On the other hand if you run out of statements to execute, if you reach the last statement then there is nothing more then also the function will end. There is no obligation for a function to actually have a return statement. So, a return statement is useful if the function computes the value and gives you back some result which you will use later on, but you may have functions which do not have return value, in which case you can either return some empty thing or you can return nothing and everything will work fine.

(Refer Slide Time: 10:07)

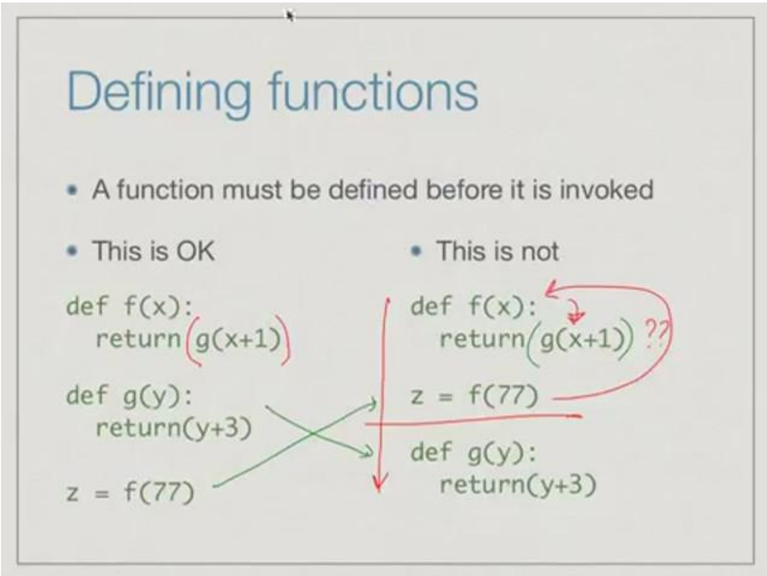



Another point to note about functions in Python is that names within a function are disjoint from names outside a function. So let us look at again at a kind of toy example which does not have anything useful to do. We have a function which we call stupid which takes essentially takes an argument and return it, so it does nothing. But in between what it does is it just for no good reason sets name n to have the 17. Now suppose we had in our program outside, a statement which assigned the value 7 to the name n and then we call this function. Now obviously, if we say stupid of 8 then v will also be the input, so v will become 28.

The question is that while executing the fact that v is 28; the function internally set n equal to 17. The question is, we have asked n to be 7 then we call this function n became 17 inside the function is n 17 now or not. So the answer is that n is still 7 and that is because the n inside and the n outside are two different copies of n. So, any name which is used in side of function is to be thought of as disjoint from the name outside. Names outside are not visible inside, the names inside are not visible outside.

Now this is not something that you would normally do because is just confusing if you use the same name inside and outside, but sometimes it is useful to have this separation because very often we do use common things like i j k run through list you know like ranges and things like that. And it will be a nuisance if we have to use a, remember and use i outside and j inside and make sure that they do not interact. Since they do not interact anyway we can freely use i j wherever we want and not worry about the fact that we are already how i or j outside in the calling function.

(Refer Slide Time: 12:07)




One of the things that we mentioned up front was that a function must be defined before it is invoked. Now this is a slightly subtle point, so let us just look at it little more. Remember that a Python program is read from top to bottom by the interpreter. So, when the Python program is read it reads the definition of f, but does not execute it, and notice that this definition of f has an invocation to g which is actually later.

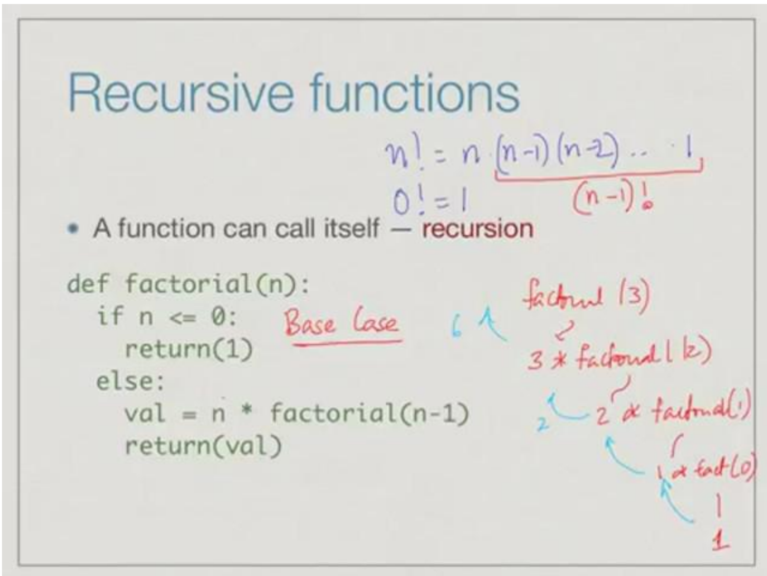
But the point is when reading definition of f g is not used it is only remembered that this statement which should be in a bracket, just to be consistent. So, this statement should be computed if I call f so it is not calling f it is just defining f. So, I define f, then I define g, finally when I come to this statement it says what is f of 77. So, f of 77 will come here and we will say f of 77 is nothing but g of 78 right so that will come here. And it will say g of 78 is nothing but 81 so it will come here. So 81. And then finally I will get 81.

So, it is only when I execute the statement f is executed at that time g has already been seen. Though we say a function must be defined before it is invoked it does not rule out the fact that one function can call a function which is defined after it, provided that you use this function only after that definition. So this sequence is fine. Suppose, we rewrote this sequence in a different way, so supposing we had the definition of f then we had this statement. We have basically exchanged these two statements.

Now what happens is that when the Python interpreter comes down this line at this point it will try and call f, so f will try and call g and g will say well I do not have a definition for g yet because I am not yet gone past this statement. So if I put this statement, execute f before I define g and f requires g then this statement will create an error whereas this statement will not.

It's really useful if we define all functions upfront because any inter dependency between functions will be resolved right way by the interpreter and we do not have to worry about it. Whereas, if we do this inter mixing of functions and statements then we have to be careful that functions do not refer to the later things which have not been scanned yet by the interpreter. This is one more reason to put all your function definitions at the beginning and only then have the statements that you want to execute.

(Refer Slide Time: 14:40)



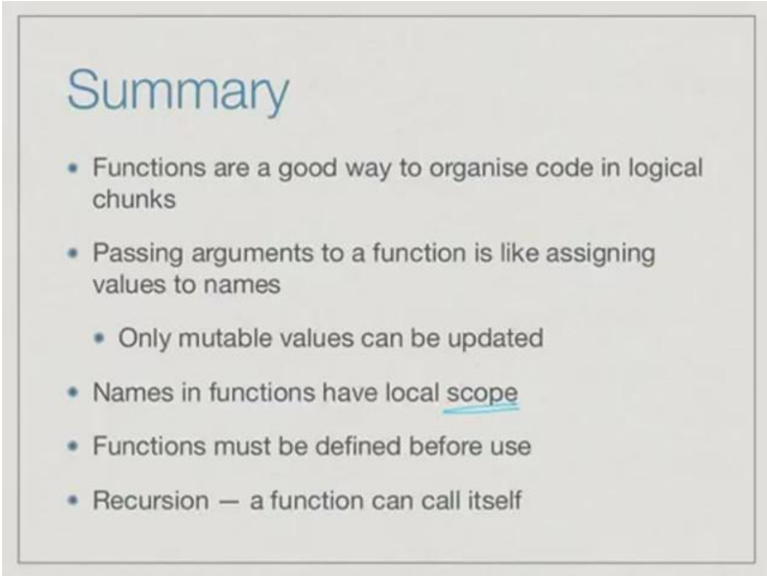
A final point that we will return to later when we go through more interesting examples as we proceed in programming, is that a function can very well call itself. The most canonical function of this kind, these are called Recursive functions. Functions which rely on themselves. Is the factorial function. If you remember n factorial is defined to be n into n minus 1 into n minus 2 into n down to 1. So you take n and multiply it by all the numbers smaller than itself up to 1 and by definition 0 factorial is defined to be 1.

What we observe in this definition is that, this part from n minus 1 to 1 is actually the same as n minus 1 factorial. In other words n factorial can be defined in terms of a smaller factorial it is n times n minus 1 factorial, so that is what this function is exploiting. There is a base case factorial of 0 is 1 and since the factorial of negative numbers is not defined and we want to be safe we can say that if n is equal to 0 or n is less then equal to 0 we return 1. So this is what we normally call the base case.

In this case the factorial is completely defined without having to do any further work. Now if n is not less than equal to 0 then n is greater than 0. If n is greater than 0 then we take the current number and we multiply it by the smaller factorial that is exactly the definition given above. So if I take say factorial of 3, this will result in 3 times factorial of 2 so that will invoke this function again and this will give me 2 times factorial of 1 and so on.

Factorial of 1 will give me 1 times factorial of 0 and the point is that factorial 0 will now terminate and it will give me 1, because it says that argument is less than or equal to 0 return 1. This 1 will return now come back and get multiplied here, so you get 1 times 1, so 1 times 1 will come here and will come here, so then this will bring back 2 and then 3 times 2 this will become 6. This is how the function will execute we will talk about this more later, but just to illustrate that functions can very well call themselves.

(Refer Slide Time: 17:03)




To summarize, functions are a good way to organize your code into logical chunks. So if you have a unit of computation which is done repeatedly and very often done with different possible starting values then you should push it aside into a function. If you break up your code into smaller functions, it is much easier to understand, to read and to maintain. When we pass arguments to a function it is exactly like assigning values to a name.

So, the values that are passed can get updated in a function only if they are mutable, if they are immutable any change within a function does not affect the argument outside. Also if we use the same name inside a function as is found outside a function the name inside the function does not in any way affect the name outside. So, functions have local notion of what we call scope. There is a scope of a name where is a name understood, so the name inside a function does not exist outside and vice versa.

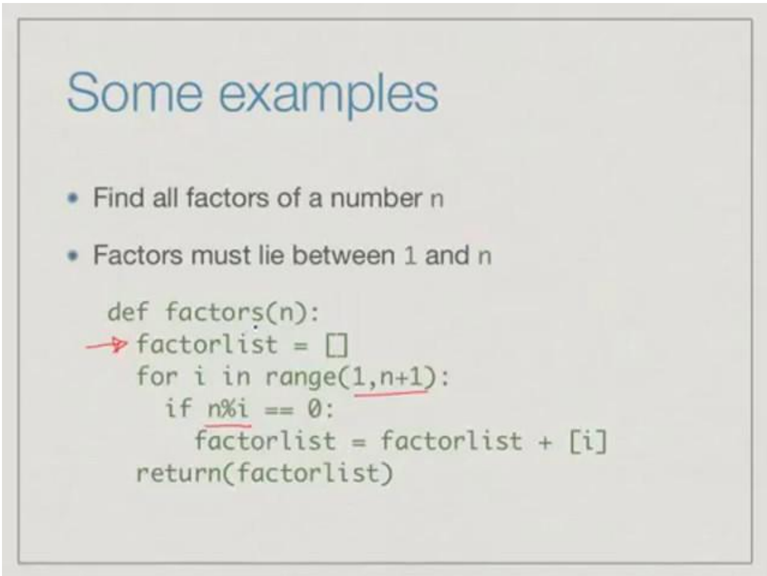
Also functions must be defined before they are used and this is a good reason to push all your function definitions to the beginning of your program, so that the Python interpreter will digest them all before there are actually invoked. So if there are mutual dependencies we do not have a problem. Finally, we saw that we can write interesting functions which call themselves and we will see many more examples of this in the weeks to come.

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

**Week - 02**
**Lecture - 06**
**Examples**

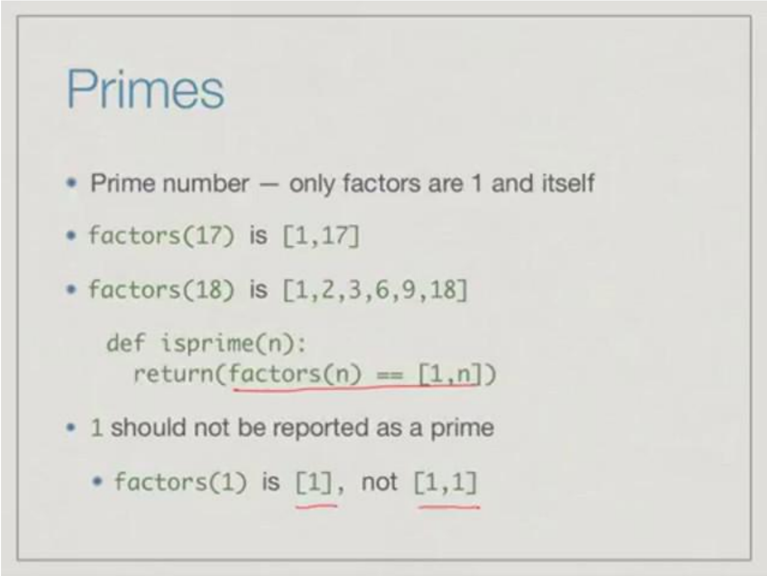
To round of this week, let us look at some examples to illustrate some of the concepts we have seen so far.

(Refer Slide Time: 00:11)




We have already seen a function which computes the factors of a number n. So, we observed that all the factors must lie between 1 and n. This is something we can naturally compute using a for loop. We define factors of n as follows; we assume that the list of factors is empty, and for each number in the range 1 to n if that number is a divisor, is a factor of n we append it to the list of factors and eventually we return this list. This is a simple function which just takes the list of factors gives back the list of factors of the input n.

(Refer Slide Time: 00:47)



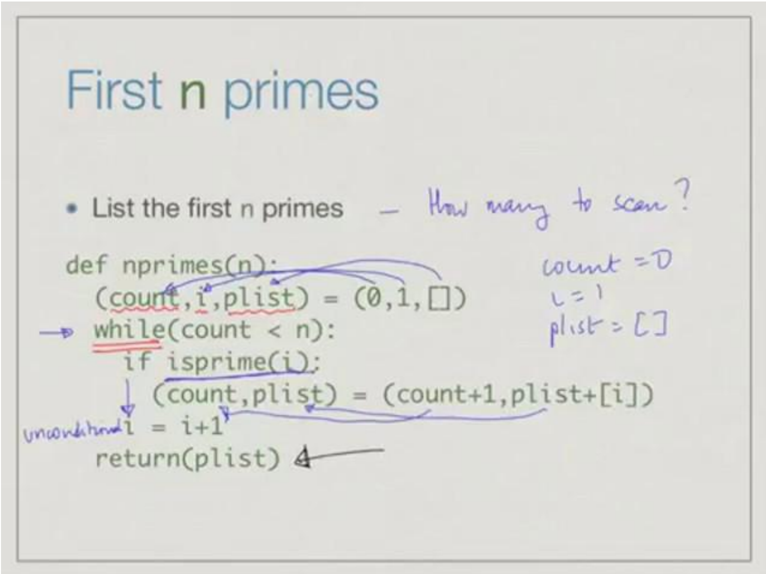

A prime number is one which is divisible by no other number other rather than 1 and itself. In other words, the only factors of n should be 1 and n, if n is a prime. So, 17 for example, which is a prime number has only two factors 1 and 17. Whereas, 18 which is not a prime have many more factors, it is also 2 times 9 and 3 times 6. So the list of factors of 18 is a longer list and just 1 comma 18. This allows us to write a very simple definition of prime based on factors which we have already seen. Number is prime, if the list of factors is exactly 1 comma n.

So what we do is we invoke the function factors and check what it returns and see if it is equal to the list 1 comma n. This is another illustration of the fact that if we break up our code into functions then we can use functions one inside the other, and break up our problem into smaller units which are easier to digest and to understand. Here, we said that a prime number has only two factors 1 and itself. We have separately written a way to compute the list of factors, so we can take that list and directly check whether or not a given number is prime.

One small thing to be aware of when we are dealing with prime numbers is that we should not accidentally define 1 to be a prime, because if you just look at this definition that the only factors are 1 and itself, it is a bit ambiguous because 1 is a factor and itself 1 is also a factor. We could naively read this as saying that 1 is a prime, but by convention 1 is not a prime. So, there should be two factors separately 1 and itself.

Fortunately we call our function factors then factors of 1 will return as single list containing - singleton list containing the value 1, because it will run from the code will run from 1 to 1, so it will only find it once. Whereas, in order for this return statement to be true, I would like the actual value to be 1 comma 1, 1 comma n, so n is 1. So fortunately, the way we have written isprime, it will correctly report that 1 is not a prime, but these are the kind of boundary conditions that one must be careful to check when one is writing functions in python or any other programming language.

(Refer Slide Time: 03:09)




What if you want to list all the prime numbers which lie below a given number n. So, we have to just check for every number from 1 to n, whether or not it is a prime. We already know how to check if something is a prime. Once again, we can write a function primesupto which takes an argument n. So, initially we say that there are no primes up to n. Now for all numbers in the range 1 to n plus 1, which means from 1, 2 up to n. We check if that number i is a prime, now this is a function we have already written. If it is a prime then we append it our list, if not we go to the next one and finally we return the list of primes we have seen.

Once again, now we have use a function we have written before isprime. isprime in turn uses the function called factors which we do not see here. We have three levels of functions now, primesupto which calls isprime, which calls factors. This is a very typical way you write nice programs where you break up your work into small functions. Now the other advantage of breaking up your work into small functions is that if you want to optimize or make something more efficient, you might first write most obvious or naive way to implement a function so that you can check that your overall code does what it supposed to do then you can separately go into each function and then update or optimize it to make more efficient. So, breaking up your code into functions makes it easier to update your code and to maintain it, and change parts of it without affecting the rest.

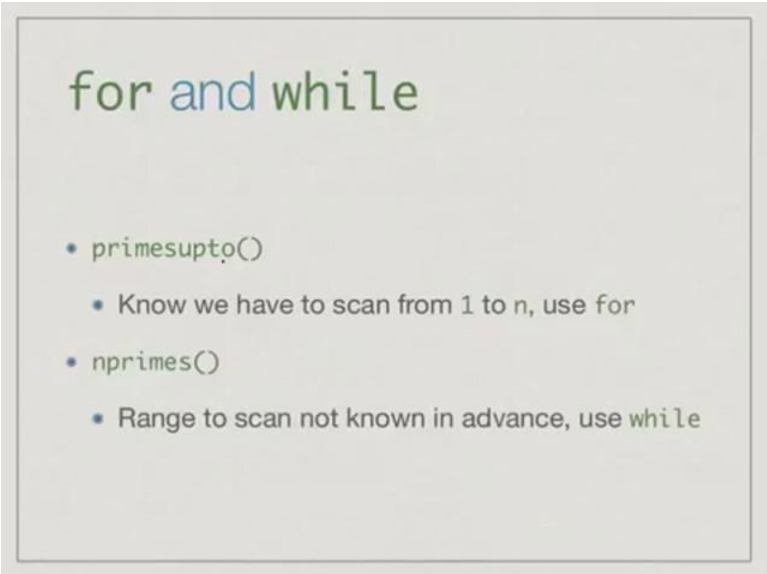
Primes up to n, we knew in advance that we have to check all the numbers from 1 to n, so we could use for. What if we change our requirements saying that we do not want primes up to n, but we want the first n primes. Now, if we want the first n primes, we do not know how many numbers to scan. We do not know where the n th prime will come. If n is small, we might be able to figure out just by looking at it, but if n is large, if we ask the first thousand primes it is very difficult to estimate how big the thousandth of prime will be. So, we have to keep going until we find thousand primes and we do not know in advance. This is a good case for using the other type of loop namely while.

So, what we need to do is we need to keep a count of how many primes we have seen. We need to go through all the numbers one at a time to check if they are primes, and we need a list of all the primes we have seen so far. Just to illustrate another point that we have seen this is a simultaneous assignment which says okay, initially we have seen 0 prime, so count is 0. We start with the number 1, this is the first number we check whether it is a prime or not, and initially the list of primes is empty. So this says, take these three values, take three these names and assign them these three values. Assign this is same, same as count is equal to 0; i is equal to 1; and plist is empty. So, this has the same effect this particular assignment.

Now so long as we have not seen n primes, while count is less than n, we have to check whether the current i is a prime and if so, add it. If the first, if the value i we are looking at is a prime then first of all we have found one more prime, so we increment the value of count. And we have found a prime, so we must add it to the list of primes. If this is not a prime, we must go to the next number; in any case, we must go to the next number and until we hit a count of n. So, we outside if, so this is unconditional we always update. So, for each i we first check if it is a prime, if it is a prime we update count and we append i to plist. And whether or not it is a prime, we increment i; and each time, we increment count we are making progress towards this while terminating. Remember that it is our job to make sure that this while will eventually get out this condition will become false.

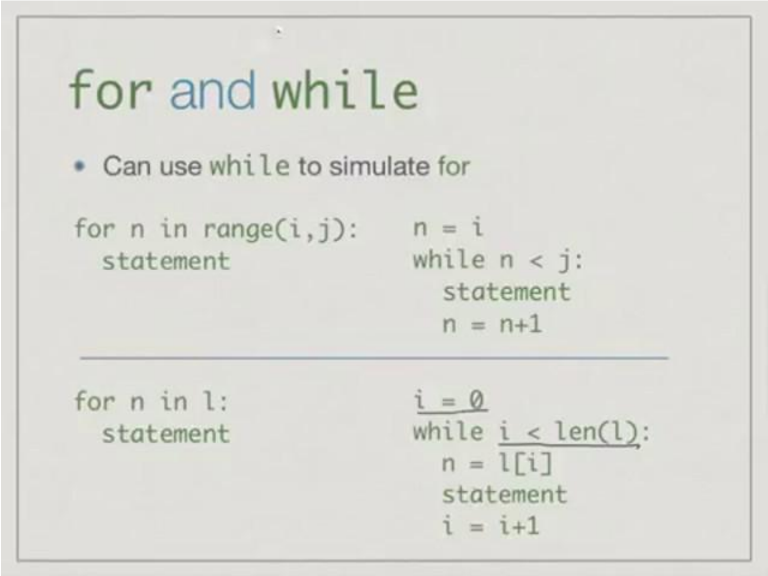
Every time we see a prime count is going to become count plus 1. We start with count equal to 0, so eventually it is going to cross n. Of course, we are using fact the implicit fact we know that there are an infinite number of primes. We were always for any n, we able to find the first n primes. So, when we do find n primes, when we have gone from 0 to n minus 1, and we have reach the count n, we have seen n primes then we return the list that we found so far and this is plist.

(Refer Slide Time: 07:31)




These two examples, the primes up to n and n primes illustrate the difference between the uses of for and while. In primes up to n, we know that we must scan all the numbers from 1 to n, so it is easy to use the 'for'. In nprimes, we do not know how many primes, how many numbers we have to scan to find nprimes, so we use a while and we keep count and we wait for the count to cross the number that we are looking for.

(Refer Slide Time: 07:58)

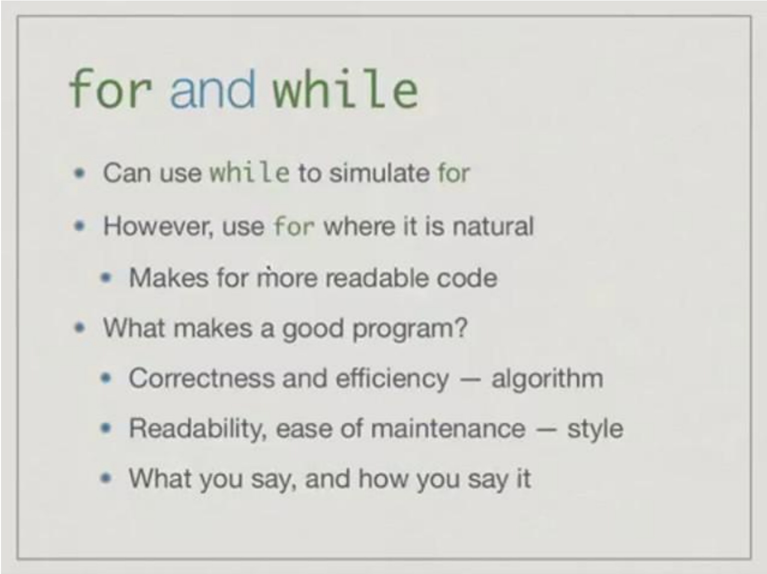



So, it turns out that you do not actually need a 'for', you can always simulate a 'for' by a while. Let us look at the two typical ways in which we write for. The first way is this for in a range, so we say for n in the range i to j. We start at i and we let n go through the sequence i, i plus 1 up to j minus 1; and for each value of n, we execute this statement or there might be more statements here. This is a block of things inside the loop that we will execute. If you want to do this without for, we could do it with a while as follows. So we initialize, so says with the first value of n is i; and so long as n does not cross j, you execute this statement - the exact same statement, and you increment n.

So, it comes back here and you check now you get i plus 1, i plus 2, and then when you reach j minus 1 then next time it will be j and it will exit. We have a range in the 'for', we can just setup a counter and manually increment the counter and check the counter value against the upper bound in the while. The other way that we use for is to iterate through the elements of a list. So, n now if a l is a list of values x1, x2 up to xk, n will take each value in this list.

Here, we can now setup positions to walk through the list and pickup the value at each position. We say we start at the 0th position; and so long as we have not reach the end of the list in terms of positions, we set n to be the value at position i then we execute this statement and we increment position. So, both forms of the 'for' can be written as while, but notice that the right hand side is significantly more ugly and complicated than the left hand side.

(Refer Slide Time: 09:40)




While we can use this, the while statement to simulate the 'for statement' it is much nicer to use the 'for', where it is natural where we know in advance what exactly we are repeating over. This makes for more readable code. And in general, more readable code makes for a better program. So, what do you need to do to write a good program? Well first of all it must do what you expect it to do, so it must be correct. And secondly, it must do it in as efficient a way as possible; we are not looked at efficiency so far, we will look at it as we get further in the course. But we want to do things quickly we want to do things the most with the efficient manner; and this is where the algorithm comes in, how you do something is what the algorithm will tell you.

And the second part is that you must write down your instructions in a way that as easy for you to validated as being correct, and for somebody else to understand and if necessary update. Now we should not under emphasize this fact of maintenance you write something today, which serves the certain function very often somebody will have to later on either make it more efficient or change the functionality increase the range of things your program does and in that case very often, it is not the person who writes the codes but somebody else who have the job of understanding and updating the code.

So, at every stage if the person who is making modification starting from the person, who wrote the code initially writes it in a clear and readable style it makes it much easier to maintain the code in a robust manner as it evolves.

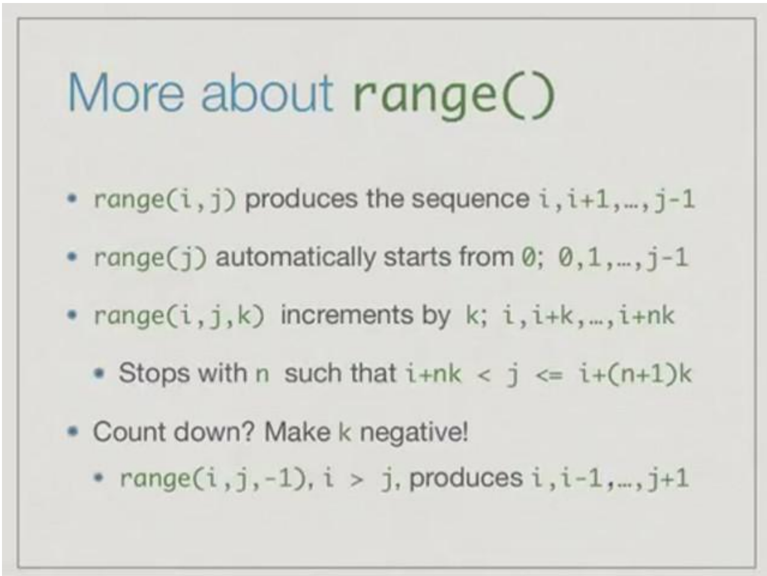
To summarize, there are two parts of programming; first is what you want to say which the algorithm is, in the second part is how you say which is the style. So, every programming language has a style use the style to make for the most effective and readable code you can find; if you have a 'for' - use it, do not force as I said to use a while and so on.

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

**Week - 03**
**Lecture - 01**

# More about Range ( )

(Refer Slide Time: 00:02)




We have seen the range function which produces a sequence of values. In general, if we write range i comma j, we get the sequence i, i plus 1 up to j minus 1.

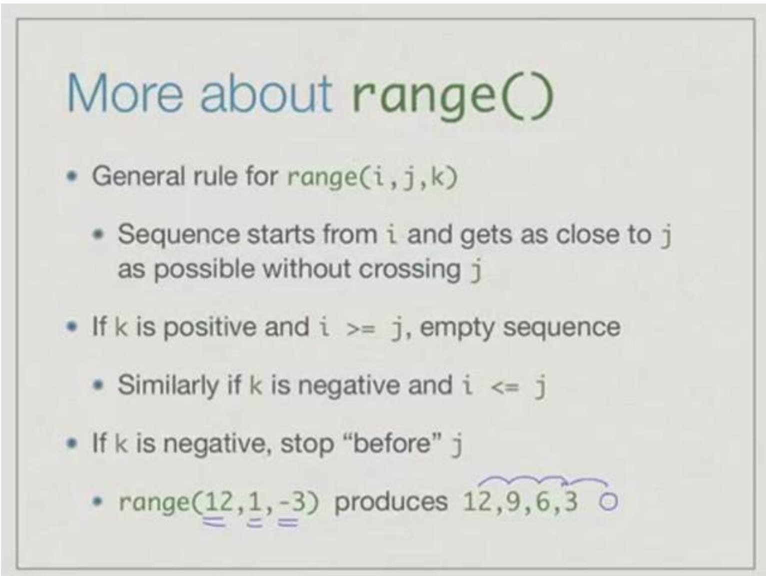
Quite often we want to start with 0. So, if we will give only one argument if we just write range j, this is seen as the upper bound and the lower bound is 0. This is like a slice, where if you do not write the first argument of the slice, if we write for instance l colon n then it will run from 0 to n minus 1. In the same way if we just write range j, automatically we will get 0, 1 up to j minus 1.

Often we may want to generate a sequence where we skip by a value other than 1. We want a kind of arithmetic progression if you are familiar with that. So, we want i, i plus k, i plus 2 k and so on, we do this by giving a third argument. The third argument if we give it to range tells the range function to skip every k item. So, we have i then we go directly to i plus k. So, implicitly if we do not say anything it is like we put a 1 here right. So, the default value is 1 i, i plus 1 and so on.

Now, how far does this go? Well, we want to go until we reach normally j minus 1. In general, we do not want to cross j. So, what we will do is we will get i plus n k for the largest n such that i plus n k is smaller than j, but if i go one more step, if i go to i plus n plus 1 times k then i will cross j right. So, if we have a step then we will keep going until we cross j and we will stop at the last value that is before j.

Having a step also allows us to count down. All we have to do is make the step negative. So, if we say i comma j comma minus 1 then provided we start with the value which is bigger than the final value, we will start with i produce, i minus 1, i minus 2 and so on and we will stop with j plus 1.

(Refer Slide Time: 02:07)




The general rule for the range function is that you start with i and you increment or decrement if k is negative, in steps of k such that you keep going as far as possible without crossing j. In particular, what this means is that if you are going forward, if you are crossing, if you have positive, if your increment is positive then if you start with the value which is too large then you will generate the empty sequence because you cannot even generate i because i itself is bigger than j or equal to j then that would not be allowed.

Conversely in the negative direction what happens is that if we start with the value which is smaller than the target value, we are already below j and so we cannot proceed, we get an empty sequence. This idea about not crossing j it is not same as saying stops smaller than j because if you are going in the negative direction you want to stop at a value larger than j. So, you can think of it as before and before means different things depending on whether you are going forwards or backwards.

Just to see an example, suppose we want to have a range which starts from 12 and whose limit is 1, but the increment is minus 3. So, we will start with 12 and then we will go to 9, then we will go to 6, then we will go to 3 and if we were to go one more step you would go to 0, but since 0 crosses 1 in the negative direction we would stop at 3 itself, we would not cross over to 0.

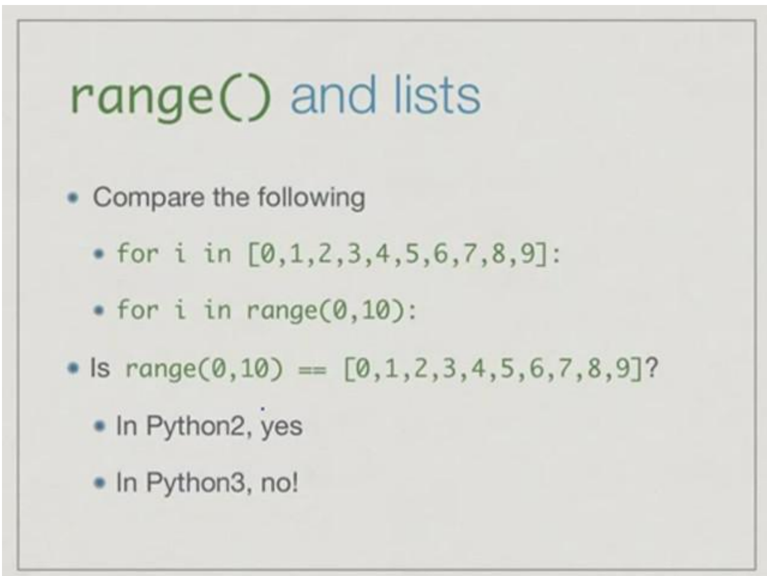
(Refer Slide Time: 03:42)


It is often confusing to new programmers that range i comma j is actually from i to j minus 1 and not to j itself. So, there is no real reason why it should be this way, it is just a convenience and the main convenience is that this makes it easier to process lists. Remember that if we have a list of length n, the positions in the list are numbered 0, 1 up to n minus 1. So, very often what we want to do is range over the indices from 0 to n minus 1 and if we do not know n in advance, we get it using the length function. We would like to range from 0 to the length of the list.

If the range is defined as it is now, where the actual value stops one less than the upper limit then range 0 comma length of l will produce the current range of valid indices. If on the other hand it did, what we would perhaps think is more natural and it will do i, i plus 1 up to j, if this were the case then every time when we wanted to actually range over the list positions, we would have to go from 0 to length of l minus 1. So, we have to awkwardly say minus 1 every time just to remind ourselves that the position stops one short. It mainly for this convenience that we can freely use the length of the list as the upper bound that the list stops, that the range stops at j minus 1.

As I said this is not, I mean, required you could easily define a range function which does the natural thing which is i, i plus 1 up to j, but then you have to keep remembering to put a minus 1 whenever you want the indices to stop with the correct place.

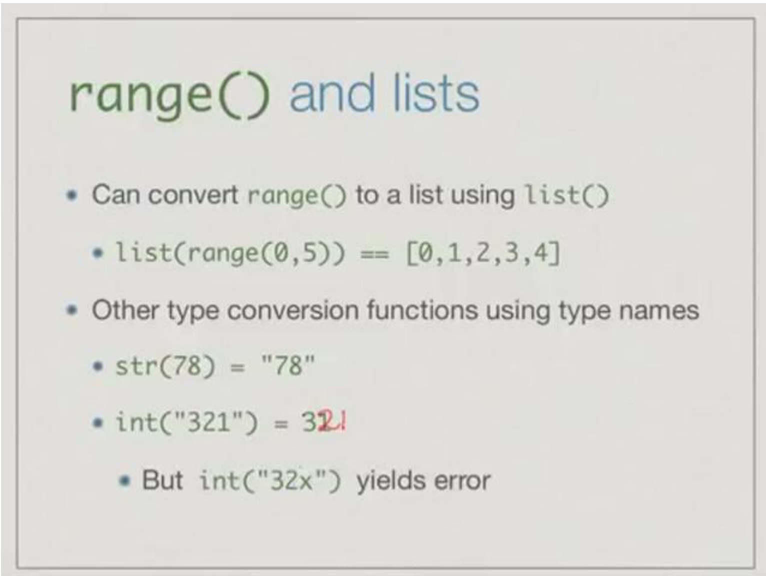
(Refer Slide Time: 05:23)



A range is a sequence and it is tempting to think of range as a list. We saw that for comes in 2 flavors, we can either say for i in a list or we can say for i in range something. So, range 0 to 10 generates a sequence 0, 1 up to 9. So, is this the same as saying for i in the list 0, 1, 2, 3, 4 up to 9. In other words, if we ask Python the following comparison, is range 0 comma 10 equal to the list 0, 1, 2, 3 up to 9 then the question is the result of this comparison true or false.

Here, we encounter a difference between Python 2 and Python 3. In Python 2, the output of range is in fact, the list. So, if you run this equality check, in Python 2 the answer would be true, but for us in Python 3 range is not a list. So, range produces something which is a sequence of values which can be used in context like a 'for', but technically the range is not a list, we cannot manipulate the output of range the way we manipulate the list.

(Refer Slide Time: 06:37)



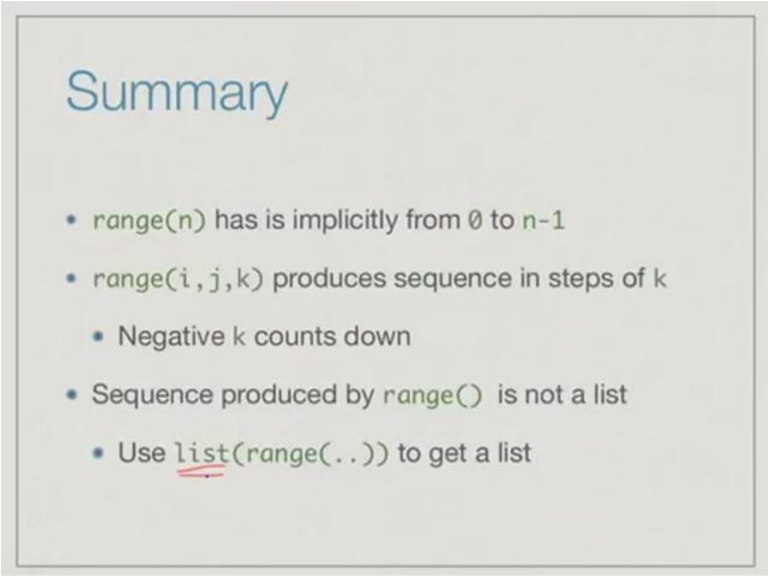

Now, it is possible to use range to generate a list using the function list. So, name of the function is actually list. What we do, for example, is give the range as an argument of list. So, the sequence produced by a range will be converted to a list. If I want the list 0 to 4, I can say give me the range 0 up to 5. Remember, the range was stopped at 4 because 5 is the upper bound and this sequence will be converted to a list by the function list.

This is an example of a type conversion; we are converting the sequence generated by a range, if we said is not a list into a list by saying, make it a list. So, the function list takes something and makes it a list if it is possible to make it a list. If it is something which does not make sense it will not give you a valid value, now this is a general feature. So, we can use the names that Python internally uses for types also as functions to convert one type to another, for example, if we have the number 78 and we want to think of it as a string then the function str will take its argument and convert it to a string. So, str will take any value and convert it to a string representing that value.

This happens implicitly for instance as we will see when we want to display a value using a print function. So, what print will do is take a value, convert it to a string and only strings can actually be displayed, because strings are texts what we see on the screen or when we print out something is text. So, str is implicitly used very often. Sometimes we want to do the reverse we want to take a string and convert it to a value. So, for instance if the string consists only of digits then we should get a value corresponding to that string. If we give it the string 321 then it should give us back the integer 321. So, the value, remember the name of the function is the same as the name of the type to which you want the conversion to be done. So, we want to take a string and make it into an int, we use the name int.

Now, in all these things the function will not produce a valid value if it cannot do so. If I give the function int a string which does not represent the number then it will just give me an error. So, long as a type conversion is possible it will do it, if it does not it returns an error. We will see later on that actually the fact that you get an error is not a disaster there are ways within Python to recover from an error or to check what error it is and proceed accordingly.

(Refer Slide Time: 09:16)




To summarize what we have seen is that our simple notion of range from i to j has some variants. In particular, if we do not give it a starting point we just give it one value it is interpreted as an upper bound. So, range n is a short way of writing 0, 1, 2 up to n minus 1. Also we can use the third argument which is a step in order to produce sequences which proceed in steps i, i plus k, i plus 2 k and so on. In particular, if k is a negative step then we can produce decreasing sequences, and the last thing to remember is though range produces a sequence which can be used exactly like a list in things like for, in Python 3 a range is not a list.

If we want to get a list from a range output we must use this type conversion called list. In general, we can use type names to convert from one type to another type, provided the value we are converting is compatible with the type we are trying to convert to.

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

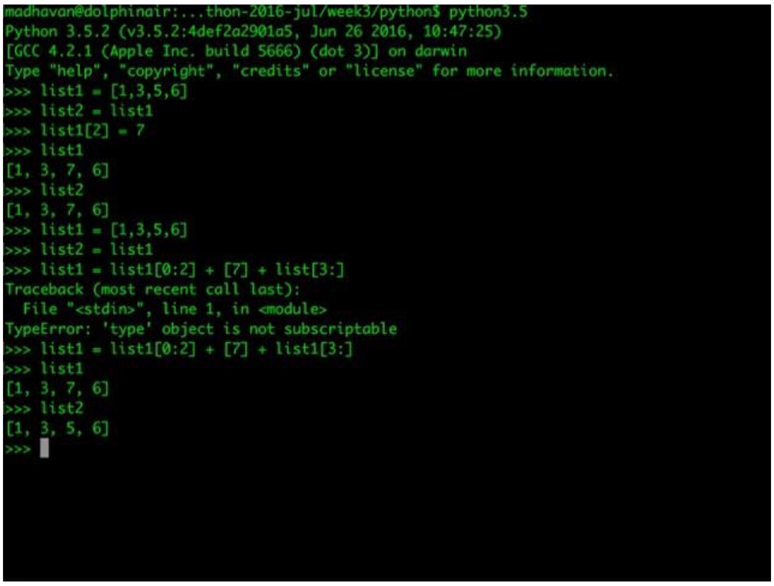
**Week - 03**
**Lecture - 02**
**Manipulating lists**

(Refer Slide Time: 00:02)


So lets take a closer look at lists now. We said that lists are mutable objects. So, if we have a list called list1 whose values are [1, 3, 5, 6], and then we assigned list1 to the list named list2 then we said both list1 and list2 in this case because lists are mutable will be pointing to the same list [1, 3, 5, 6]. Now if I take the position 2 which is this position and replace it by the value 7 then clearly list1 is [1, 3, 7, 6], but because list2 and list1 were pointing to the same object we have that list2 also has the same value [1, 3, 7, 6].

On the other hand, if we made this change in a more roundabout way. So what we did was, we took this list and then we first took its slice 1, 3 from 0 up to position 1 not 2 so I get 1, 3. Then I insert a 7, and then I take from position 3 on wards which is 6. then I also get [1, 3, 7, 6] in list1. But on the other hand because I used plus, what I have done is I have created a new list and therefore list2 has not changed, in this case list2 remains [1, 3, 5, 6]; in other words, concatenation using plus results in producing a new list.

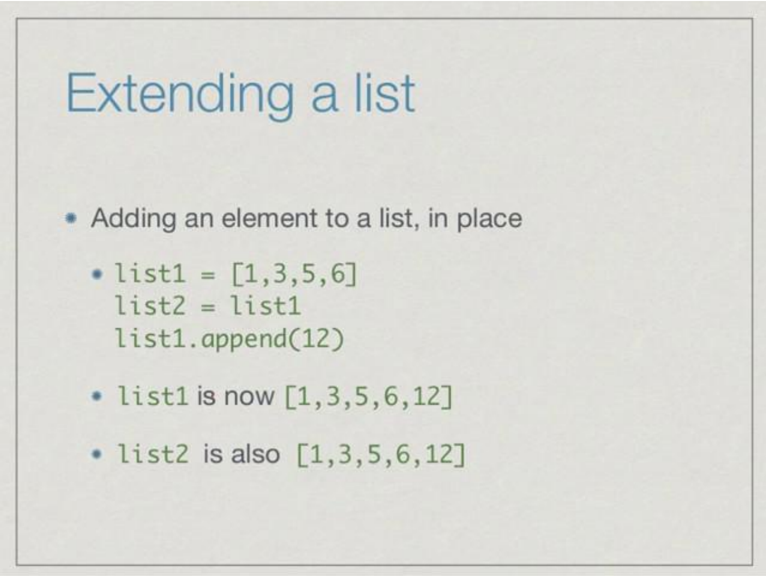
(Refer Slide Time: 01:38)




Let us check this in the python interpreter. So, if I say list1 in to 1, 3, 5, 6 for example, and I say list2 is equal to list1 and then I just change the position 2 of list1 then list1 and list2 are 1, 3, 7, 6. On the other hand if I say list1 is equal to 1, 3, 5, 6 as before and list2 is equal to list1 and now I change list1 in this slice plus concatenation way, so if I say take the first two positions then put a 7 and then take the rest of list1. Now, list1 is again 1, 3, 7, 6, but list2 which was pointing to list1 is no longer pointing to list1 because the plus has created a new list and so the new list is not the same as your old list, so list2 continues to point at the old list so it is 1, 3, 5, 6.

This is an important point that one has to keep in mind regarding mutability. If we start reassigning a list using plus we get a new list, this also applies when we do it inside a function. If inside a function we want to update a function list then so long as we do not reassign it we are OK, but if we put a reassignment using plus then the list that has been updated inside the function will not reflect outside the function. So, we always have to be very careful about this.

(Refer Slide Time: 03:13)



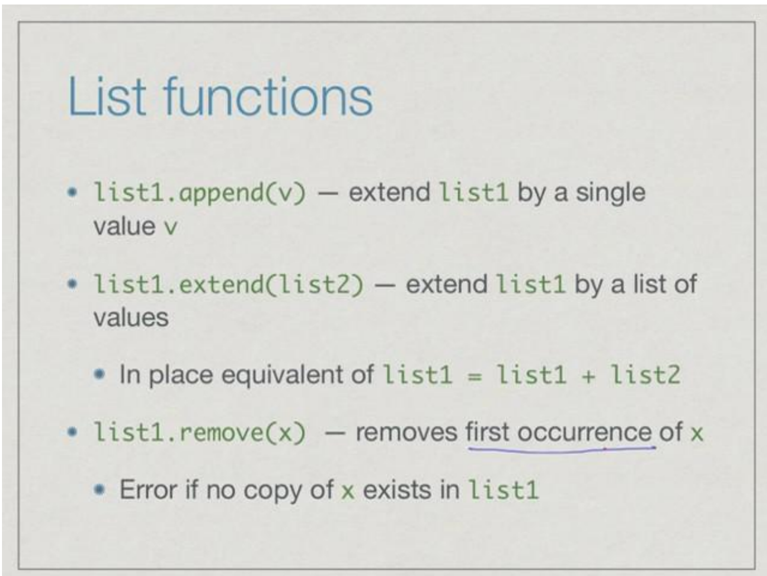

Now, how would be go about extending a list? Suppose, we want to stick a new value 22 at the end of a list; one way to do this is to say l is l plus 22. But as we saw, this plus operator will create a new list, so if you wanted to append a value to a list and maintain the same list so that for instance if it is inside the function we do not lose the connection between the argument and the value will being manipulated inside the function this would not do. We saw this function append in passing when we did gcd in the very first week.

Append is a function which will take a list and add a value to it. So here we have said list1 is 1, 3, 5, 6 as in the previous examples. List2 is list1 and now we have said take list1 and append 12. So, list1 the way we have write it is list1 dot append and in append we give the argument the new value to be appended. So what this does is, of course it will make list1 now a five element list with the original 1, 3, 5, 6 and a new value 12 at the end, but importantly this is the old list1 it is not a new list in that sense. So, list2 has also changed. Append actually adds a value in place both list1 and list2 point to be new list with 12 at the end.

(Refer Slide Time: 04:37)


On the other hand, if we had done it like I mentioned using the plus operator then we would find that list1 changes, but list2 does not because as we saw before concatenation produces a new list. So, append is a function which extends a list with a new value without changing it.

(Refer Slide Time: 04:48)

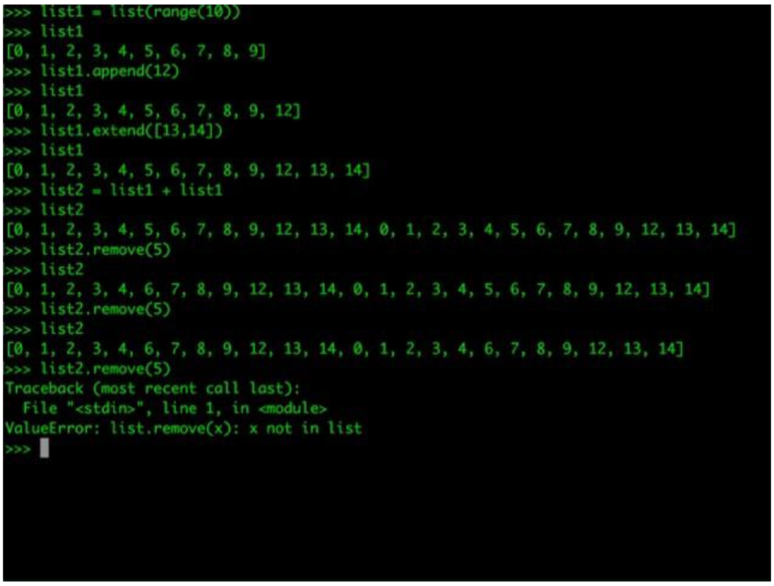


So, append takes a single value. Now, what if we wanted to append not a single value, but a list of values; we wanted to actually take a list and expand it by adding a list at the end, we had say 1, 3, 5 and we wanted to put 6, 8, 10. So, we want to take 1, 3, 5 and we wanted to expand this to have three more values, of course we can append each of these value one at a time. But there is a function which is provided which like append extends a list, but here this must be a list itself.

So extend takes a list as an argument, append takes a value as an argument. So, list1 extend list2 is the equivalent of saying list1 is equal to list1 plus list2, but remember that this must be a list it is not a single value it is not a sequence of value it is a list so it is must be given in square brackets you must give 6, 8, 10 as an argument to the extend function.

Now, this is to add elements to a list there is also a way to remove an element from a list. So, this is one way to remove it by specifying the value. We are not looking at a particular position we are looking for a value x and list1 removes the first occurrence of x in the list. Now, you may ask what happens if there is no occurrence x in the list. Well, in fact this will give us an error so you have to be careful to use remove only if you know that there is at least one copy of x and remember it only removes the very first occurrence, doesnot remove all the occurrences. So, if there are ten occurrences of x in list1 only the very first one will be removed.

(Refer Slide Time: 06:30)

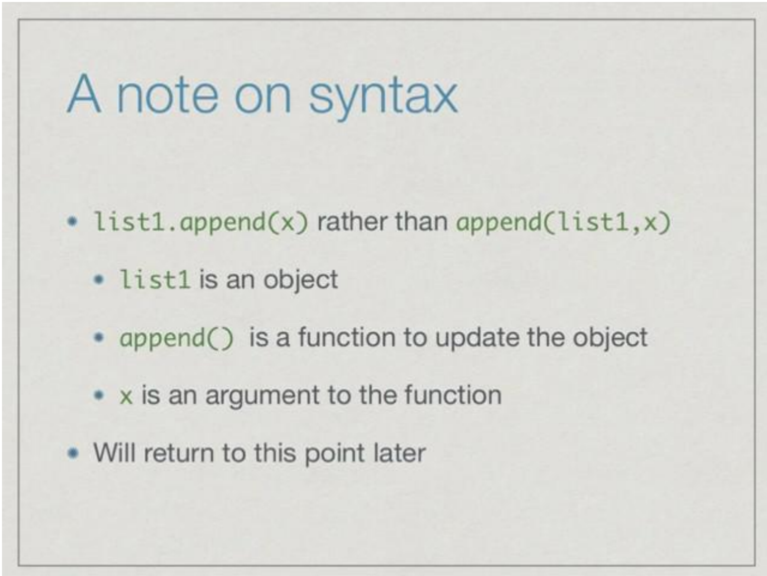



Let us explore these things. Let us start say with list1, so remember from the previous lecture we said we can take range and produce a list. Now if I do this I have list1 is 0, 1, 2 to 9, now if I say list1 dot append 12, then list1 is appended with 12. Now if I say list1 dot extend say 13, 14, then list1 now has 13, 14 appearing. So this is how append and extend work. Now supposing, just for the sake of argument I take list2 and I make two copies of list1. Now, list2 goes from 0 to 14 with a gap of course in between at 10 and 11, again from 0 to 14.

Now if I say list2 dot remove say 5, now there are two copies of 5 remember the first copy which is here in the beginning and second copy which is later, so this will remove the first copy. Now, if I look at list2 the first one skip at 4 to 6, but the second copy is still there. If I say it again then both copies have gone, because I do not have this 4, 6 and I also do not have a 5 here again its 4, 6. Now what happens if I have remove it a third time, now I get an error saying x is not in the list.

Remember that remove works only if x is in the list, if it is not in the list you get an error. Now it is important we will see later that we get an error it also has a name, so it says a value error. This will be useful because later on we will find that within Python we can actually examine errors and take alternative action if an error occurs and we can signal what type of error it is by looking at the value that the error returns.

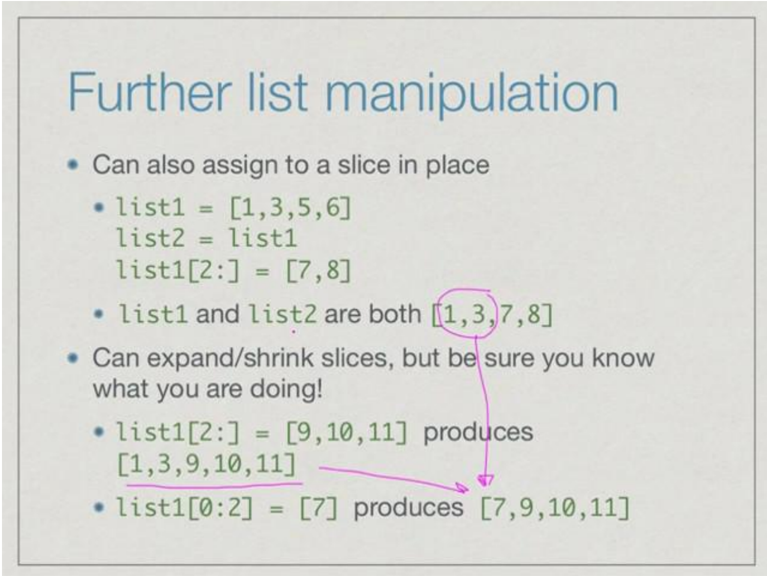
(Refer Slide Time: 08:18)




The append function looks a little bit different from the other functions we have seen so far. We would normally expect the function append to take two arguments; the list and the value to be appended. So we would think that the correct way or the natural way to write append would be to say append to list1 the value x. On the other hand what we have is this funny notation it takes says to list1 apply the function append with value x. In a Python terminology list1 is an object and append is a function to update the object and x is supposed to be an argument to the function append.

In such a situation we have an object and we then apply a function to it, so we use three functions attached to the object by using the dot notation rather than passing the object to the function which is a more normal way in which you think of functions. We will come back to this point later on and may be two - three weeks from now and we look at what is called Object Oriented Programming within Python.

(Refer Slide Time: 09:28)




There is another way to expand and contract lists and place, this is by directly assigning new values to a slice. So, we go back to our old example: list1 is 1, 3, 5, 6 and list2 is list1. Now what we are saying is that take the slice from position 2 on wards and assign it the value 7, 8. So remember the positions are 0, 1, 2, 3. So what this is saying is, take this slice namely 5, 6 and replace it by 7, 8. What we get is that, of course list1 is the slice 5, 6 is replaced by 7, 8, but this slice replacement happens in place.

It is a bit like assigning a new value at a given position. If I say list 2 is equal to 7 we said that position two becomes 7. In the same way if I say that list1 from slice two to the end become 7, 8 it changes 5, 6 to 7, 8 both in list1, but it also does not change where it is pointing to, so list2 also get affected. So both of them now say 1, 3, 7, 8.

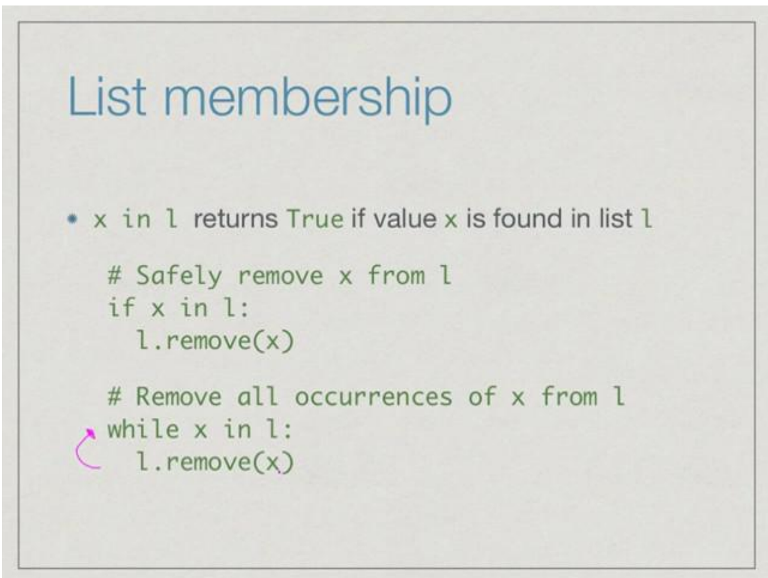
Now, here we had a slice of length two and we replaced it by a new list of length two. So, we preserved the structure of the list in terms of the number of positions. This is not required, Python allows you to both expand and shrink a slice. For instance, you could have taken that list, now let us say we have this is 1, 3, 7, 8 and again we want to take slice 2 onwards which has two positions and we can say replace it by a list with three values. We are saying take this list take the slice from 2 to 3, the last two positions and replace it with three values and what we get is the old 1, 3 and this slice has now become

9, 10, 11. So, we had a four element list become a five element list. This is the one way to expand a list in place using a slice.

The other thing we can do is shrink a list; we can put a smaller thing. Supposing, we want the list to have just one value in the position 0 on 1, so we take the slice 0 to 2 which will give us these two positions so now you have a slice of length two, but we assign it a list of length one. So, this 1, 3 is replaced by just the single 7. Now we had a list of length five after the previous expansions which has now become a list of length four after this contraction.

With slices you can replace a slice in place, this can produce a bigger list or a smaller list depending on what you put in, but as you can imagine this can be very confusing. So, you should be very careful that you know what you are doing if you are trying to directly updates slices in the list.

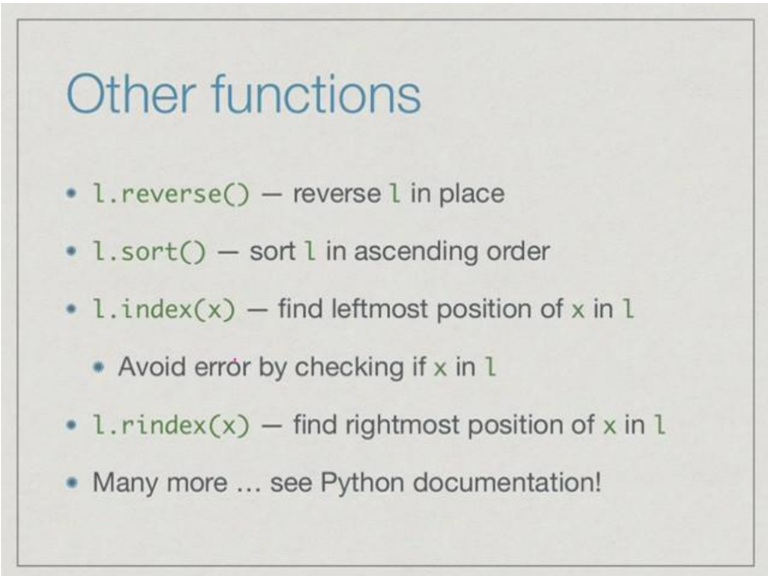
(Refer Slide Time: 12:07)




One of the very common things that we want to know about a list is whether a value exists in a list. So, Python has a very simple expression called x in l. So, x in l returns true if the value x is found in the list l. Now we can use this for instance to make our remove a safe operation; before we invoke l dot remove x we first check that x actually is in l. So, if x is in l then the condition will be true and only then will we try to remove it, if x is not in l then we would not remove x. In this case we are guaranteed that l dot remove will not be called in an error prone context where it will say there is no x in l.

Also recall that remove removes only the first element. We can replace this if by a while and say that so long as there is a value x in l keep applying remove. This will in one short remove all the x's in l because every time we remove an x we go back and check if there is still an x in l if there are still on x in l we remove it, so from left to right this loop will remove all the x's in l.

(Refer Slide Time: 13:14)

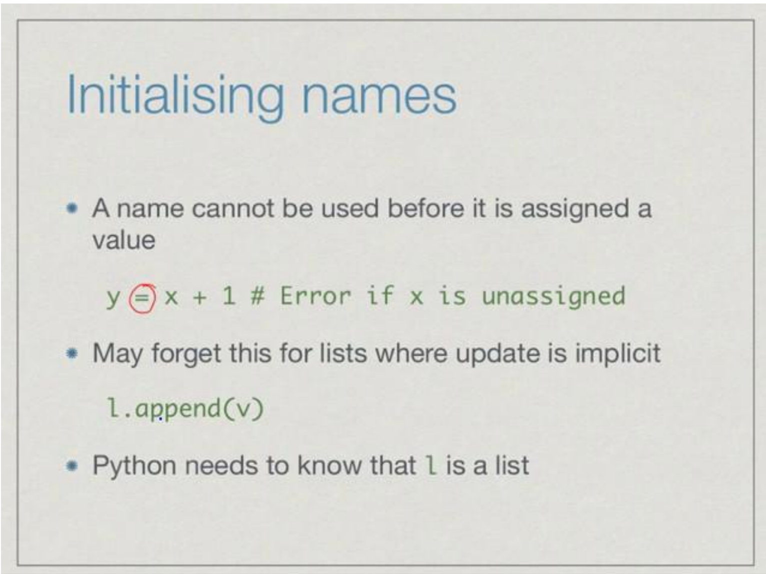



Now there are a host of other functions defined for list, for instance l dot reverse will reverse a list in place, l dot sort will sort a list in ascending order. You can also sort it in other orders you can look up and see how to do that. If we only want to know whether an element is in l we say x in l, but if we want to know where it occurs then we use index it will find the leftmost position, but again it will give us an error if there is no x in the list, so we should first check if x in l and then find the index of the leftmost position.

Now you might want not the leftmost or the rightmost position so there is an r index and there is a host of other functions and you must look up the Python documentation there is no way that this course or any course can cover every function which is defined in Python for every type.

So you do have to look up the documentation and if you think that there should be a function that which does something natural very often there will be. So, try and look it up and see for yourself how it works and try to use it. If you have a question like what happens if I do this well Python is an interactive language. What happens if I do this? Just try it out and see and try to figure out from what you see in the interpreter, how the function works, in case there seems to be some ambiguity in the documentation. But above all do not be afraid to see in documentation only by looking up a documentation will you be able to learn the functions that you need because it is very difficult as I said to say up front every possible function that is there.

(Refer Slide Time: 14:48)



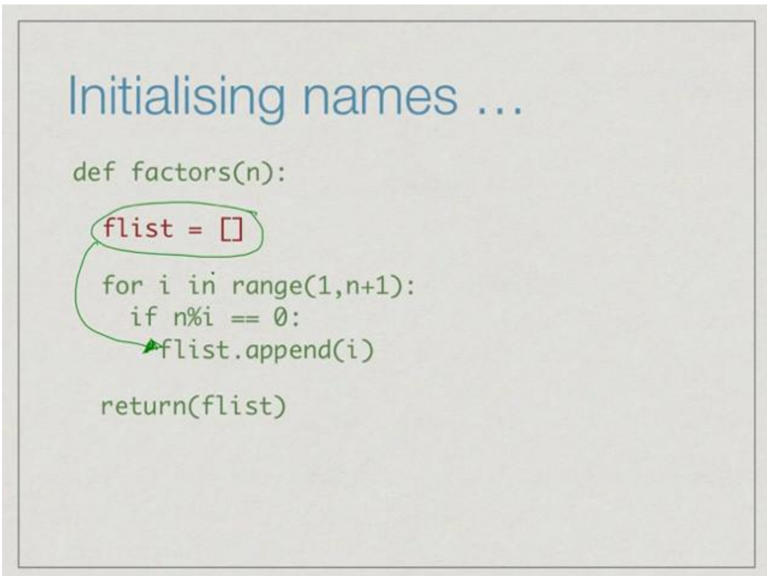

Final point regarding list is something we talked about in passing, which is that since names do not have types in Python, we do not have to announce the name, names just pop up as the code progresses. So every time a name pops us Python needs to know what value it is. Typically the first time we use a name we have to put it as part of an assignment, we have to assign a value to it and that value has to be something which is computable given the current names. So if we want to assign for instance to the name y the expression x plus 1, at this point implicitly x must have a value, otherwise the x plus

1 cannot be evaluated.

So, if x has not been seen before and for the first time in my code I see it on the right hand side of an assignment it means that I am expected to produce a value for x but no value has been assigned so far and this will give you an error. This is quite easy to spot, so when you write something and you see something on the right hand side and you have not seen it before then it means a Python will flag an error and it is not very difficult to understand why this is so.

Now the kind of list functions we saw now, it is bit more subtle. When I say l dot append v there is no equal to sign. So it is not immediately obvious that l dot append v requires l to already be having a list value, why cannot I just append v for example to an empty list. Well, of course I can append v to an empty list how does Python know that l is in empty list. So, python needs to know that l is a list, before it can apply this append function.

(Refer Slide Time: 16:21)

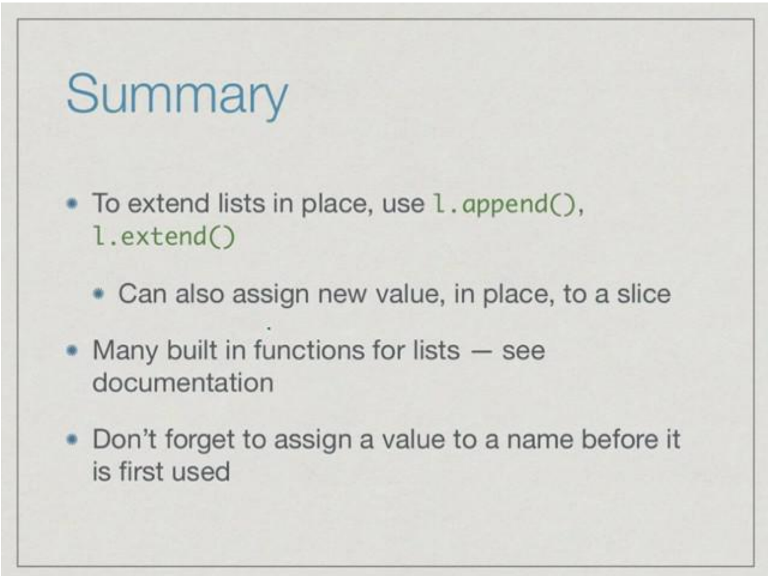



So, we saw this small function earlier which computes factors of n. So essentially what it does is, it takes all numbers in the range 1 to n. I take 1 to n plus 1 so that I run through the sequence 1 to n. And if a number divides n evenly if there is no remainder I have used the append function now to append i to the list of factors which I will return. Now, the catch with this is that when I come for first time to this statement the first factor which will be 1 of course because 1 will have always be a factor.

Python will have to ask why flist has the ability to append a value, because flist has never been encountered to this point. We were careful when we wrote the code, of course we used plus because we did not use append in that code but it is the same thing. We have to be careful to insert this initialization. This initialization is only needed to tell Python when this first append happens that it is indeed the case that flist is of type list and therefore the append function is a valid function to apply to this name, without this you will get an error.

Just remember that you always have to make sure that every name that you use is initialized to a value the first time, so that whenever it appears later on, the value is clear and therefore what operations are allowed for this name are also clear to Python.

(Refer Slide Time: 17:47)



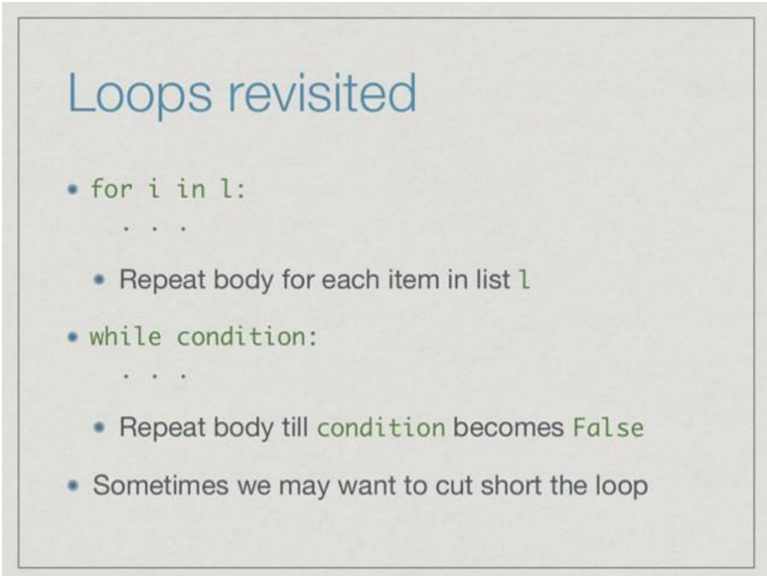

To summarize, what we saw is that we can extend lists in place using functions like append, extend and so on. We can also assign a new value in place to a slice of a list and in the process expand or contract the list, but this is something to be done with care; you must make sure you know what you are doing. There are several built in functions on list; we will see some of them as we go along and use them and describe them as we see them, but it is impossible to document all of them and to go through all of them and it is also a very boring to just list out of a bunch of functions.

So, do look up the tutorial and other documentation which is available which I mentioned in the earlier weeks, so that you can find out what kind of functions are available. And finally, do not forget that you must assign a value to a name before it is first used otherwise, because names do not themselves have types, Python will not know what to do with the given name.

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

**Week - 03**
**Lecture – 03**
**Breaking out of a loop**

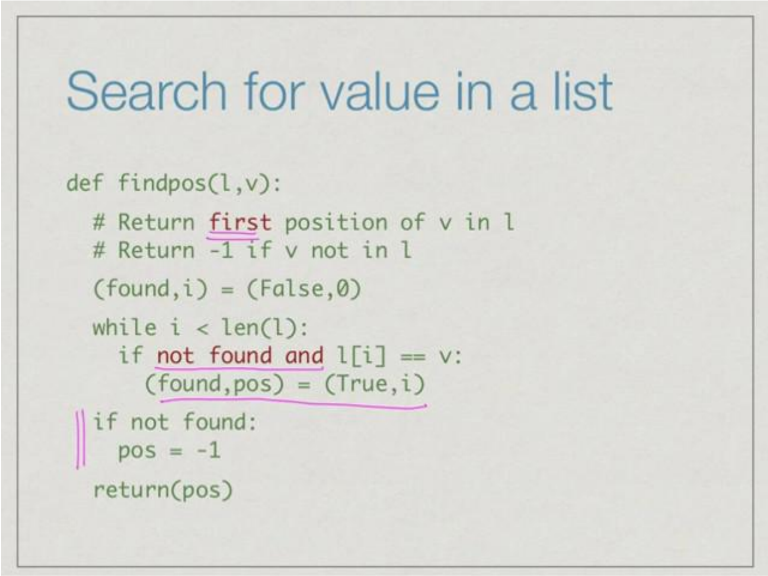
(Refer Slide Time: 00:01)




Let us revisit Loops. So, we saw that we have two types of loops in Python. The first is a 'for loop' where the value ranges over a list in general or say the output of the range sequence; so for i in l will repeat the body for each item in the list l.

Sometimes we do not know how many times we need to repeat in advance, so then we use the while forms. While takes a condition and it keeps repeating the body until the condition becomes false. But the main thing to notice in both these cases is that how long will the loop takes is determined in advance either we have a fixed number of iterations depending on number of values in l for a for loop or we keep going until the condition becomes false we cannot stop before that. Now it does turn out the sometimes we have natural situations where we want to cut short the loop and not proceed till the very end.

(Refer Slide Time: 01:00)




Suppose, we want to write a function findpos which finds the first position of a value v in the list 1. We had seen that there is a built in function called index which pretty much does this, so l dot index v will report the first position where v occurs in the list l.

However, the index function will report an error if there is no occurrence of v in l. So, we want to change that slightly and what we want to say is that if there is no v in l report minus 1. So either we get a value between 0 to n minus 1, where n is the length of the list or we get the value minus 1 saying that v is not in the list. Here is a while loop implementation. We use a name found as the name suggest to indicate whether the value have been found or not as we have been seeing so far. Initially we have not seen any values in l, so v is, the found - is false. And we use i as a name to go through the positions, so i is initially 0 and we will increment i until we hit the length of l.

So, so long as, while i is less than the length of the l, if we see the value we are looking for then we update found to be true and we mark the position that we have found it to be pos. At the end of this, if we have not found the value, so the value found is not been set to true that means there was no v in the list then we will set pos to minus 1 which is the default value we indicate at beginning. Then we return the current value of pos which is either the value of pos we found or the value will set to minus 1 because we did not find it.

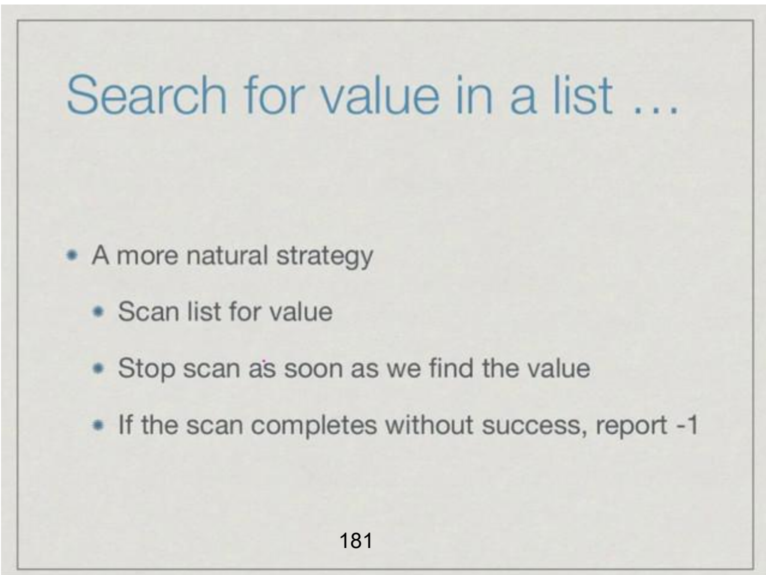
There are two points to observe about this; the first point which is the main issue at hand is that we are going to necessarily go through every position i from 0 to the length of l minus 1 regardless of where we find it. Supposing, we had several hundreds of thousands of elements in our list and we found the value at the very second position we are still going to scan the remaining hundreds of thousands of positions before we return the position two. Now this seems very unnecessary.

The other point, which is an issue of correctness, this is an issue of efficiency that we are running this loop too many times unnecessarily. In case we are actually able to report the first value quite early. The other problem which is correctness is that we want to report the first position, but the way we have written it every time we continue pass the first position and we find another copy of v we are updating the position to be the new thing.

So, we are actually finding not the first position but the last position so this is not a very good way to do this. So, we first change that. So we say that we want the first position. So, we want the first position we want this update to happen only if we have not found the value so far We had this extra condition which says that if l i is v and we have not founded so far then we update found to true and we said pos to i.

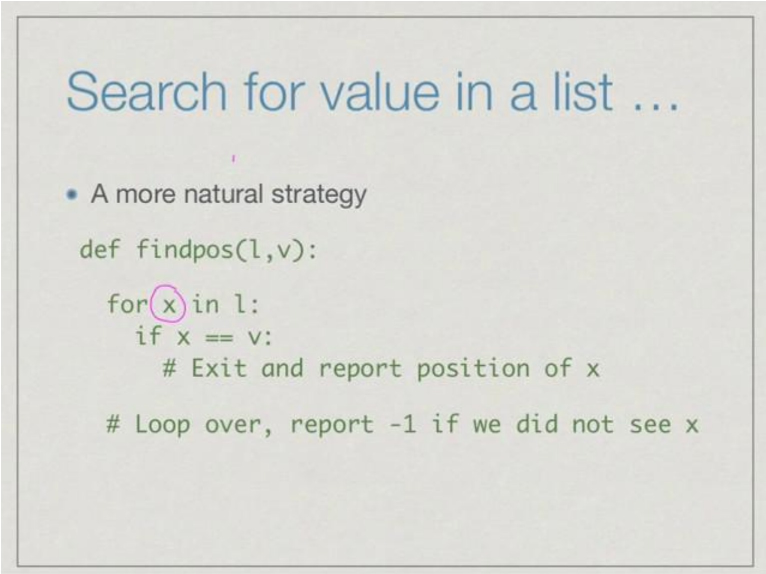
This ensures that pos is updated to i only the very first time we see v after that the value found prevents us from doing this update again. So, at least we are correctly finding the first value of the position. And finally as before if we never find it then the value found is never set to true and so we report minus 1. Now the issue is why we have to wastefully go through the entire list even though after we find the very first position of v in l we can safely return and report that position.

(Refer Slide Time: 04:32)



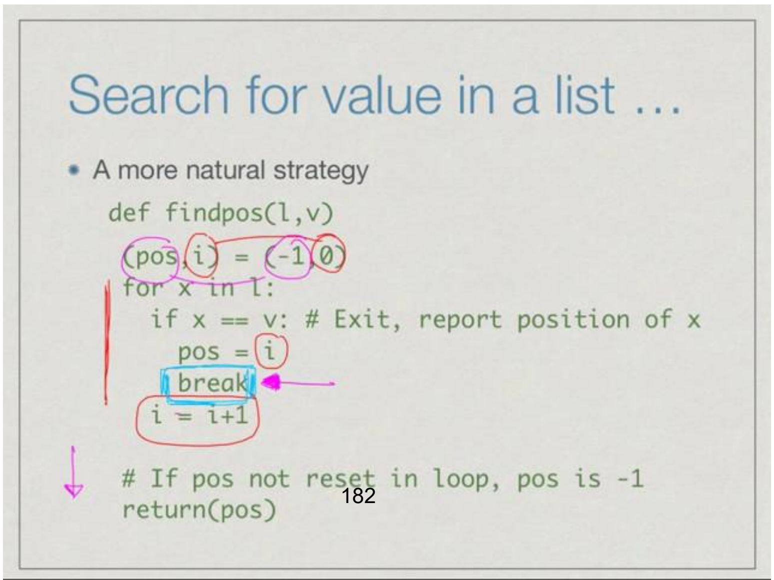
So, a more natural strategy suggests the following; we scan the list and we stop the scan as soon as we find the value. So, whenever we find the value for the first time we report that position and we get out. Of course, the scan may not find the value, so if the scan completes without succeeding then we will report minus 1.

(Refer Slide Time: 04:58)




We could think of doing this using a 'for loop'. So, we go through every value in l using a variable x. So, x is the name which goes through the first element, second element, these are not positions now these are the actual values. We check whether the value x that we currently pick up in the list is indeed the value v we are looking for, if so we exit and we report the position of this x. If we come to the end of the loop and we have not seen x so far like before we want to report minus 1.

(Refer Slide Time: 05:30)



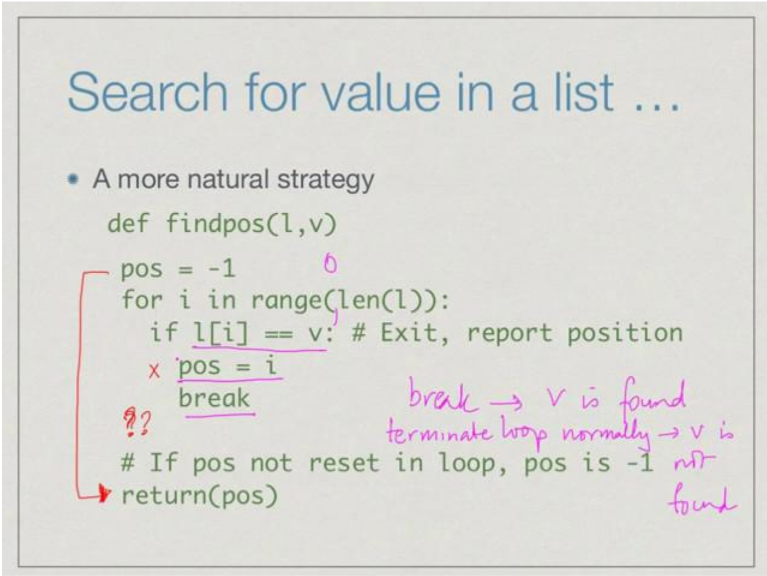
Here is a first attempt at doing it, so let us go through it systematically. First of all we have this loop, so for each x in the list if x is v then we report the position. Now we have to find the position because we are going through the values so this has forced us to use a name i to record the position and we have to manually do this. So, like in the while loop we start with the position 0 and then every iteration we increment the position. This is only the first version of this we will see how to fix this. So, we have to separately keep track while we are doing this for, kept, separately keep track of the positions and report it.

But what is new and this is a main issue to be highlighted here is this statement called break. So what break does is it exits the loop. So this is precisely what we wanted to do if x is v we have found the first position we do not want to continue we just want to break, if not we will go increment the position and go back. Now, how do we record at the end we do not have this found variable anymore. How do we know at the end whether or not we actually saw it? So, the question is was pos set to i or was it not set to i. Well, the default value is minus 1. Supposing, pos is not reset we want to report minus 1. So, this is why in our function we have actually initially set pos to be minus 1. So, the default value is that we do not expect to find. It is like saying found is false. So, by default the position is minus 1.

At the end of this loop if you have not found it pos has not been reset so it remains minus 1. On the other hand if we have found it without looking at all the remaining elements we have set it to the first position we found it and we have taken a break statement to get out of the loop so we do not unnecessarily scan. When we come here either way we have either report it to first position we have found it or in the worst case we have scanned through the entire list, and we have not found it in which case the initial value minus 1 is there. So, in any case we can return pos and we have no problem

This is just to illustrate the use of the word break, which allows us in certain situations to get out. Now remember in the worst case we do not find x in it. So, the worst case behavior of this loop is no different from the situation without the break we have to go through the entire list and we have to come out only when we have scanned everything, but if we do find it we can avoid some unnecessary computations.

(Refer Slide Time: 08:10)

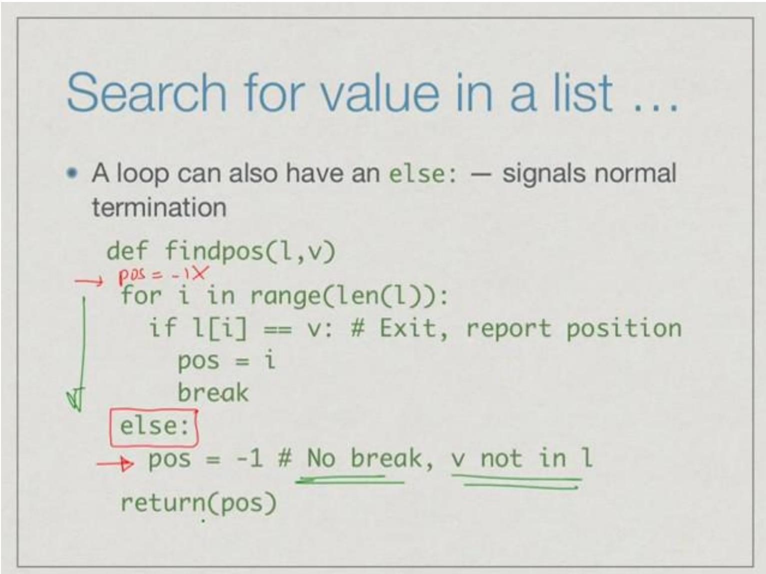



We can simplify this a little bit by first removing this i instead of scanning x actually it is better to scan the positions. So, it is better to say pos is minus 1, but instead of going through x in l it is better to go through i in the range 0. Remember now this is implicitly 0 to the length of l minus 1. So, we go for i in the range 0 to length of l, so if we do not give a 0 it will give only a one argument this is taken as the upper bound. This will go through all the positions. And instead of checking x we check l at position i, so if l a position i is v then we set the position to this current value and we break. So, by changing the variable that we put in the 'for', we have got a slightly more natural function.

And again, we have this clever trick which says that since we set pos initially to minus 1 if we did not reset it here if no value in the range 0 to n minus 1 produced a list value which is equal to the value we are looking for we will return minus 1 as before. This requires this clever trick. So the question is what if we do not have a situation where such a clever trick is possible or if we do not think of this clever trick how would we know at this point. So remember now there are two situations either we break, so if we break that means that the value is found, or if we do not break, if we terminate normally, if the loop ends by going through all the things then v is not found.

Remember even if v is found at very last position we will first break, we will not go back and say that the loop ended. So, if we see v at any position from beginning to end we will execute the break statement if not we will not. The question is can we detect whether or not we broke out this loop or whether we terminate it separately.

(Refer Slide Time: 10:06)

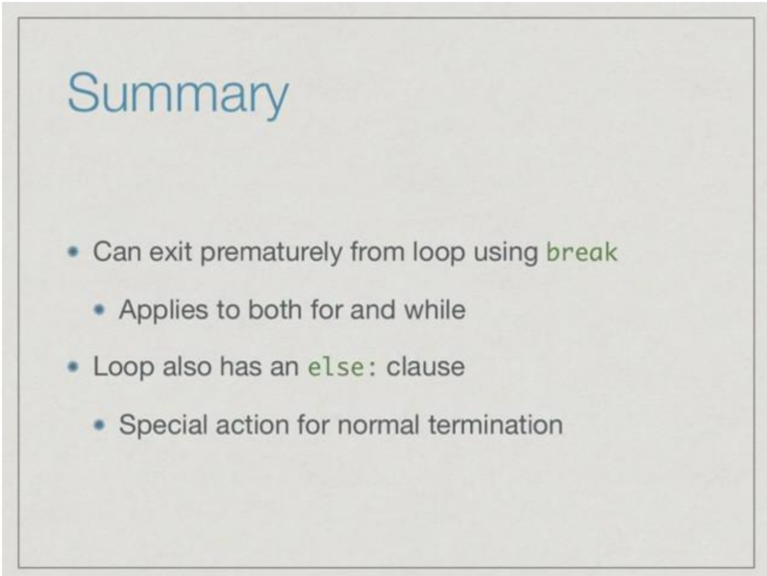



Python is one of the few languages which actually provide way to do this. So it allows something that looks a bit odd because of the name it allows something called else which we saw with if, it allows an else for a loop as well. The idea is that else this part will be executed if there is no break if the loop terminated normally, so do not worry about the fact that else does not mean this in English so it is just a way of economizing or the number of new words you need to use. If you see an else attached to a 'for' it could also be attach to a 'while' it means that the 'while' or the 'for' terminated in the natural way either for iterated through every value that it was supposed to iterate through or the while condition became false. On the other hand aborted by a break statement then the else will not be executed.

Here for instance, now we do not initialize. We no longer have this clever trick, we do not have this anymore. So what we say is that for i this range we check and if it is there we set the position to be the current i and then we break.

Now, if we have actually gone through the entire list and not found it pos is undefined. If pos is undefined we need to define it before we return the value, so we have this else statement. Now we say, we have come through this is whole thing and there will be no break, there is no v in l because otherwise we would have done a break and otherwise pos will be set to a valid value in the range 0 to n minus 1. So, there has been no break there is no v in l. Let us say pos to minus 1 and then return it.

(Refer Slide Time: 11:44)

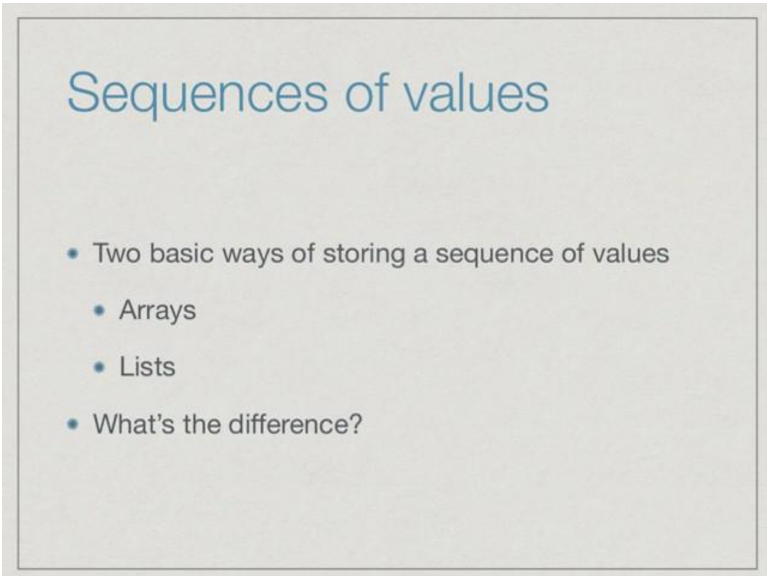



To summarize, it is very often useful to be able to break out of a loop prematurely. We can do this for both for and while we can execute the break statement and get out of a loop before its natural termination. And sometimes it is important to know why we terminate it, we terminate it because the loop ended naturally or because we used the break statement. This can be done by supplying the optional else. Both, for and while also allow an else at the end, and the statement within else is executed only when loop terminates normally.

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

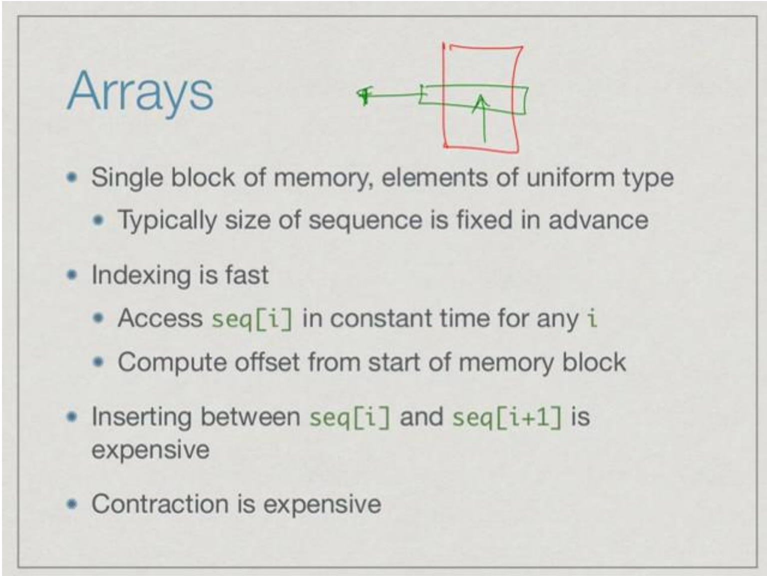
**Week – 03**
**Lecture – 04**
**Arrays vs. Lists, Binary search**

(Refer Slide Time: 00:02)




We have seen several situations where we want to store a Sequence of values. Now it turns out that in a program or in a programming language implementation, there are two basic ways in which we can store such a sequence. These are normally called Arrays and Lists. So, let us look at the difference between Arrays and Lists.

(Refer Slide Time: 00:22)




An array is usually a sequence which is stored as a single block in memory. So, you can imagine if you wish that your memory is arranged in certain way and then you have an array, so usually memories arranged in what are called Words. Word is one unit of what you can store or retrieve from memory, and an array will usually be one continuous block without any gaps.

And, in particular this would apply when an array has only a single type of value, so all the elements in the sequence are either integers or floats or something where the length of each element of the array is of a uniform size. We would also typically in an array no in advance how big this block is. So we might know that it has say 100 entry, so we have a sequence of size 100.

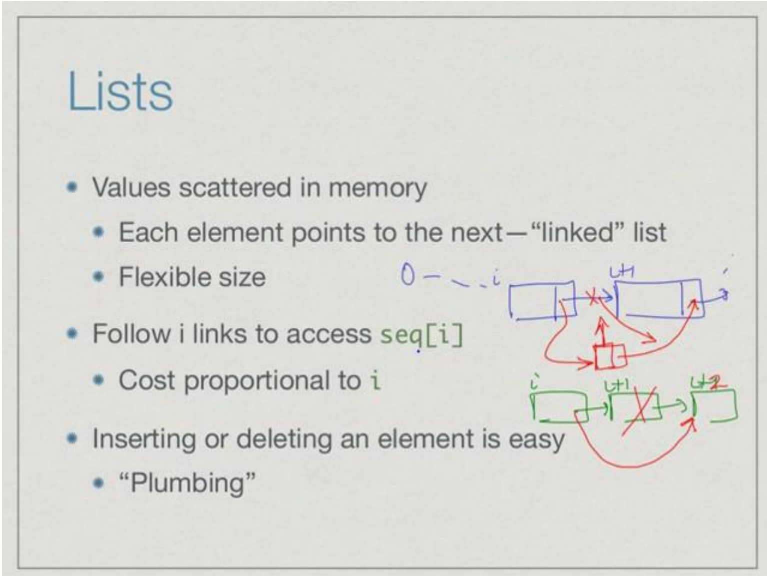
Now when this happens, what happens is that if you want to look at the jth element of a sequence or the ith element of a sequence, then what you want to think of is this block of memory starting with 1, 2, 3, up to i right and you want to get to the ith element quickly. But since everything is of a uniform size and you know where this starts, we know where the sequence starts you can just compute i times this size of one unit and quickly go and one shot to the location in the memory where the ith element is saved.

So, accessing the ith element of an array just requires arithmetic computation of the address by starting with the initial point of the array and then walking forward i units to the ith position. And this can be done in what we could call Constant time. By constant time what we mean is it does not really depend on i. It is no easier or no difficult to get the last element of an array as it is to get to the second element of an array, it is independent of i. It takes the fixed amount of time to get to sequence of y for any i.

Now, one consequence of this is inserting or contracting arrays is expensive, because now if i have an array with 0 to 99 and I want to add a new value here say at position i then first of all this array now becomes from 0 to 100 and now everything which is after i has to be shifted to accommodate space if we want to keep the same representation with the entire array is stored as a single block. So, when we have a single block of memory though it is efficient to get to any part of it quickly it is not very efficient to expand it because we have to then shift everything. The worst case for example, if this green block comes into 0th position then the entire array has to be shifted down by one position.

In the same way contraction is also expensive because we have to make a hole in some sense. If we remove this element out then we have a hole here and then we have to push everything up to block this hole, because – remember the array must have all elements contiguous that is without any gaps starting from the initial position.

(Refer Slide Time: 03:27)




The other way of storing a sequence, is to store it one element at a time and not bother about how these elements are with respect to each other in the memory. I can think of this memory as a large space and now I might have one element here, so this is my first element and then I will have a way of saying that from here the next element is somewhere else, this is what we call a Link. So very often in the implementation these are called linked list, so I may have the first element here. Now because of various reasons I might end up putting the second element here and so on.

You can imagine that if you have some say space in your cupboard and then you take out things and then you put things back but you put things back in the first place where you have an empty slot, then the sequence in which you put thinks back may not respect the sequence in which they appear finally in the shelf. So, here in the same way we do not have any physical assumption about how these elements are stored, we just have a logical link from the first element to the next element and so on.

The other part of this is that we do not have to worry about the overall length of the list because we know we started at the 0th position and we keep walking down. On the last position so say suppose the last position is in fact two then there would be some indication here saying that there is no next element, so two is the last element. A list can have a flexible size and obviously because we are just pointing one element to another, we can also accommodate what we see in Python where each element of the list maybe of a different type and hence each value might have a different size in itself. It is not important unlike an array that all the values have exactly the same size because we want to compute how many values to skip to get to the ith element. Here, we are not skipping we are just walking down these links.

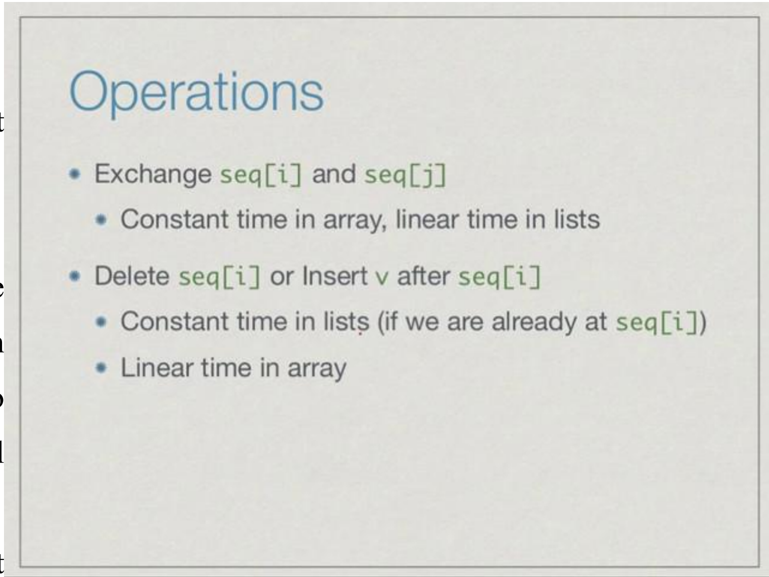
Since we have to follow these links the only way to find out where the ith element is is to start from the 0th element and then go to the first element then go to the second element and so on, because a priori we have no idea where the ith element is. So, after i steps we will reach the ith element. And if we have a larger value of i it takes longer to get there. So accessing the ith position in a sequence when the sequence is stored as a list takes time proportional to i, we cannot assume that we can reach any position in the list in constant time unlike in an array.

On the other hand it is relatively easy to either insert or delete an element in a list like this. Supposing, we have a list like this. Suppose, we start at 0th position and may come to the ith position and currently if we say that the ith position points to the i plus 1th position which point to the rest, and suppose we want to insert something here, then it is quite simple we just say that this is the new i plus 1th position. We create a new block in memory to store this value and then we will make this point here. So, it is like plumbing, we remove one pipe and we attach a pipe from the ith element to the new element and attach another pipe to the new element to what was beyond the ith element previously.

We just have to shift these three links around and this does not matter wherever we have to do it, any place in the list if we have, I have just have to make this local change in these links. And so this insertion becomes now a constant time operation if we already are at the position where we want to make the change. In the same way if we want to delete something that is also easy in fact it is even easier. So, I have say i pointing to i plus 1 pointing to i plus 2 and I want to remove this element, well then I just make this link directly point to the next one. Remember all these links are available to us we know this link we know this link, so we know where i plus second element is.

Similarly here, when we want to create a new element we get a link for it because we create it and we know what link to copy there because we already have it here. So we can copy it from the ith element to the new element. Therefore, in a list it is expensive to get to the ith element it takes time proportional to the position we are trying to get to, however, having got to a position inserting or deleting an element at that position is of constant time. Unlike in an array, where if we insert or delete at some position we have to shift a lot of values forwards or back words and that takes time.

(Refer Slide Time: 07:47)

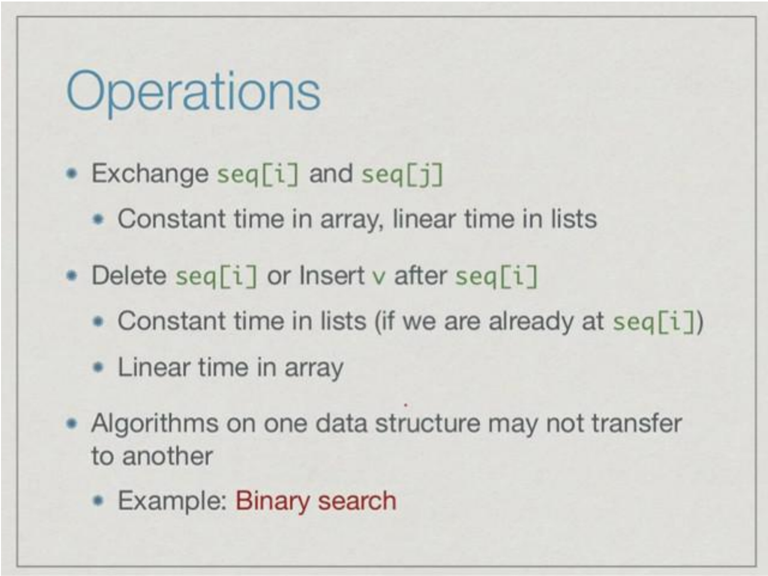



Let us look at typical Operations that we perform on sequences. So one typical operation, now if i just represent a sequence more abstractly as sequences we have been drawing it. Supposing, I want to exchange the values at i and j. This would take constant time in an array because we know that we can get the value at ith position, get the value at the jth position in constant time independent of i and j and then we exchange them it just involves coping this there and the other one back.

On the other hand in a list I have to first walk down to the ith position and then walk down to the jth position to get the two positions so I will have in a list I would have the sequence of links and then I would have another sequence of links. Then having now identified the block where the ith value is and the block where the jth values then i can of cause exchange them without actually changing the structure I just copy the values back and forth, but to find the ith and jth values it takes time proportional to i and j, so it takes linear time.

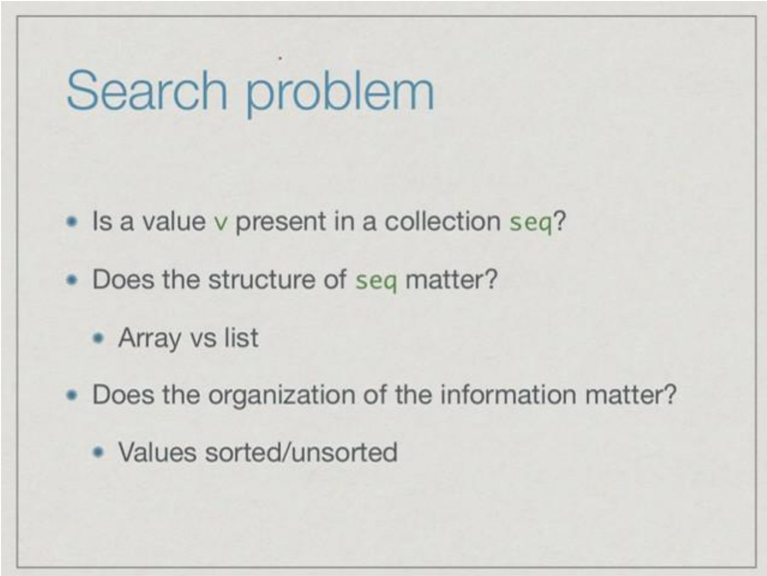
On the other hand as we have already seen, if you want to delete the value at position i or insert the value after position i this we can do efficiently in a list because we just have to shift some links around, whereas in an array we have to do some shifting of a large bunch of values before or after the thing and that requires us to take time proportional to i.

(Refer Slide Time: 09:12)




The consequence of these differences between the two representations of a sequence as an array and a list is that we have to be careful to think about how algorithms that we want to design for sequences apply depending on how the sequence is actually represented. An algorithm which works efficiently for a list may or may not work efficiently for an array and vice versa. To illustrate this, let us look at something which you are probably familiar with at least informally called Binary search.

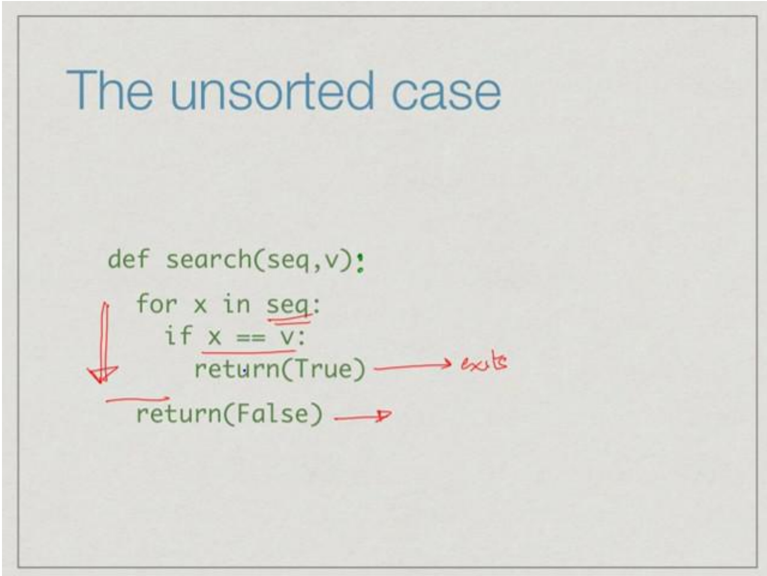
(Refer Slide Time: 09:42)




The problem we are interested in is to find out whether a value v is present in a collection or we can even call it a sequence to be we more precise in a sequence which we call seq. So, we have a sequence of values we want to check whether a given value is there or not. For instance, we might be looking at the list of roll numbers of people who have been selected for a program you want to check whether our roll number is there or not.

There are two questions that we want to ask; one is is it important whether the sequence is maintained as an array or as a list and is it also important given that it is maintained as an array or a list whether or not there is some additional information we know for example, it is useful for array to be sorted in ascending order that is all the elements go in strictly one sequence from beginning to end, lowest to highest, or highest to lowest, or does it matter, does it not matter at all.

(Refer Slide Time: 10:37)



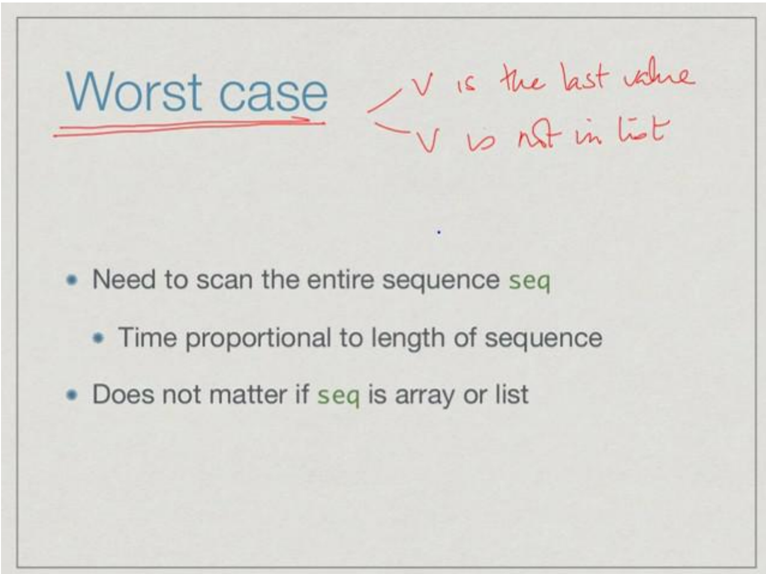

Here is a very simple Python program to search for a value in a unsorted sequence. This is a similar to what we saw before where we are looking for the position of the first position of a value in a sequence, which is we not we do not even need the position we only need true or false, is it there or is it not, it is a very simple thing. What we do is we loop through all the elements in the sequence and check whether any element is the value that we are looking for.

Once we have found it we can exit, so this exits the function with the value true. And if we have succeeded in going through the entire list, but we have not exited with true that means we have not found the thing, so we can unambiguously say after the for that we have reached this point we have not found the value v that we are looking for and so we should return false.

Since we are not looking for the position we have much simpler code if you go back and see the code we wrote for findpos, so there we had first of all keep track of the position and check the value at position i rather than the value itself. And secondly, when we finish the loop we had to determine whether or not we had found it or we had not found it, whether we had remember we use the break to get out of the loop for the first time we found it.

We used to detect whether we broke or not, if we did not have a break then we had found it, if we did not had a break we did not find it. Accordingly either the value of pause was set or it was not set and if it is not set we should make it minus 1. So that was more complicated, this is very simple.

(Refer Slide Time: 12:07)

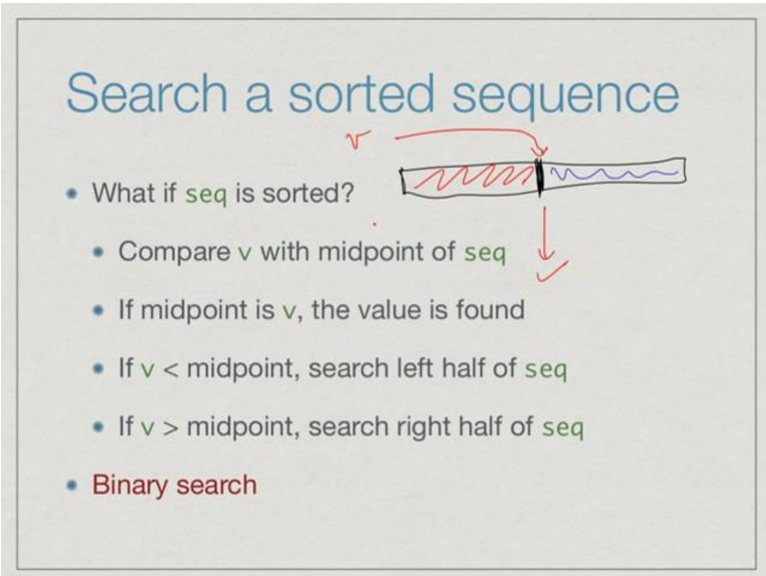



The main point of this function is that we have no solution to search other than to scan from beginning to end. The only systematic way to find out v occurs in the sequence or not is to start at the beginning and go till the end and check every value, because we do not have any idea where this value might be. This will take time in general proportional to the length of the sequence.

We are typically interested in how long this function would take in the worst case. So what is the worst case? Well, of cause one worst case is if we find the value at the end of the list. So, v is the last value then we have to look at all. But more generally v is not in the list. v is not in the list the only way we can determine the v is not in the list is to check every value and determine that that value is not found.

And this property that we have to scan the entire sequence and therefore we have to take time proportional to the sequence to determine whether v is in the sequence or not it does not matter if the sequence is an array or a list, whether it is an array or a list we have to systematically go through every value the organization of the information does not matter. What matters is the fact that there is no additional structure to the information, the information is not sorted in any way at no point can we give up and say that since we have not seen it so far we are not going to see it later.

(Refer Slide Time: 13:26)



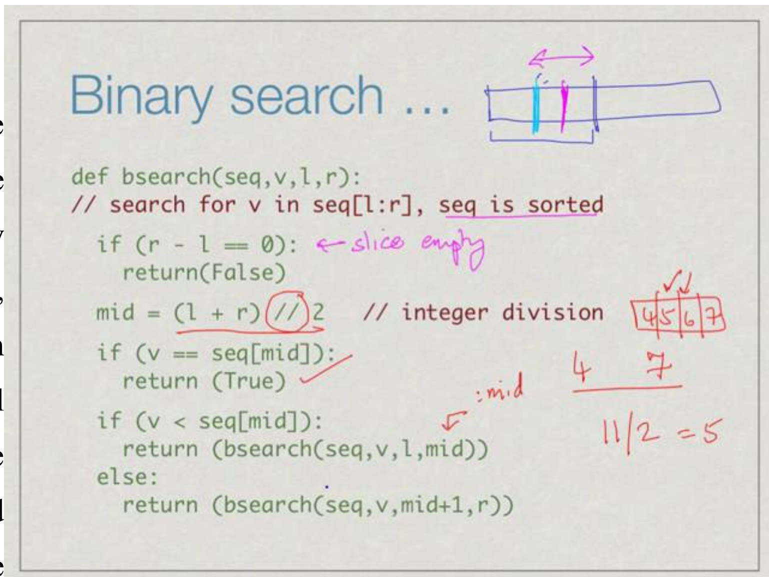

On the other hand, if we have a sorted sequence we have a procedure which would be at least informally familiar with you. When we search for a word in a dictionary for example, the dictionary is sorted by alphabetical order of words. If we are looking for a word and if we open a page at random, supposing we are looking for the word monkey and we open the dictionary at a page where the values or the word start with i, then we know that m comes after i in the dictionary order of the English alphabet. So, we need to only search in the second half of the dictionary after i, we do not have to look at any word before i.

In general if we have a sequence that efficient way to search for this value is to first look at the middle value, so we are looking for v, so we check what happens here. So, there are three cases either we have found it in which ways which case we are good, if we have not found it we compare the value we are looking for with what we see over there. If the value we are looking for is smaller than the value we see over there, it must be in this half.

On the other hand if the value we are looking for is bigger it must be in this half. So we can halve the amount of space to search and we can be sure that the half we are not going to look at positively does not have the value because we are assuming that this sequence is sorted. This is called Binary search.

This is also for example what you do when you play game like twenty questions, if you play that when somebody ask you to guess the name of a person they are thinking of then you might first ask the question whether the person is female, if the person is female then the persons and their answer is yes then you only need to think about women, if the person says no then you only need to think about m, so we have men. So, you have half number of people in your imagination we have to think about. At each point each question then further splits into two groups depending on whether the answer is - yes or no.

(Refer Slide Time: 15:22)




Here is some Python code for binary search. So, binary search in general will start with the entire list and then as we said it look at the midpoint and decide on the left, so we will have to again perform binary search on this. How would we do that? Again we will look at the midpoint of this part then we are again look at say the midpoint of the next part that we look at and so on.

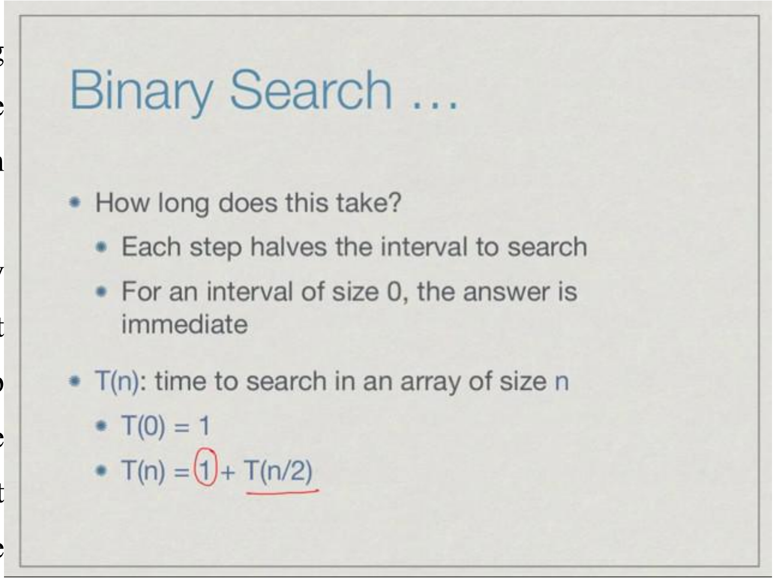
In general binary search is trying to do a binary search for a value in some segment of the list. So we will demarcate that segment using l and r. So, we are looking for this slice sequence starting with l and going to r minus 1, we are assuming that sequence is sorted and we are looking for v there. First of all if the slice is empty, so this says the slice is empty that is we have gone halving the thing and we have eventually run out of values to look at. The last thing we look at was the slice of length 1 and we divided it into 2 and we got a select of slice of length 0. Then we can say that we have not found it yet, so we are not going to ever find it and we return false.

On the other hand if the slice is not empty, then what we do is we first compute the midpoint. An easy way to compute the point is to use this integer division operation. Supposing, we are have currently the slice from 4 to 7 then at the next point we will take 11 by 2 integer wise and we will go to 5. Remember 4, 5, 6, 7. We could either choose 6 or 7 then next to split it into two parts, because we are going to examine 6 and then look at 4, 5 and 7 or look at 5 and then 4, 7. If we do integer division then we will pick the smaller output. So, we find the midpoint. Now we check whether the value is the value at that midpoint if so we return true, if it is not then we check whether the smaller, if so we continue our search from the existing left point till the left of the midpoint.

Now we are using this Python, think that this is actually means this is a slice up to mid and therefore it stops at mid minus 1. So, it will not again look at the value we just examined. it will look at everything strictly to its left. If the value that we are looking for is not the value with the midpoint and it is smaller than the midpoint, look to the left, otherwise you look strictly to the right, you start at mid plus one and go up to the current right line.

This is a recursive function. It will keep doing this at each point the interval will half, so eventually supposing we have a slice of the form just one value, so 5 to 6 for example, then at the next point right we will end up having to look at just a slice from 5 to 5 or 6 to 6 and this will give us a slice which is empty because we will find at the right point at the left point are the same.

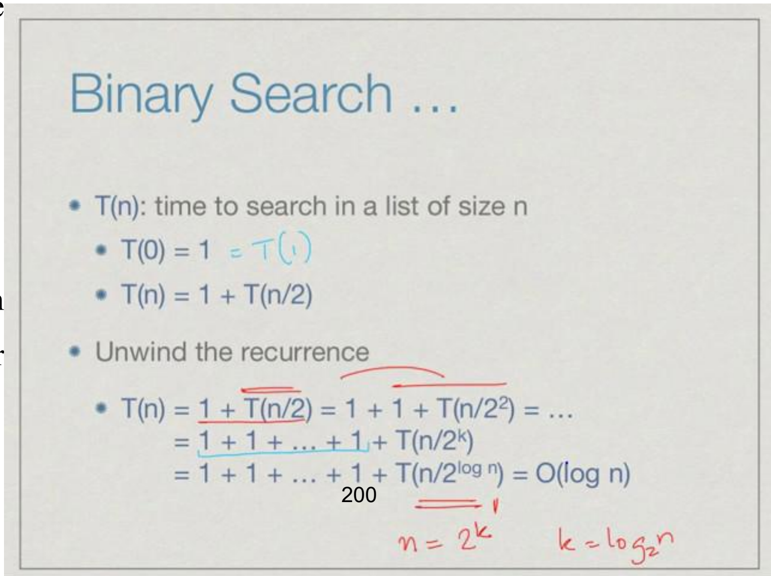
(Refer Slide Time: 18:08)




So, how long does the binary search algorithm take? The key point is that each step halves the interval that we are searching and if we have an empty interval we get an immediate answer. So, the usual way we do this is to record the time taken depending on the size of the sequence or the array or the list, so we have written array here, but it would be sequence in general. If the sequence has length 0 then it takes only one step because we just report that it is false we cannot find it if there are no elements left.

Otherwise, we have to examine the midpoint, so that takes one abstract step you know computing the midpoint and checking whether the value is we will collapse at all into one abstract step. And then depending on the answer, remember we are computing worst case the answer in the worst case is when it is going to be found in the sequence. So, the worst case it will not be the midpoint we will have to look at half the sequence. We will have to again solve a binary search for a new list which is half the length of the old list, so the time taken for n elements is 1 plus the time taken for n by 2 elements.

(Refer Slide Time: 19:18)



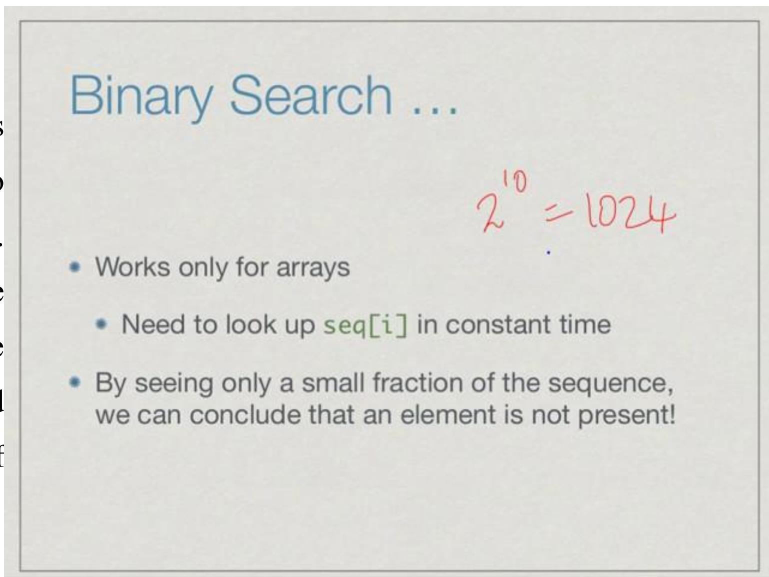

We want an expression for

T of n which satisfies, so this is what is called a recurrence normally. What function T of n would satisfy this? One way to do that is just keep substituting and see what happens. We start unwinding as itself, so, we have this by the same recurrence should be 1 plus T of n by 4, because I take this and halve it. So, T of n is 1 plus 1 plus T of n by 4. So, we start with 1 plus T of n by 2 and I expand this. Then I get 1 plus 1 plus T of n by 2 squared and in this case I will again get 1 plus 1 plus 1 by T of n by 2 cube. In general after k steps we will have 1 plus 1 plus 1 k plus 1 times or k times and t of n by 2 to the k.

Now when do we stop? We stop when we actually get T of 0 or we can also say that for T of 1 it takes one step just we want to be careful. So, when this expression becomes 1 so when n is equal to 2 to the k. So, when is n equal to 2 to the k, this is precisely the definition of log right. How many times do I have to multiply 2 by itself, in order to get n and that is the value of k that we want. After log n steps this term will turn out to be 1. We will end up with roughly log n times 1 added up and so we will get log n steps.

So what we are saying is really, if we start with the 1000 values, in the next step we will end up searching 500, next step 250, next step 125, next step 62 and so on. And if we keep doing this when will we get to a trivial sequence of length 0 or 1. Well, be keep dividing 1000 by 2 how many times can we divide 1000 by 2 that is precisely the log of 1000 to the base 2 and that is an equivalent definition of log.

(Refer Slide Time: 21:20)

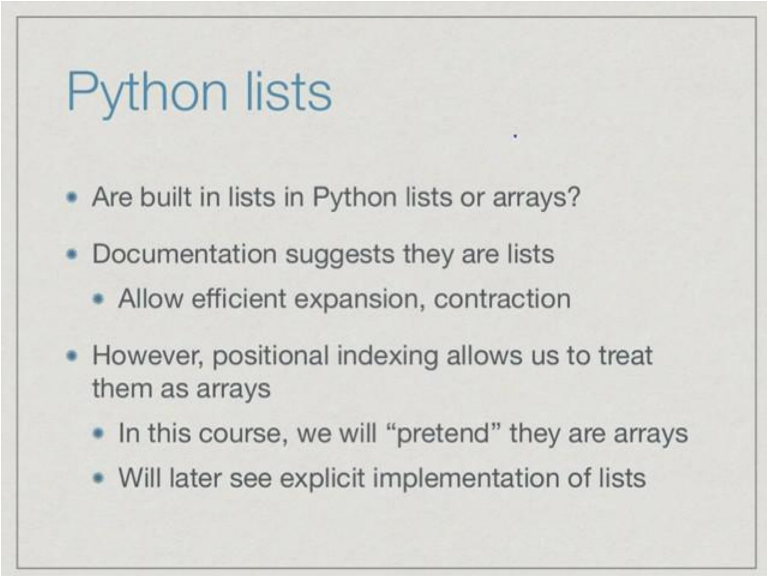



This comes back to another point. Now we have said that if we had a sorted sequence of values we can do this clever binary search, but remember that it requires as to jump in and compute mid which is fine and we need to then look at the value at the midpoint and we are assuming that this computation of mid and comparing the value of the midpoint to constant amount of time, that is why we said that it is 1 plus T n by 2 this 1 involves computing mid and looking up the frequency at the midpoint. But this can only be done for arrays because only for arrays can we take a position and jump in and get the value at that position in constant time, it will not work for lists, because we need to look up the sequence at the ith position in constant time.

Of course, one important and probably not so obvious thing if you think about binary search is that by only looking at a very small number of values, say for example we give you a sorted list of 1000 entries as I said if a value is not there we only have to search 10 possible entries, because we keep having after log n which is about to remember the 2 to the 10 is 1024 right two times, two times, two ten times is 1024. After 10 halvings of 1000 we would have come down to 0 or 1. We would definitely be able to tell quickly whether it is there or not. So, we only look at 10 values out of 1000, 999 values we do not look at all unlike the unsorted case where we have to look at every possible value before we solve.

It is very efficient binary search, but it requires us to be able to jump into the ith position in constant time therefore if I actually did a binary search on a list even if it is sorted and not on an array where I have to start at the 0th position and walk to the ith position by following links unfortunately binary search will not give me the expected bonus that I get when I use an array.

(Refer Slide Time: 23:18)




So having discussed this abstractly, we are of course working in the context of Pythons. The question is, are built in lists in python are they lists as we have talked about them or are they arrays. Actually, the documentation would suggest if you look at the Python documentation that they are lists because you do have these expansion and contraction functions so we saw we can do an append or we can do a remove of a value and so on. They do support these flexible things which are typical of lists, however Python supports this indexed position right so it allows us to look for a to the i.

If you try it out on a large list you will find that it actually does not take that much more time to go say it construct a list of a hundred thousand elements, you will find it takes no more time to go to the last position as to the first position as you would normally expect in a list we said that it should take longer to go to the last position.

Although they are lists as far as we are concerned we will treat them as arrays when we want to, and just to emphasise how lists work when we go further in this course we will actually look at how to implement some data structures. And we will see how to explicitly implement a list with these pointers which point from one element to another.

For the rest of this course whenever we look at a Python list we will kind of implicitly use it as an array, so when we discuss further sorting algorithms and all that we will do the analysis for the algorithms assuming they are arrays, we will get give Python implementation using Python's built in list, but as far as we are concerned these lists are equivalent to arrays for the purpose of this course.

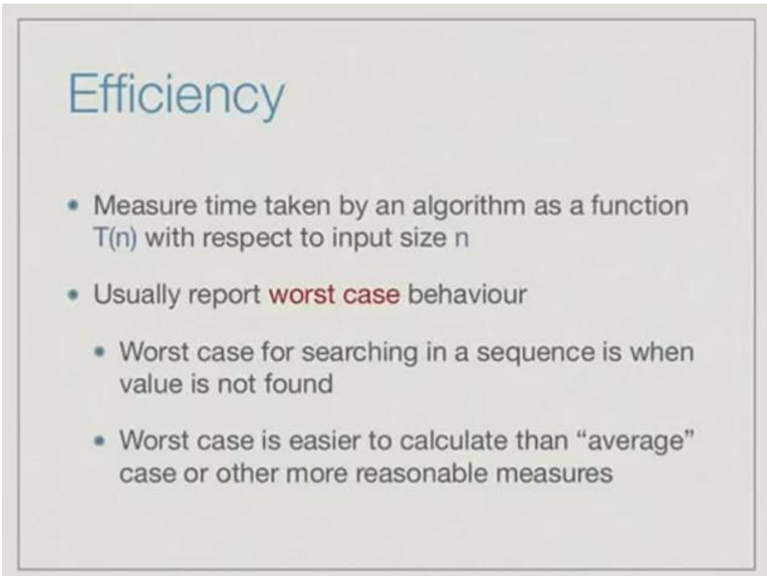
**Programming, Data Structures and Algorithms in Python**
**Prof. Madhavan Mukund**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Madras**

**Week – 03**
**Lecture – 05**

# Efficiency

When we looked at binary search, we talked about how efficient it was. So let us just spend a little bit of time informally understanding how we look at efficiency of algorithms.

(Refer Slide Time: 00:02)

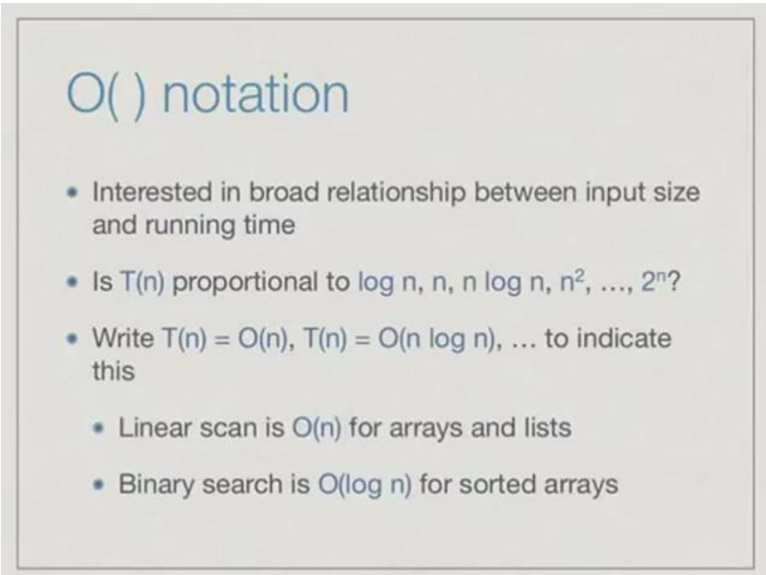



In general an algorithm will work on many different sizes of inputs, so it makes sense to talk about the efficiency as a function of the input size. The input size is n we will use a function such as T of n to talk about that time taken on an input of size n. Of course, even of the same size, different inputs will take different time for an algorithm to execute, so which of these should we taken as our measure of efficiency. The convention is to use the worst case behavior. Among all the inputs of size n which one will force our algorithm to take the longest time, and this is what we call usually the worst case efficiency.

Now in the case of searching for instance, binary search or even a linear scan, we said that the worst case would occur typically when the value that we are trying to find is not found in this sequence. So, we actually have to scan through the entire sequence or array or list before we find it in case of a linear scan. And in terms of a binary search we have to reduce the search interval to a trivial interval before we can declare that the value is not there. So that is the worst case.

Now, it may turn out that in many algorithms the worst case is rare. It may not be a representative idea about how bad or good the algorithm is and may be it could be better to give something like the average case behavior. Now unfortunately in order to determine something like an average case in a mathematically precise way is not easy, we have to have a probability distribution over all inputs and then measure different inputs and different outputs and then compute a probabilistic mean for this. So in most cases this is not possible which is why we settle for the worst case efficiency.

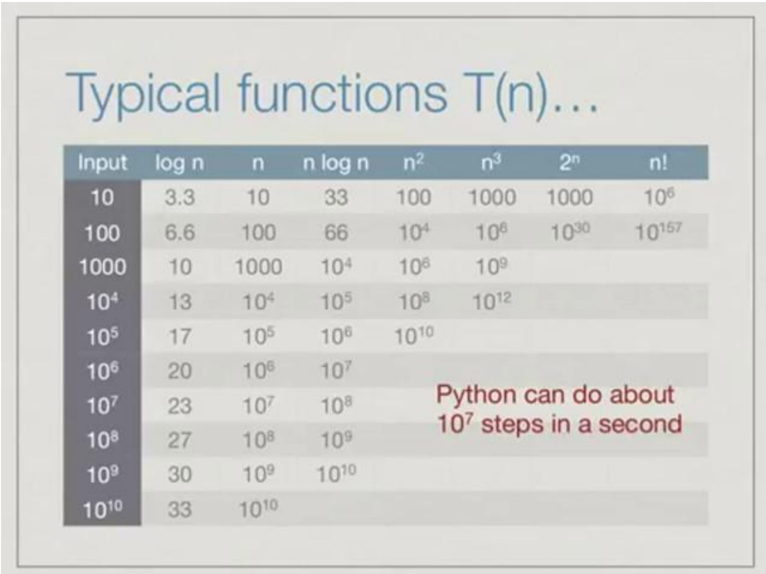
(Refer Slide Time: 01:59)




When we talk about efficiency, as we said we are broadly interested in the connection between input size and output size so we express this up to proportionality. So we are not really interested in exact constants we want to know for instance is T of n proportional to log of n, for example in the case of binary search or n in the case of linear scan or larger values like n log n, n squared, n cubed, or is it even exponentially dependent on the input, is it 2 to the n. We write this using this, what is called the big O notation. So when you say T of n is big O of n what we mean is that T of n is some constant times n. Same way T of n is big O n log n means T of n is some constant times n log n. In other words, is proportional by some constant to that value.

So, we are not going to go into much detail in this course about how big O is defined and calculate it, but it is a useful short hand to describe the efficiency of algorithms. So we will use it informally and you can go and read an algorithms text book to find out how it is more formally defined. In terms of this notation when we say that linear scan is proportional to the length of an array or a list we can say that linear scan takes time big O of n. In the same way for a sorted array binary search will take time big O log of n.

(Refer Slide Time: 03:24)



Typical functions T(n)...

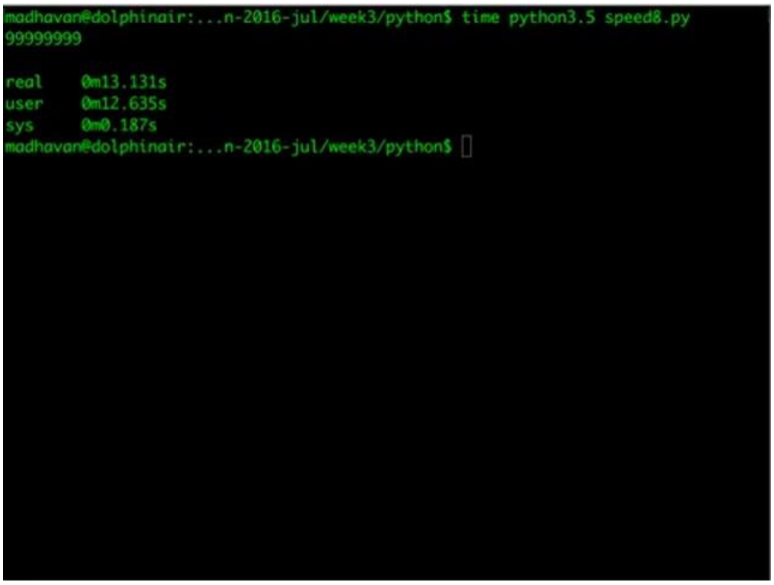
| Input | log n | n | n log n | $n^2$ | $n^3$ | $2^n$ | n! |
|---|---|---|---|---|---|---|---|
| 10 | 3.3 | 10 | 33 | 100 | 1000 | 1000 | $10^6$ |
| 100 | 6.6 | 100 | 66 | $10^4$ | $10^6$ | $10^{30}$ | $10^{157}$ |
| 1000 | 10 | 1000 | $10^4$ | $10^6$ | $10^9$ | | |
| $10^4$ | 13 | $10^4$ | $10^5$ | $10^8$ | $10^{12}$ | | |
| $10^5$ | 17 | $10^5$ | $10^6$ | $10^{10}$ | | | |
| $10^6$ | 20 | $10^6$ | $10^7$ | | | | |
| $10^7$ | 23 | $10^7$ | $10^8$ | | | | |
| $10^8$ | 27 | $10^8$ | $10^9$ | | | | |
| $10^9$ | 30 | $10^9$ | $10^{10}$ | | | | |
| $10^{10}$ | 33 | $10^{10}$ | | | | | |



So, here is a table which tabulates for different values of input n what would be the corresponding values of log n, n, n log n, n squared and so on. And what we want to probably estimate is given these values, these absolute numbers, what could be reasonable inputs that we can expect to compute within a few seconds.

Now if we type something on our computer and we do not get a response very soon these days we realize that something may be wrong. So, let us say we want to see the input in one or two seconds otherwise we will deem it to be inefficient. So, if we look at this, we have to now figure out how fast our computers are. So, by some simple hand experiments you can validate that Python can do about 10 to the 7 basic steps in a second.

(Refer Slide Time: 04:19)




So what we can do is try and execute a large loop and see how much time it takes. Here we have a bunch of programs if you already written and here is a template. So if I say look at speed4 dot py. It basically executes a loop 10 to the 4 times, hence the name 4. So, for m in range 0 to 10000 minus 1, it just assigns m to be the value i and finally there is this statement we have not seen so far, but it should be quite intuitive which says print the value of n.

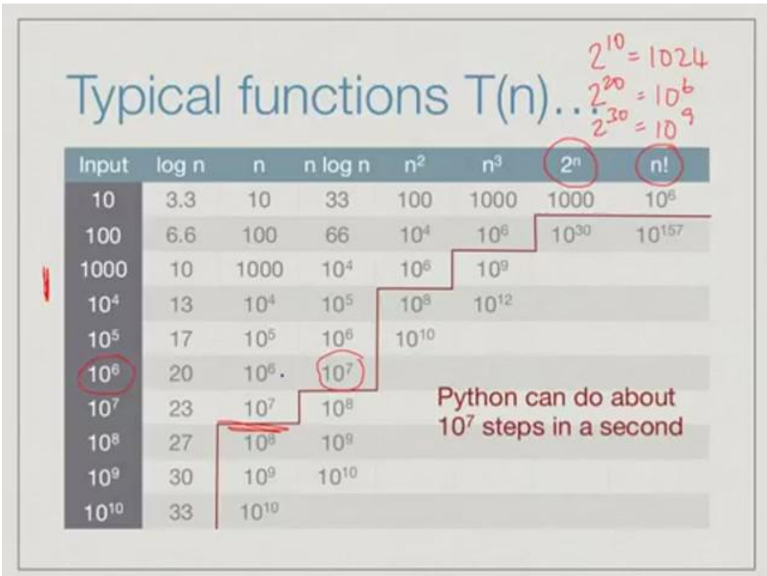
In the same way speed5 does this for 10 to the 5 times, speed6 does this 10 to the 6 times, speed7 does this 10 to the 7 times and so on. These are a bunch of scripts we have written for Python from speed4 to speed9. Now if you are working in Unix or in Linux there is a nice command called time.

First of all I can take python and I can take directly use a name of the Python program like this. So, I can say Python 3.5 and give the name of this script and it will execute it and give you the answer. But now in addition there is also a useful command called time. So, time tells us how much time this thing takes to execute and it typically reports this in three quantities; real time, user time, and system time. So, what we really need to look at is the so called user time it says that if I do this loop 10 to the 4 times it takes us fraction of a second 0.03 seconds. If i do this on the other hand 5 times, then it goes from 0.03 to 0.5. So, it is roughly a factor of 10 as you would imagine which is reasonable.

If I do this point 6 times then again it goes up not quite by a factor of 10, but it is gone up to about 0.2 seconds. Now we come to the limit that we claim 10 to the 7. So, if we run speed7 dot py, which is the loop 10 to the 7 it takes about 1 second. I mean this is not a precise calculation, but if you run it repeatedly you say at each time, because there are some other factors like how long it takes for the system to load the Python interpreter and all that, but if you just do it repeatedly you see that the 10 to the 7 takes about the second or more. This is the basis of my saying that Python can do about 10 to the 7 operations in a second.

And just to illustrate, if you actually do it for 10 to the 8 you can see it takes a very long time, and in fact it takes roughly 10 to 12 seconds to execute so soon we would hopefully see the output. As you can see 10 seconds does not seem to us like a very long time, but it is a enormously long time when you are sitting in front of a screen waiting for the response. So what we claim now is that, something that takes a couple of seconds is what we will deem as an effective input that we can solve on our computer.

(Refer Slide Time: 07:10)




| Input | log n | n | n log n | $n^2$ | $n^3$ | $2^n$ | n! |
|---|---|---|---|---|---|---|---|
| 10 | 3.3 | 10 | 33 | 100 | 1000 | 1000 | $10^6$ |
| 100 | 6.6 | 100 | 66 | $10^4$ | $10^6$ | $10^{30}$ | $10^{157}$ |
| 1000 | 10 | 1000 | $10^4$ | $10^6$ | $10^9$ | | |
| $10^4$ | 13 | $10^4$ | $10^5$ | $10^8$ | $10^{12}$ | | |
| $10^5$ | 17 | $10^5$ | $10^6$ | $10^{10}$ | | | |
| $10^6$ | 20 | $10^6$ | $10^7$ | | | | |
| $10^7$ | 23 | $10^7$ | $10^8$ | | | | |
| $10^8$ | 27 | $10^8$ | $10^9$ | | | | |
| $10^9$ | 30 | $10^9$ | $10^{10}$ | | | | |
| $10^{10}$ | 33 | $10^{10}$ | | | | | |

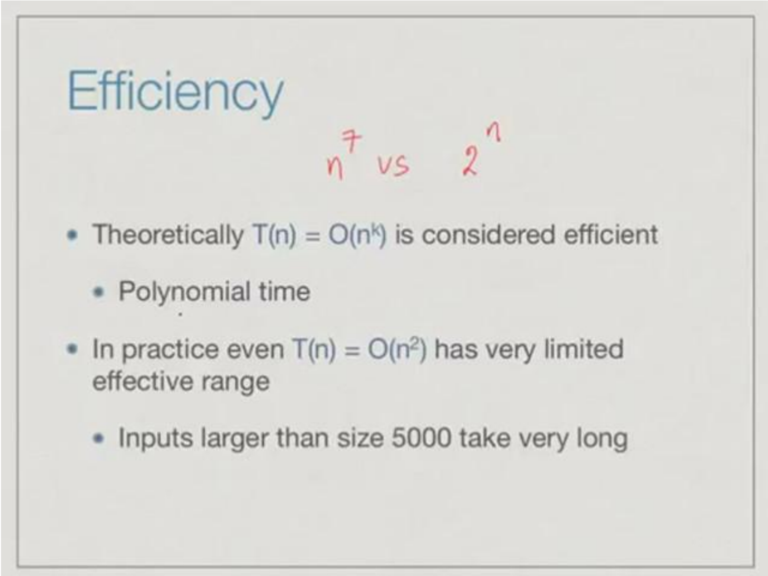
So, coming back to our table assuming that 10 to the 7 is the limit that we are looking at, let us see what happens when we mark of 10 to the 7 on these different columns. It turns out as something takes log n of time then even for 10 to the 10 it takes only 33 steps and we are fine. Of course, if input is linear then we are ignoring the constant then the input of size 10 to the 7 will take 10, so this line comes here. On the other hand if we have n log n.

Now it turns out that n log n, so it is useful to know that 2 to the 10 as we mentioned before is 1024. Therefore, 2 to the 20 will be 10 to the power 6, and 2 to the 30 will be 10 to the power 9. Here the log grows linearly as this thing grows in terms of powers of 10. So, when we have 10 to the 7 then the log is going to be something like 20 something, so it is going to be of the order of 10, its going to drop one 0. So, that is why we say that for input of size 10 to the 6, here the log is going to contribute a factor of 10 so that is going to take time 10 to the 7.

Now notice that when you do square then 10 to the 3 is already going to take 10 to the 6. So, somewhere between 1000 and 10000 say around 5000 may be if you are lucky will be the feasible limit for something which takes n squared time. And as we go to n cubed the limit drops from a few thousand to a few hundred. So, here we have between 10 to the 6 and 10 to the 9. So, somewhere between 100 and 1000 the scaling goes from 10 to the 6 to 10 to the 9, so where 10 to the 7 will be somewhere around 200 or 300. When you get to the exponentials like 2 to the n and n factorial, then unless you have an input that is really small like 10 or something like that we are going to hit problems, because we have a few tens you already get to enormous numbers like 10 to the 30.

This gives us an idea that given that our system that we are working which Python can do about 10 to the 7 steps in a second, we need to really examine this table to understand what kind of inputs will be realistic to process given the time type of algorithm that we are executing. Now Python is 10 to the 7. Python is a bit slower than other languages, but even if you are using a very fast language like C or C++ you cannot realistically expect to go beyond 10 to the 8 or 10 to the 9. So this table is more or less valid up to a scaling of a few tens in different languages. So, you can take this as a reasonable estimate across languages.

(Refer Slide Time: 09:43)




Theoretically if you look at algorithms books or complexity theoretic books, any polynomial, any dependence on n which is of the form n to the k for a constant k is considered efficient. These are the so called Polynomial time algorithms. So n cubed, n to the 5, n to the 7, all of these are considered to be theoretically efficient algorithms as compared to 2 to the n and so on. So you have n to the 7 versus 2 to the n. So, n to the 7 is considered efficient, 2 to the n is not.

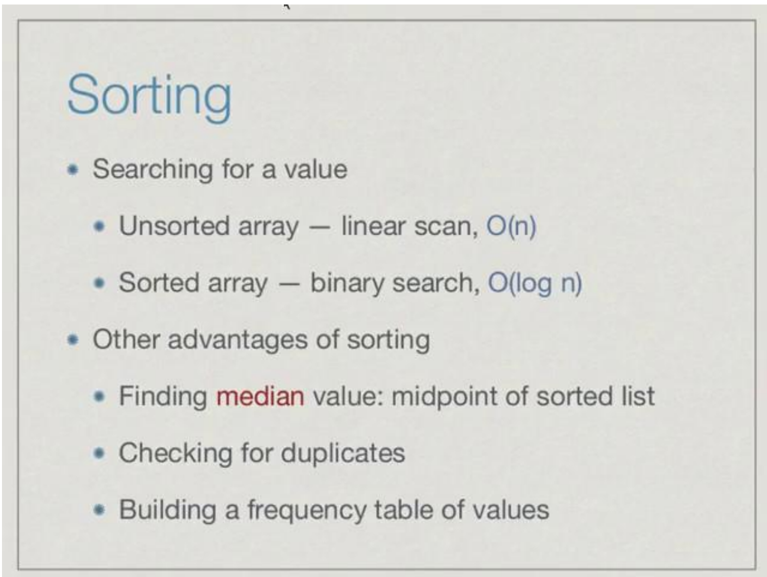
But what the table tells us if you look at the previous table, is that even n square has a very severe limit, we can only do about 4 to 5000. If you are doing something in n squared time we cannot process something larger than a few thousands. Now many of the things that we see in real life, like if we have a large spreadsheet or we have anything like that and we want to sort it then it is very likely to have a few thousand entries.

Supposing, even if you want to just look at all the employees in a medium sized company or all those children in a class and in a school or something like that, a few thousands is not at all a large number. Therefore, what we see is that if we go beyond that an n squared algorithm would take enormously long time to compute. So really we have to think very hard about what are the limits of what we can hope to do and that is why it is very important to use the best possible algorithm. Because by using something which is better you can dramatically improve the range of inputs on which your algorithm works.

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

**Week - 03**
**Lecture - 06**
**Selection Sort**

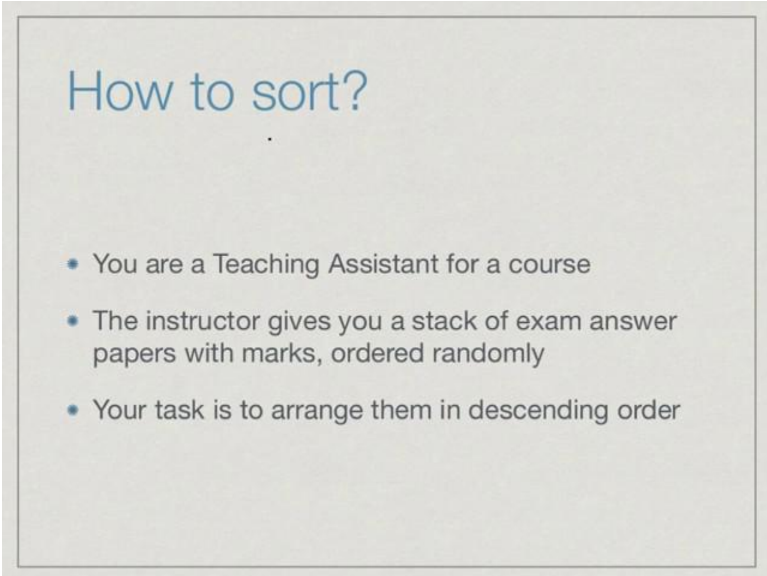
(Refer Slide Time: 00:02)




We have seen that searching becomes more efficient if we have a sorted sequence. So, for an unsorted array or a list, the linear scan is required and this takes order n time. However, if we have a sorted array we can use binary search and have the interval we half to search with each scan and therefore, take order log n time. Now sorting also gives us as a byproduct some other useful information. For instance, the median value - the median value in a set is a value such that half the value is a bigger and half are smaller.

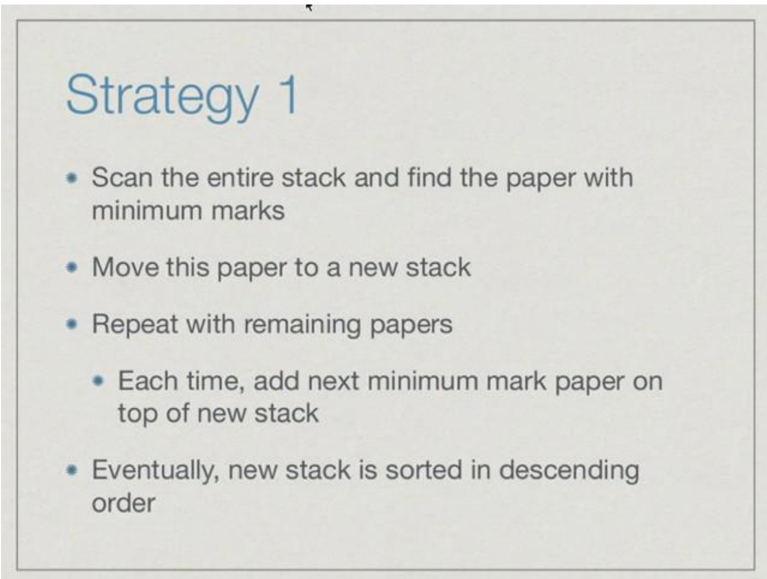
Once we have sorted a sequence, the midpoint automatically gives us the median. We can also do things like building frequency tables or checking for duplicates, essentially once we sort a sequence all identical values come together as a block. So, first of all by checking whether there is a block of size two, we can check whether there is a duplicate in our list; and for each block, if we count the size of the block, we can build a frequency table.

(Refer Slide Time: 01:06)




Let us look at some ways to sort sequences. So, forget about arrays and list for the moment, and let us think of sorting as a physical task to be performed. Suppose you are a teaching assistant for a course, and the teacher or the instructor has finished correcting the exam paper and now wants you to arrange them, so that the one with the largest marks - the highest marks is on top, the one with the second highest mark is below and so on. So, your task is to arrange the answer papers after correction in descending order of marks, the top most one should be the highest mark.

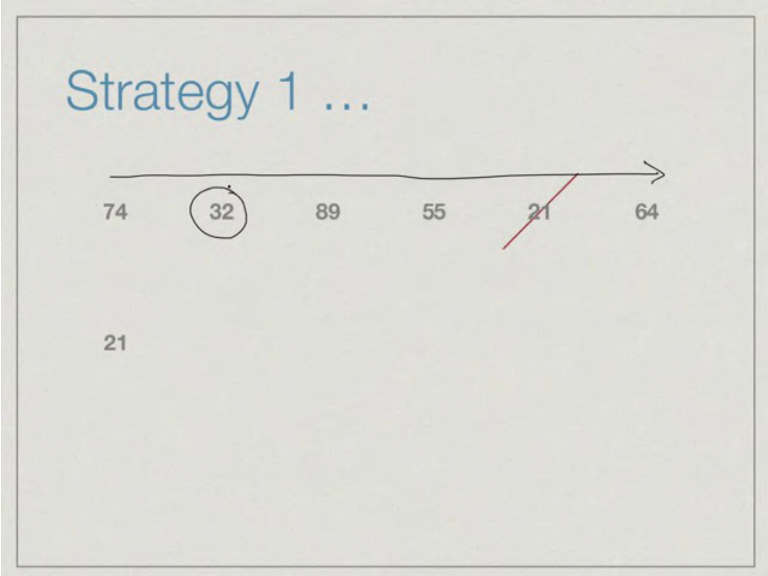
(Refer Slide Time: 01:46)




Here is one natural strategy to do this. So, what we can do is repeatedly look for the biggest or the smallest paper. Now in this case, we are going to build up the stack from the bottom, if the highest mark is on the top then the lowest mark will be at the bottom. So, what we do is we scan the entire stack, and find the paper with minimum marks. How do we do this, where we just keep looking at each paper in turn, each time we find a paper with the smaller mark then the one we have in our hand we change it and replace it by the one we have just found. At the end of the scan, in our hand we will have the paper with a minimum marks.

Initially, we assume that the top most paper has the minimum marks and we keep going down and replacing it with any lower mark we find. After this scan, we take the paper we have in our hand and put it aside and make a second stack where this is the bottom most thing. Now we have n minus 1 paper, we repeat the process. We look for the minimum mark amongst these n minus 1 papers and put this second lowest mark over all on top of the one we just put. Now, we have two papers stacked up, in order as we keep doing this we will build up the stack from bottom to top which has the lowest mark at the bottom, and the highest mark on the top.

(Refer Slide Time: 03:07)

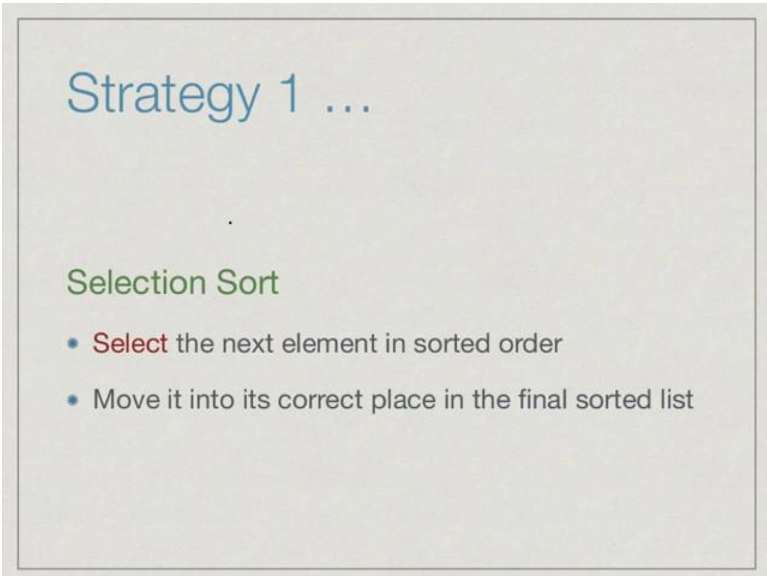



Suppose these are 6 papers. So, we have papers with mark 74, 32, 89, 55, 21 and 64. If we scan this list from left to right, then we will find that 21 is the lowest mark. So, our strategy says pick up the one with the lowest mark and move it to a new sequence or a new stack, so we do that.

(Refer Slide Time: 03:38)

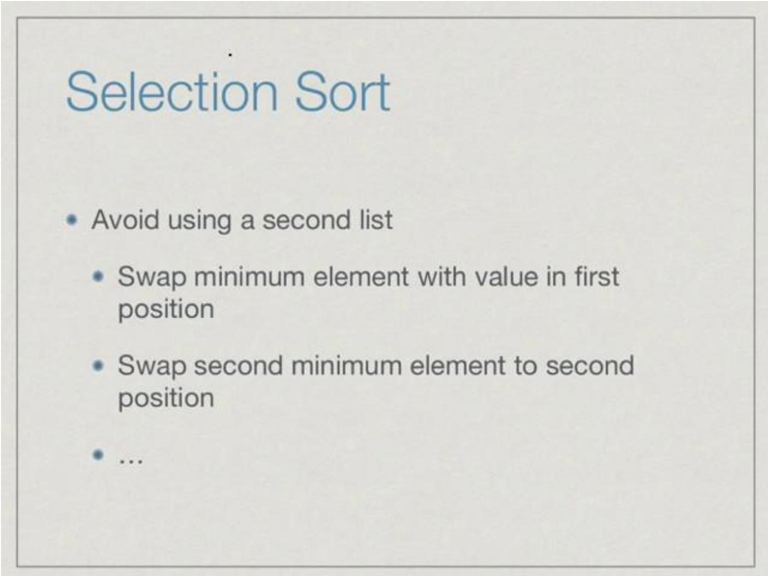
Now again, we scan from left to right this time of course 21 is gone, so we only have five numbers to scan. We will find that 32 is our next. And then proceeding in this way at the next step we will pick up 55 and then 64 and then 74, and finally 89. In this way by doing six scans on our list of six elements, we have build up a new sequence which has these six elements ordered according to their value.

(Refer Slide Time: 03:59)




This particular strategy which is very natural and intuitive has a name is called Selection Sort, because at each point we select the next element in sorted order and move it to the final sorted list which is in correct order.

(Refer Slide Time: 04:14)




In the algorithm that we executed just now, we needed to build up a second list or a second sequence to store the sorted values. So, we kept pulling out things from the first sequence, and putting it in the second sequence. However, a little bit of thought will tell us that we do not need to do this. Whenever we pull out an element from the list as being the next smallest, we can move it to the beginning where it is supposed to be and exchange it with what is at the beginning. We can swap the minimum value with the value in the first position, after this we look at the second position onwards and find the second minimum value and swap it to the second position and so on.

(Refer Slide Time: 04:53)


So, if we were to execute this modified algorithm on the same input that we had before. In our first scan, we would start from the left in the first position is 74, and the minimum is at 21. Now, instead of moving 21 to a new list, we will now swap 21 and 74.

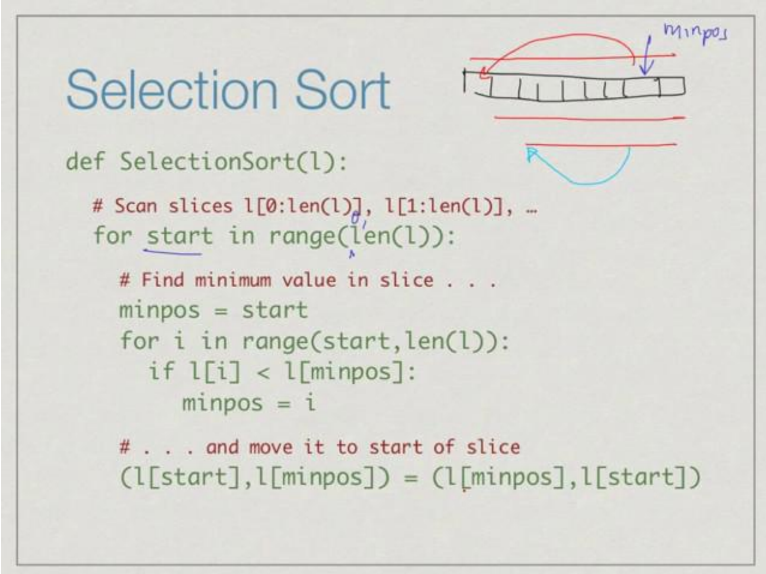
(Refer Slide Time: 05:09)


So, 21 comes in the beginning and 74 goes to the position where 21 was. Now we no longer have to worry about anything to do with 21, we only need to look at this slice if you want to call it that starting from 32. We do this and we find the second smallest element. Now, the starting element is 32 and the second smallest element also happens to be 32 that is the smallest element in this slice. So, we just keep 32 where it is. Now we start the next slice from position two. The beginning element is 89 but the smallest element is 55. So, having finished this scan we would say 55 should move to the third position and 89 should replace it.

(Refer Slide Time: 05:58)


This way we just keep going on. Now we put 64 where 89 is, and finally 74 is in the correct place and 89 is also in the correct place. And we have a sorted sequence using selection sort where instead of making a second sequence, we have just systematically moved the smallest element we have found to the start with the segment or section that we are looking at right now.

(Refer Slide Time: 06:17)

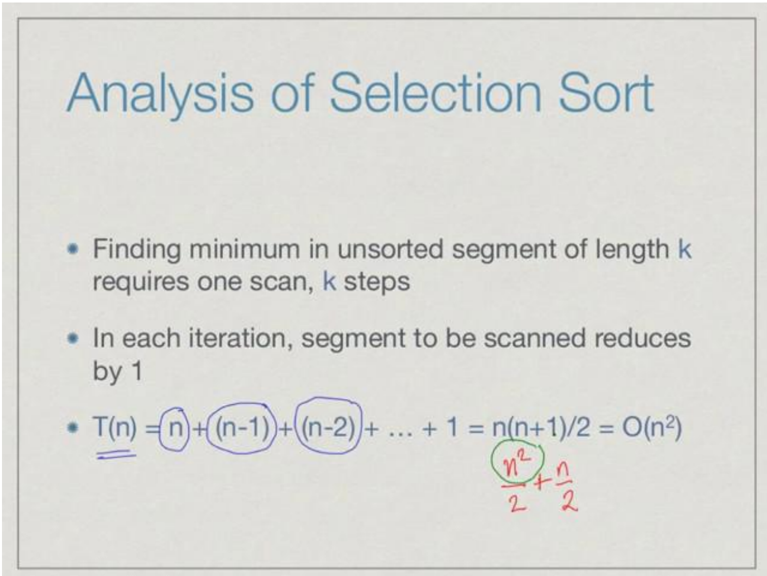



Here is the very simple Python function which implements selection sort. The main idea about selection sort is that we have this sequence which has n elements to begin with. The first time, we will scan the entire sequence, and we would move this smallest element to this point. Then we will scan the sequence from one onwards, then we will scan the sequence on two onwards, and at each point in whichever segment where we are we will move the smallest element to the beginning.

We have this starting points of each scan, so the starting point initially starts at 0, and then it goes to 1, 2 up to the length of l minus 1. So, for the starting values from 0, implicitly this is 0 remember, 0 to the length of l minus 1, we first need to find the minimum value. We assume that the minimum value is at the beginning of that position of this slice. So we said the minimum position to be the starting position; remember the starting position is varying from 0 to the length of l minus 1.

So, each slice the starting position is the first position of the slice we have currently looking at. Then we scan from this position onwards and if we find a strictly smaller value. If l of i is smaller than what we correctly believe is the minimum value, we replace the minimum position by the current index. In this way after going through this entire thing, we would have found that say this position is the position of the minimum value. Then we need to exchange these two, so we take the start position and the min position and we do this simultaneous walk, which we have seen before we take two values we exchange them using this pair notation.

(Refer Slide Time: 07:59)

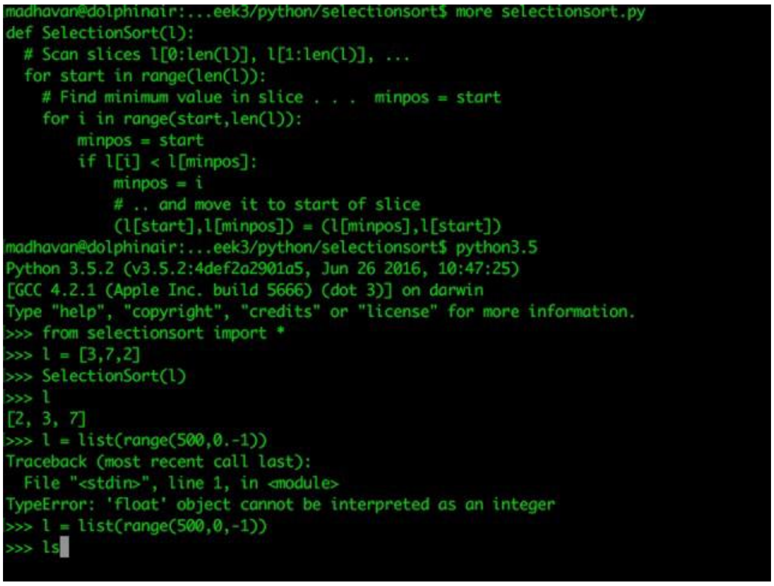



Let us see how much time this takes. In each iteration or in each round of this, we are looking at a slice of length k, and we are finding the minimum in that and then exchanging it with the beginning. Now we have an unsorted sequence of values of length k, we have to look at all them to find the minimum value, because we have no idea where it is. We cannot stop at any point and declare that there are no smaller values beyond this. So, to find the minimum in an unsorted segment of length k, it requires one scan of k steps. And now we do this starting with the segment of the entire slice that is slice of length n then a slice of length n minus 1 and so on.

And so, if we write as usual T of n to be the time it takes for an input of size n to be sorted using selection sort this will be n for the first slice, n minus 1 for the second slice on I mean position one onwards, n minus 2 for the position two onwards and so on. And if I add this all up we have this familiar sum 1 plus 2 plus 3 up to n, which you will hopefully remember or you can look up is given by this expression n into n plus 1 by 2. Now n into n plus 1 by 2, if we expand it becomes n square by 2 plus n by 2.

Now this big O notation which tells us that it is proportional to n square; when we have expressions like this which have different terms like n, n square, n cube, it turns out that we only need to record the highest term. Since, n square is the highest term n square grows faster than n, we can simplify this to O n square. If you want to see why this is so, you should look up any standard algorithms book, it will explain to you how you calculate big O, but for our purposes it is enough to remember that big O just takes the highest term in the expression that we are looking at.

(Refer Slide Time: 09:57)



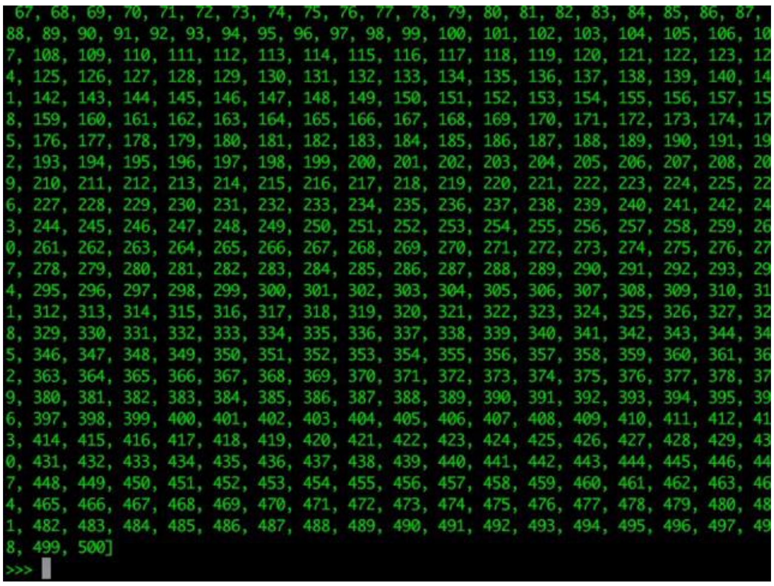

We said that for sorting algorithm like selection sort, which takes order n square will not work for the very large value say for length larger than about 5000. So, let us look at how this things works. First, this is the same code that we had in the slide, so selection sort scan slices from 0 up to the length of l minus 1. Let us start the Python interpreter. And now we will load selection sort from this file. Now notice the way selection sort works, it does not actually return a value that what selection sort does is it takes the value that the list that is passed to it and it sorts it in place.

In order to see anything from this, we have to first give it a name. So, let us take a list such as 3, 7, 2, for example, and say selection sort of l. And now we look at l, it is correctly sorted in the ascending order as 2, 3, and 7. Now in general we can take a longer list. For instance, we can use this range function and say give me the list which is created by taking the range say from 500 to 0 with step of minus 1. So, this is an descending list of length 500.

(Refer Slide Time: 11:28)

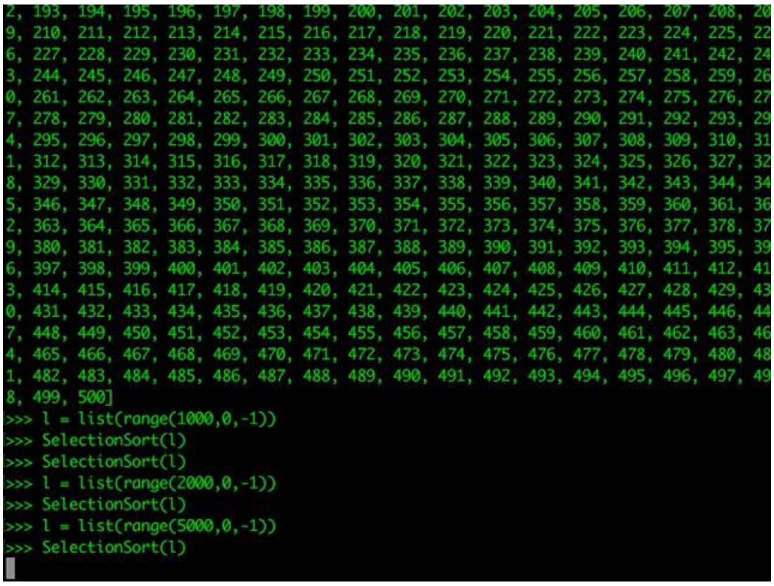

If I look at l, it is 500 down to 1.

(Refer Slide Time: 11:30)



And now if I say insertion uh selections sort of l, then it gets sorted as 1 to 500.

(Refer Slide Time: 11:44)



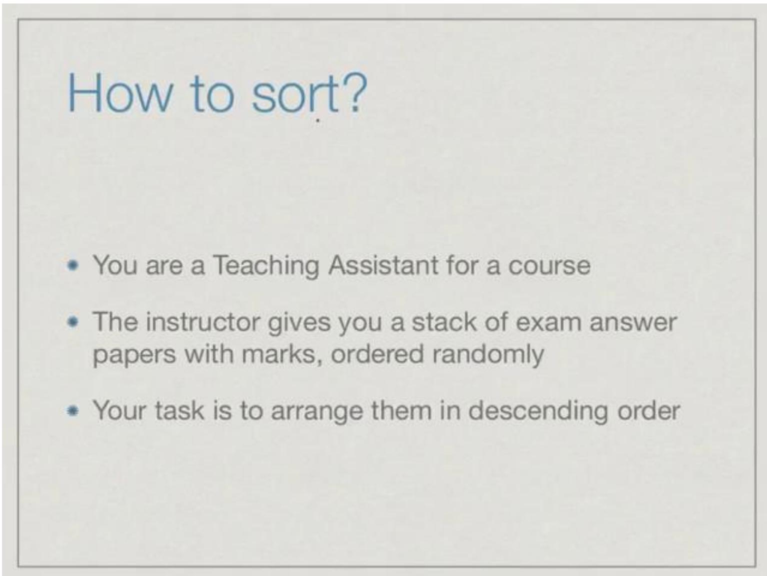

Now our claim is that this will stop working effectively around 5000. So, let us see if I make this as 1000 instead of 500, and run selection sort then you can see there is an appreciable gap. Now if I do it for say 2000, then there is slightly longer gap. If I do it for 5000 then you can see it takes a little bit of time right it takes more than one second for sure. This is just to validate our claim that in Python if you expect to do something in one second then you better make sure that the number of steps is below about 10 to the 7. And since 5000 square takes you well beyond 10 to the 7, you can expect to take a very long time.

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

**Week - 03**
**Lecture - 07**
**Insertion Sort**

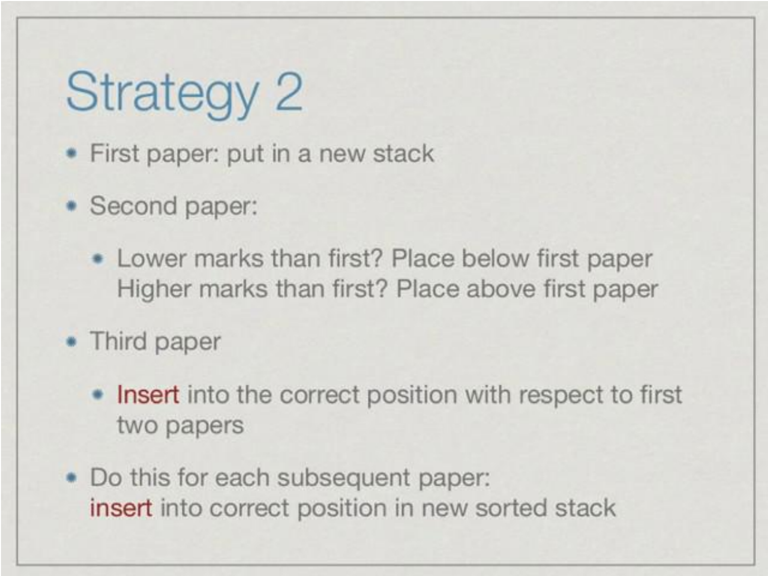
In the previous lecture we saw one natural strategy for sorting, which you would apply when we do something by hand namely selection sort.

(Refer Slide Time: 00:02)




Now let us look at another natural strategy which all of us use at some point. So, the second strategy is as follows:

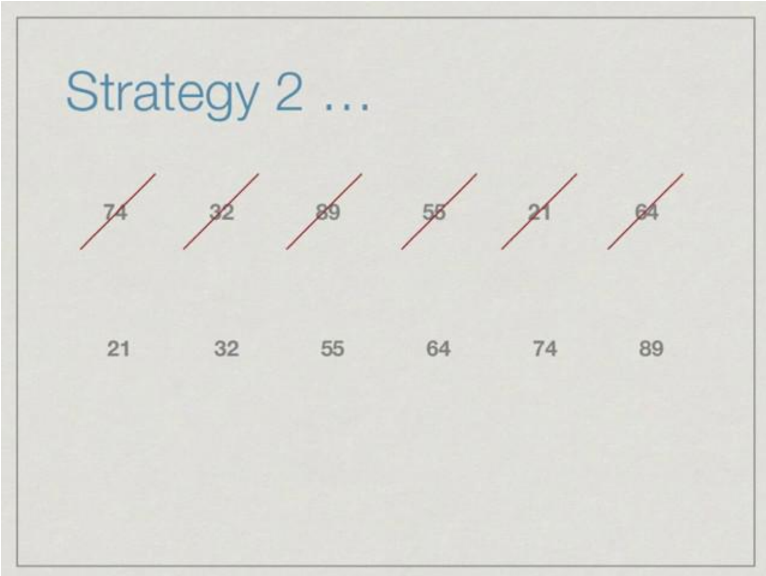
(Refer Slide Time: 00:17)




We have now a stack of papers remember with marks on them and we have to compute a new stack which has this marks arranged in descending order from top to bottom. So, we will take the first paper of the stack we have and create a new stack by definition this new stack is now sorted because it has only one paper. Now we pick the second paper from the old stack and we look at its marks as compared to the first paper that we pulled out. If it is smaller, we put it below; if it is higher, we put it above. So, in this process, we now have the new stack of two papers arranged in descending order.

What do we do with the third paper, well the third paper can be in one of three positions; it can either be bigger than the two we saw before. So it can go on top, or it could be in between the two, or it could go below. So, what we do is we scan from top to bottom and so longer if it is smaller than the paper we have seen, we push it down until we find a place where it fits. We insert the paper that we pick up next into the correct position into the already sorted stack we are building. So, keep doing this for each subsequent paper, we will take the fourth paper and insert into a correct position among the remaining three and so on.

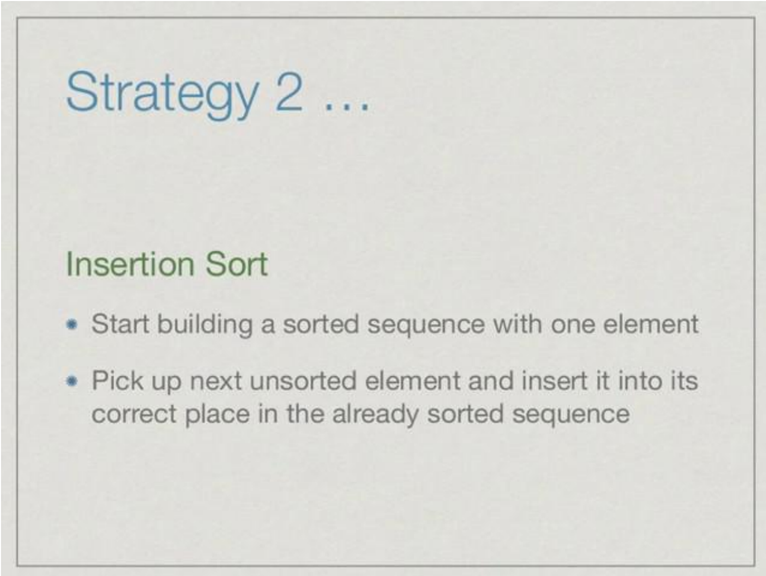
(Refer Slide Time: 01:31)




This is obviously called insertion sort. So, let us see how it would work. So, what we do with this same list that we had for selection sort is we will pick up the first value and move it to the new stack saying now I have a new stack which has exactly one value namely 74. Then when I pick up 32, since 32 smaller than 74, I push it to the left of 74. Now 89 is bigger than both, so I keep it on top of the stack at the right end; 55, I have to now look with respect to 89 and 74, so it is smaller than 89. So, it goes to the left of 89 then I look at 74 it is smaller than 74 it goes to the left of that.

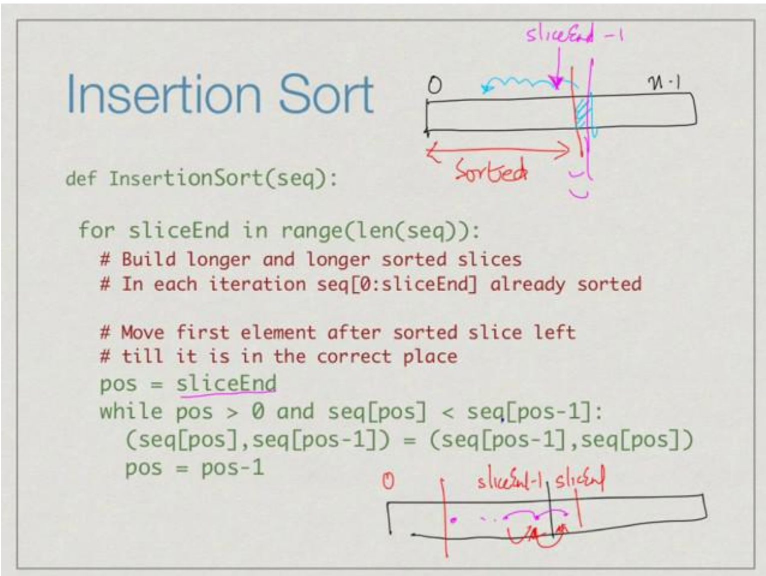
So, eventually it settles down as 32, 55, 74, 89. 21, similarly I have to start from the top and say it is smaller than 89 smaller than 74 smaller than 55 smaller than 32, so it goes all the way to the left. And finally, 64 will move down to positions past 84 and 89 and 74, but it will stop above 55. So, this is how insertion sort would build up a new list. You keep picking up the next value and inserting it into the already sorted list that you had before.

(Refer Slide Time: 02:44)




We start building a sort sorted sequence with one element pick up the next unsorted element and insert it in to a correct place into the already sorted sequence.

(Refer Slide Time: 02:56)




We can do this as we did with insertion sort without building a new sequence, and here is a function insertion sort defined in python which does this. So, what we will assume is that at any given point, we have our sequence from 0 to n minus 1 and there are some positions, so that up to this point everything is sorted.

And now what I will do is I will pick up the next element here and I will move it left until I find the correct place to put it, so that now the sorted thing extends to this length right. So, we take a sorted sequence of length i and we extend it to a sec sorted sequence like i plus 1 by inserting in the i plus one th position in the current list. So, we are going to take this position the slice end right, the slice end is going to be the last position that we have sorted already. So, this is supposed to be slice end.

So, we say sliceend it starts from the value 0 and goes up to the n minus 1th position. And at each time, we look at the value at. Actually the slice is up to sliceend minus 1 sorry. So, sliceend is a number of elements that we have sorted. We look at the value immediately after that which will be in the position called sliceend and so long as this position is bigger than 0; and if the value at that position is strictly smaller than the value at the previous position, we exchange these two right. So, what we were doing is that we are saying we draw it again. We have an already sorted slice to from 0 to slice n minus 1, and we have this position sliceend. We then assume that this is sorted. So, we compare with this value and if this is smaller then we exchange it.

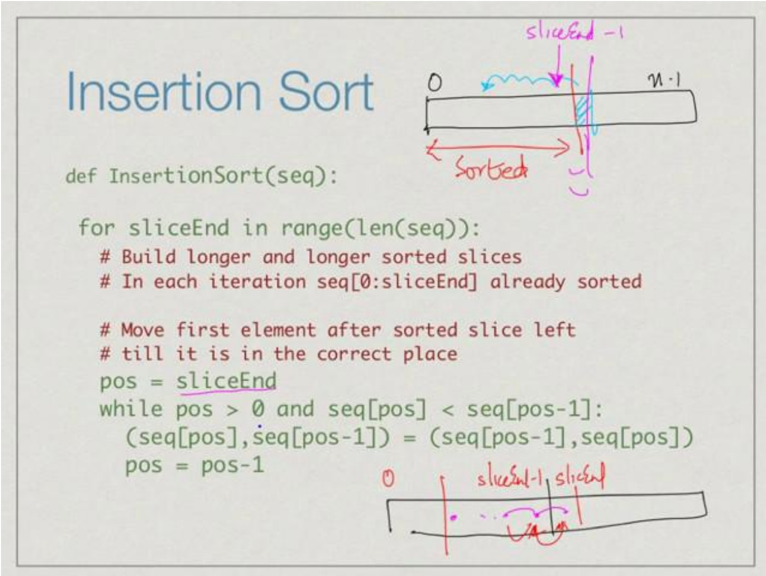
Now if you have exchanged it that means, that this value has now gone here. Now, we again a compare it to the previous value, and if it is smaller we exchange it. So, again this means that it goes one more position. We just keep going until we find that at this position the value to the left of it is equal to or bigger than this sorry equal to or smaller than this. So, we should not swap it and we have it in the correct position right, so that is what this is doing. So long as you have not reached the left hand end, you compare the value you are looking at now to the value to its left; with the value to its left is strictly bigger, this one must exchange and then you decrement the position.

(Refer Slide Time: 05:29)


Let us run this the way we have written it on this particular sequence. So, what we do is we initially assume that this thing is unsorted. So, our first thing is here. And so when we sort it, we just get a sorted list of length one which is 74. Then we look at this and we must insert it into this list 74. So since this is smaller than 74, it gets exchanged and we get now new sorted list 32, 74 and now we must insert 89 into this list right and now we see 89 is bigger than 74, so nothing happens. This list now I sorted from 32 to 89, now we try to insert 55 in this. We first compare it with this, and this will say that 55 is smaller than the value to its left, so we must exchange.

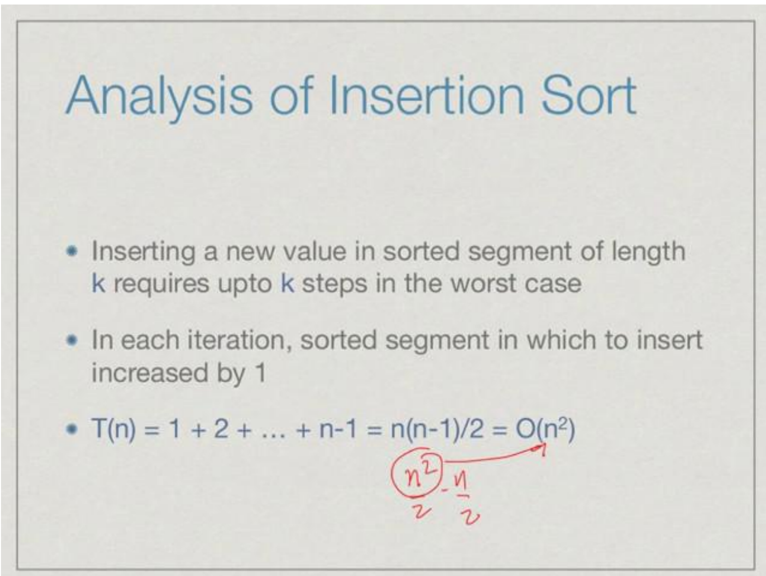
Now we will compare 55 again to the value to its left again, we will exchange. Now we will compare 55 to the value to its left and there is no change. Now we have a sorted list of length 4. Similarly, we will take 21 right, and we will compare it to 89; since 21 is smaller than 89, it will swap; since 21 is smaller than 74, it will again swap; since 21 is smaller than 55, it will swap; since 21 is smaller than 32, it will swap, but now the position sorry will swap and now the position is 0. So, we stop not because we have found something to the left which is bigger, but because we have nothing to the left.

(Refer Slide Time: 07:02)




We have two conditions if you remember that algorithm is said that either pos should be positive, the position should be greater than 0 or we should compare it to the value on its left right. In this case, we have no value to its left, so we stop.

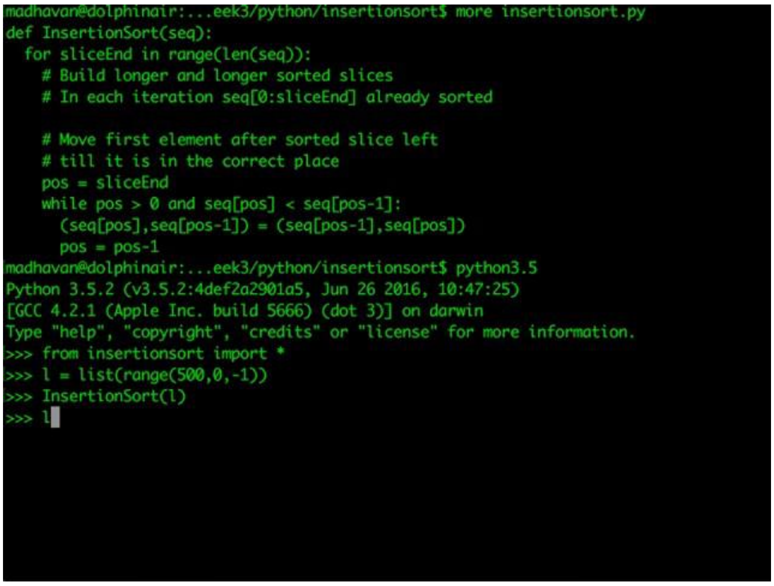
(Refer Slide Time: 07:15)




How do we analyze this? Well, at each round, what are we doing, we are inserting a new value into a sorted segment of length k. So, we start with the length 0 segment, we insert one value to it, we get a sorted length of sequence of length one, we insert a value into that we get a sorted sequence of length two and so on. Where in the worst case, when we are inserting we have to take the value all the way to the beginning of the segment.

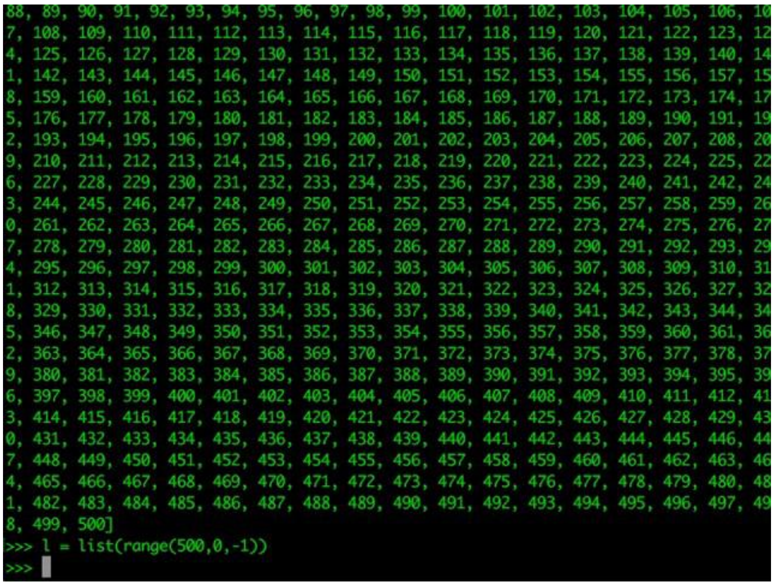
Sorting a segment of length k in the worst case takes k steps, so again we have the same recurrence not expression that we had for selection sort says that T of n is 1 plus 2 plus 3 up to n minus 1 which is n into n minus 1 by 2 which is order n square. So, again remember that this is n square by 2 minus n by 2 and so this is the biggest term and that is what we get.

(Refer Slide Time: 08:10)




Once again let us see how insertion sort actually works in the python interpreter and we will see something slightly different from selection sort when we run it. First, let us look at the code. This is the code that we saw in the slide. We just keeps scanning segments, keeps taking a value at a position and inserting it into the already sorted sequence up to that position. If we start the python interpreter, and say import this function, then as before if we for example, take a long list and sort it then l becomes sorted. So, before l was in descending order.

(Refer Slide Time: 08:57)




Now we sort it, and now l is in ascending order.

(Refer Slide Time: 09:01)

(Refer Slide Time: 09:03)


Now as before what we said is that if we try to do this for a length of around 5000 then it will be much smaller and much slower right. So, you can see it takes a long time and that is because InsertionSort, it is again order n square sort.

(Refer Slide Time: 09:24)


So, though it does it eventually it takes a long time,

(Refer Slide Time: 09:28)


But there is a small difference here. So, suppose we do it the other way, suppose we take a list which is already sorted, and now we ask it to sort, then it comes back instantly. Why should this be the case well think about what is happening now the list is already in sorted order. So, when we try to take a value at any position and move it to the left, it immediately finds that the value to its left is smaller than it, so no swapping occurs. So, each insert step takes only one iteration. It does not have to go through anything beyond the first element in order to stop the insert step. So, actually if we take even a large value like 10,000 or even 100000 this should work.

Insertion sort when you already have a sorted list will be quite fast because the insert step is instantaneous whereas this does not happen with selection sort. Because in selection sort, in each iteration we have to find the minimum value in a cell in a sequence and with no prior knowledge about what the sequence looks like it will always scan the sequence from beginning to end.

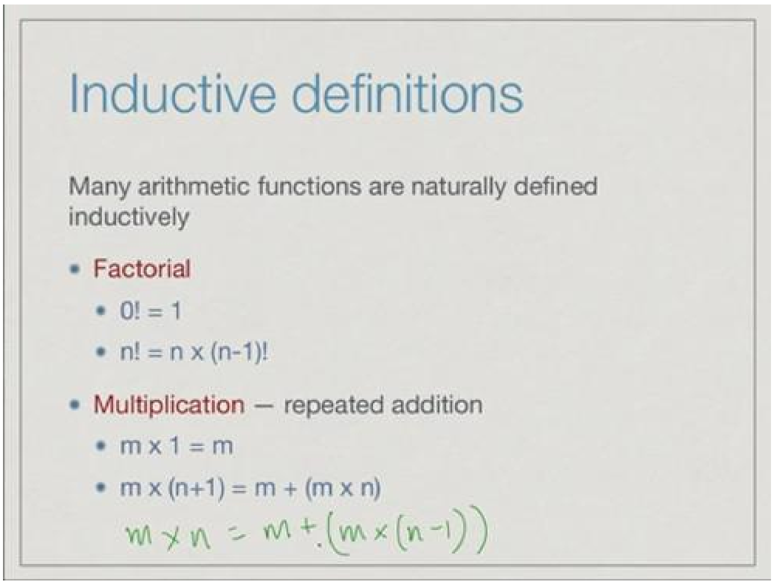
The worst case for selection sort, will happen regardless of whether the input is already sorted or not; whereas insertion sort if the list is sorted, the insert step will be very fast, and so you can if actually sort larger things. In that sense insertion sort can be behave much better than selection sort even though both of them technically in the worst case are order n squared sorts.

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

**Week - 03**
**Lecture - 18**
**Recursion**

For the last lecture of this week, we will look at recursive functions.

(Refer Slide Time: 00:05)

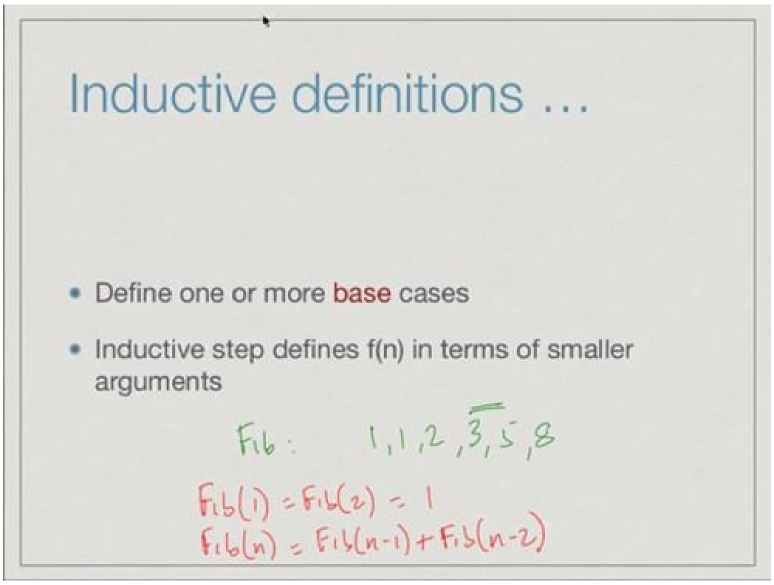



Recursive functions are typically based on what we call inductive definitions. So, in arithmetic many functions are naturally defined in an inductive way. Let us explain this by some examples. The first and most common example of a function defined inductively is the factorial function. So, we say that zero factorial is 1 and then, we say that n factorial can be obtained from n minus 1 factorial by multiplying by n.

Remember that n factorial is nothing but n into n minus 1 into n minus 2 product all the way down to 1. So, what we are observing is that after n, what appears can be rewritten as n minus 1 factorial. Inductively n minus 1 factorial can be extended to n factorial by multiplying by the value n. So, we can also do this for other functions. You may remember or you may not that multiplication is actually repeated addition when I say m times n, I mean m plus m plus m plus m, n times.

So, how do we define this inductively well we say that m times 1 is just m itself and m times n plus 1 is m plus inductively applying multiplication to n. We could equivalently write this. If you want to be symmetric with the previous case as m times n is m plus m time n minus 1, the same thing. What you are saying is that you can express m times n in terms of m times n minus 1 and then adding it. So, in both these cases what we have is that we have a base case.

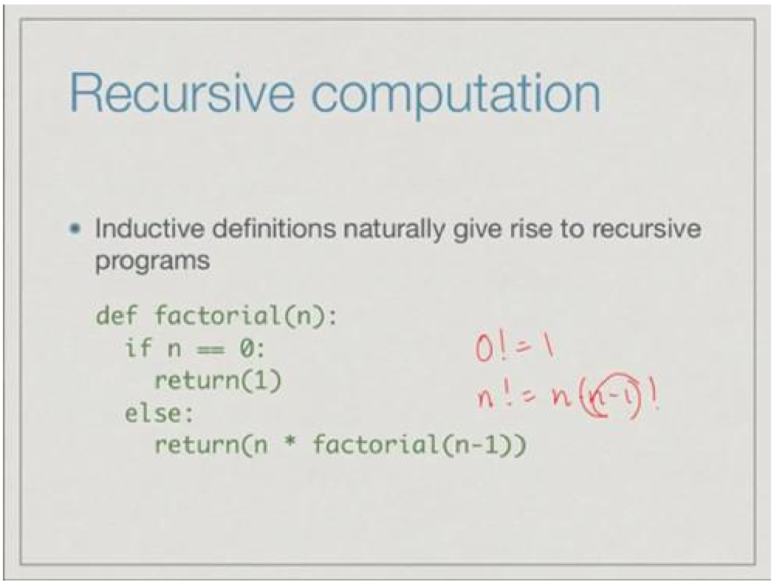
(Refer Slide Time: 01:50)




We have like 0 factorial or m times 1, where the values are given to us explicitly and then, we have an inductive step where f of n is defined in terms of f of n minus 1 and in general, it can be defined in terms of even more smaller arguments. So, one example of that is the Fibonacci series.

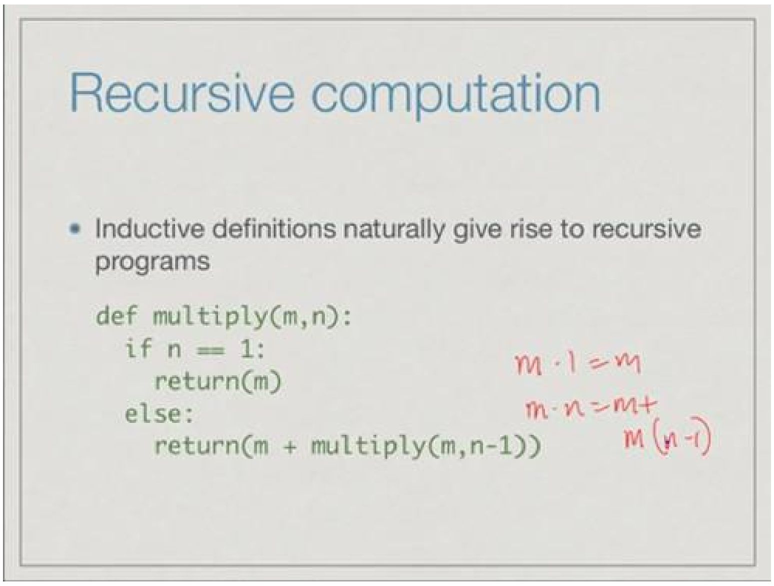
If you have seen the Fibonacci series, the Fibonacci series starts with 1 2 3 5 and so on and this is obtained by taking the previous two values and then adding. So, the general rule for Fibonacci is that the first value is equal to the second value is equal to 1 and after the second value Fibonacci of n is Fibonacci of n minus 1 plus Fibonacci of n minus 2. In general a recursive or inductive definition can express the value for n in terms of 1 or smaller values of the function for smaller inputs.

(Refer Slide Time: 02:49)




Our interest in inductive definitions is that an inductive definition has a natural representation as a recursive computation. Let us look at factorial. Here is a very simple python implementation of factorial as it is defined, it checks the value n and says that n is 0, then the return 1 otherwise return the value n times the computation recursively of factorial n minus 1.

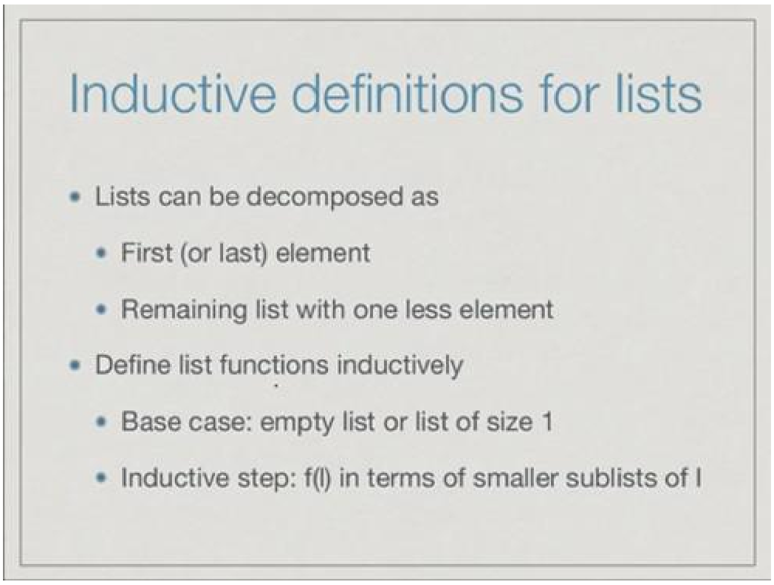
(Refer Slide Time: 03:33)




This is very clearly what we said before; it says zero factorial is 1 and otherwise if n is not 0, n factorial is n times n minus 1 factorial. So, this is exactly what we wrote before directly translated as recursive computation. We can say the same for multiplication. You can say if you want to multiply m by n, if n is 1, we return m otherwise we add m to the result of multiplying m by n minus 1.

Again we had written that before as m times 1 is n and m times n is m plus m time n minus 1. If you have an inductive definition, it immediately gives rise to a recursive function which computes same definitions. The advantage is that we do not have to spend much time arguing that this function is correct because it directly reflects the inductive definitions, the mathematical definitions of the function we are trying to compute.

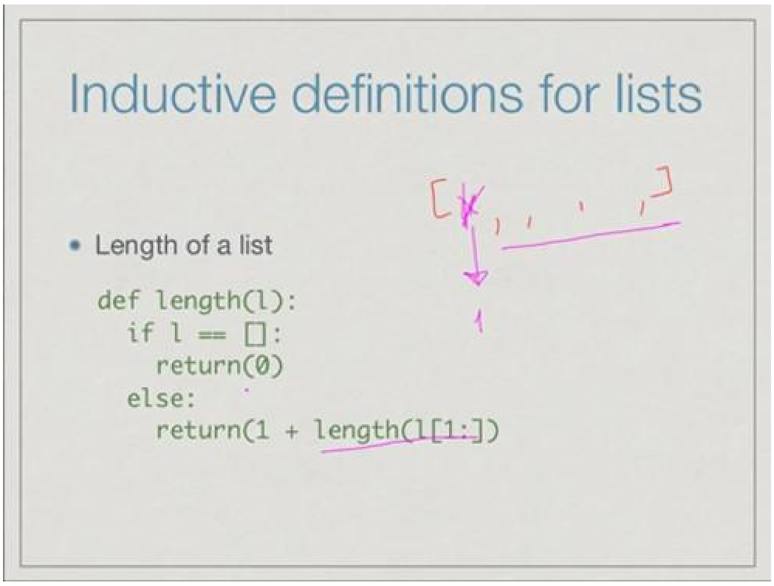
(Refer Slide Time: 04:14)




Now, what may be less obvious is that we can do the same thing for structures like list. A list has an inductive structure a list can be thought of as being built up from an empty list by adding one element at a time. So, we can think of decomposing a list, reversing the step. So, we start building a list from an empty list by adding one element at a time say we add them to the left. We add the last element and we add the second last element and so on.

But conversely we can say that given a list we can decompose it by taking say the first element and looking at that first element and the remaining list after removing the first element which has one less element. This gives us our induction we have a smaller structure on which we can try to express a function and then we can combine it with the first element to get the value for the larger thing. So, we will have a base case where the function is a defined either for the empty list or for the simple list of size 1 and in the inductive step f of l will be defined in terms of smaller sublists of l.

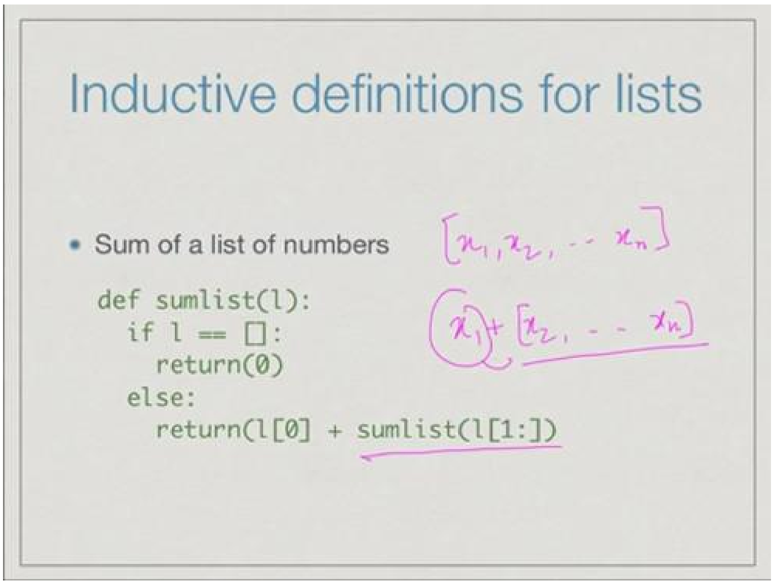
(Refer Slide Time: 05:15)




Again this is best understood through some simple definitions suppose we want to compute the length of the list l. Well the base case if the list is empty it has zero length. If l is equal to 0 - l is equal to the empty list, we return 0; otherwise what we do is we have a list consisting of a number of values. So, we pull out this first value and we say it contributes one to the length and now inductively we compute the length of the rest, right.

We return 1 plus the length of the slice starting at position one. This is an inductive definition of length which is translated into a recursive function and once again by just looking at the structure of this function, it is very obvious that it computes the length correctly because this is exactly how you define length inductively.

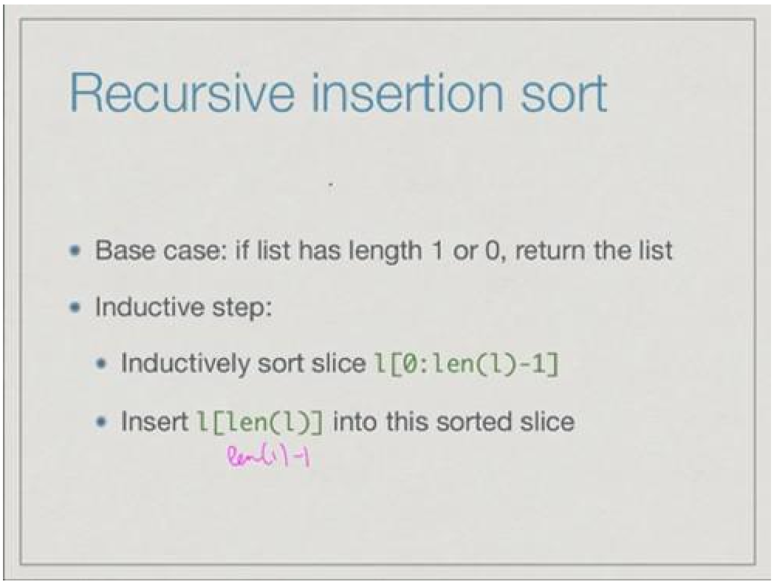
(Refer Slide Time: 06:00)




Now here is another function which does something similar except instead of computing the length, it adds up all the numbers assuming that list is a list of numbers. Again if I have no numbers to add, if I have an empty list, then the sum will be 0 because I have nothing to put into this sum.

On the other hand, if I do have some numbers to add, well the sum consists of taking the first value and adding it to the rest. If I have a list called x 1, x 2 up to x n, then I am saying that this is x 1 plus the sum of x 2 up to x n. I can get this sum by this recursive or inductive call. Then, I just add this value to that.

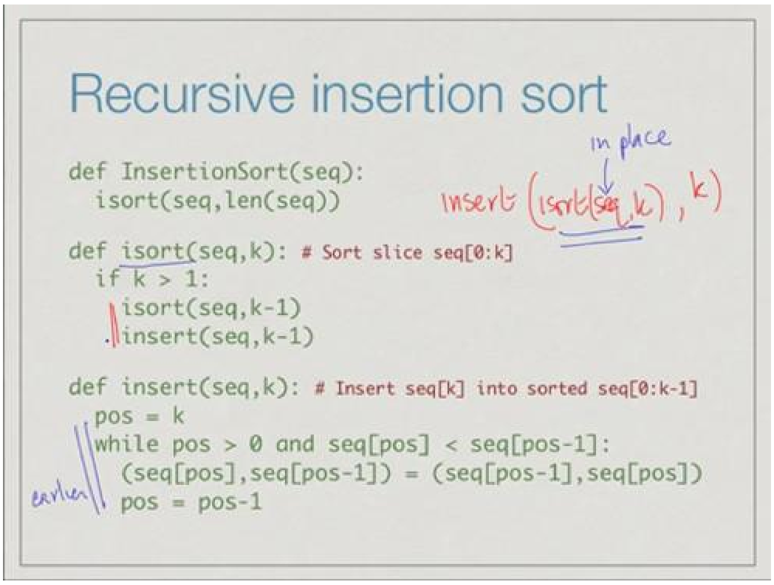
(Refer Slide Time: 06:45)




Insertion sort which we have seen actually has a very nice inductive definition. If we have a list of size 0 or size 1, it is already sorted. We do not have to do anything, so this becomes the base case.

On the other hand, if we have a list of length two or more, we inductively sort the slice from the beginning up to, but excluding the last position. This is slice from 0 to length of the list minus 1, then we take the last position and then, this should be minus 1. So, we take the value at the last position and we insert it into the sorted slice. We insert the last position into the inductively sorted slice excluding the last position.

(Refer Slide Time: 07:36)



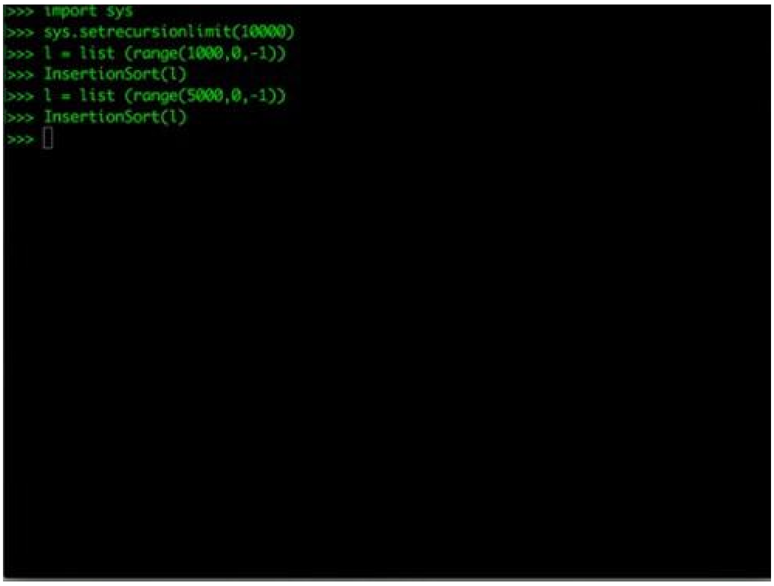

Here is a recursive definition of insertion sort in python. The natural thing in python or in any other thing would be to say that we want to insert the last position into the result of sorting the sequence up to, but excluding the last position, but the way we have written our insertion sort; this function does not return a list. It sorts this thing in place. This call would not have the correct type because insert will take a sequence and a position and insert this value at this position to its left. So, we write it now as two separate things.

First of all we have an overall insertion sort which takes a sequence and it will call this auxilliary function which says: sort this sequence up to this position. So, isort sorts the slice from 0 up to k minus 1. So, what does isort say? isort checks if is the base case if k is 0 or k is 1. If I am sorting up to the first position or I am not sorting anything at all, right - then I will just return the sequence. This is telling me sort from 0 to k minus 1.

If it is 0, then I have an empty list. If I have 0 to 1, then I have a list of one position, it is only if I have 0 to 2 that I have at least two elements. And if so, what I do is I sort k minus 1 positions and I insert the last position into this sequence. What does insert do? Insert does exactly what we did when we did the regular insertion sort. It sets a position variable or name to the last position and walks left and keeps swapping. So long as it has not reached the left hand side h and so long as it finds something to the left which is strictly bigger than the one that you looking at. So, this is what we did earlier.

What is new is this part which is this recursive call, it says sort the sequence up to this position, recursively using the same isort but change the index from k to k minus 1 and then insert the current value into this sequence.

(Refer Slide Time: 09:51)




So, as before let us run this in python. Here is the code isort rec dot py which contains this recursive implementation of insertion sort. So, if I now import this, then as before if we say l is a range of 500 values say, in descending order, then if we apply insertion sort to this, then l produces the ascending order 1 to 500.

Last time we said that for n squared sort, we should look at larger values. Supposing we take range 1000, now if we take range 1000, then something surprising happens. We get an error message from python saying that it could not sort this because it reached something called the maximum recursion depth. So, what happens when we make a recursive call is that the current function is suspended. We try to sort something of length 1000. It will try to insert the 1000th element into the result of sorting the first 999. So, we suspend the first call and try to sort 999. This in turn will try to insert the 999th element into sorting the first 998. So, it will suspend that and try to sort the first 998.

At each time we make a recursive call, the current function is suspended and a new function is started. So, this is called the depth of recursion. How many functions have we suspended while we are doing this process? Now unlike some other languages, python imposes a limit on this and the limit as we can see is clearly less than 1000 because we are not able to sort a list of 1000 using this particular mechanism. So, how do we get around this? Well, first of all, let us try and see what this limit is.

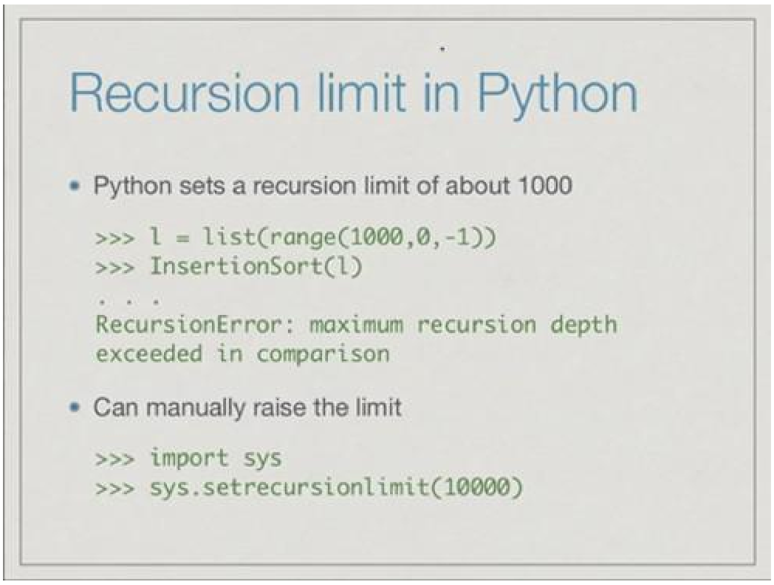
We know that we cannot sort 1000, but we could sort 500. So, can we sort 750 for example, it turns out that we can sort 750. Now, it will turn out that, for instance, we can sort 900. So, we can actually find this limit by doing what we did earlier - binary search. We can keep halving although I did not strictly halve after 750. I know it fails at 1000, it does not fail at 750, I should have tried 875, but I will spare you this binary search and I can show you that if I use 997, then it will work, but I use 998, then it will fail. So, somewhere around 998 is the recursion limit that python uses by default.

Now, fortunately there is a way to get around this. So, you can set this recursion limit and the way you do it is the following. You first have to import a bunch of functions which are called system dependent functions by saying import sys and then, in sys there is a function called set recursion limit and we can set this to some value bigger than this say 10000. Now, if we for instance ask it to sort a list of 1000, then it does not give us error. Same way, I could even say 5000 because 5000 will also only create the same kind of limit because it is well under 10000.

Remember that in this we are basically doing recursion exactly the number of times as there are elements because we keep inserting, inserting, inserting and each insertion requires us to recursively sort the things to its left. That is why we get the stack of nested recursions, but the thing to remember is that by default, python has an upper bound on the number of nested recursions which is less than 1000, if you want to change it you can by setting this recursion limit explicitly, but python does not allow you to set it to an unbounded value. You must give it an upper bound. So, you cannot say let recursion run as long as it needs to. You have to have an estimate on how much the recursion will actually take on the inputs you are giving, so that you can set this bound explicitly.

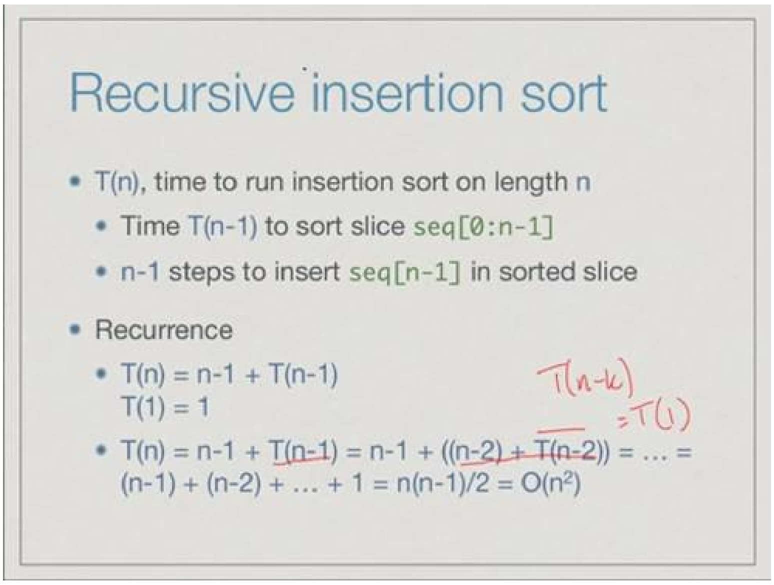
Now, the reason that python does this is because it wants to be able to terminate a recursive computation in case you have made a mistake. A very common mistake with recursive computation, it is a bit like we said for while loops, if we never make the condition false, a while loop will execute forever. Same way if we do not set the base case correctly, then a recursive computation can also go on forever. The way that python stops this and forces you to go and examine the code is by having a recursion limit saying beyond a certain depth, it will refuse to execute the code.

(Refer Slide Time: 14:51)




So, what we have seen is that we have this recursion limit and we can raise it manually by importing this sys module and setting the set recursion limit function inside sys.

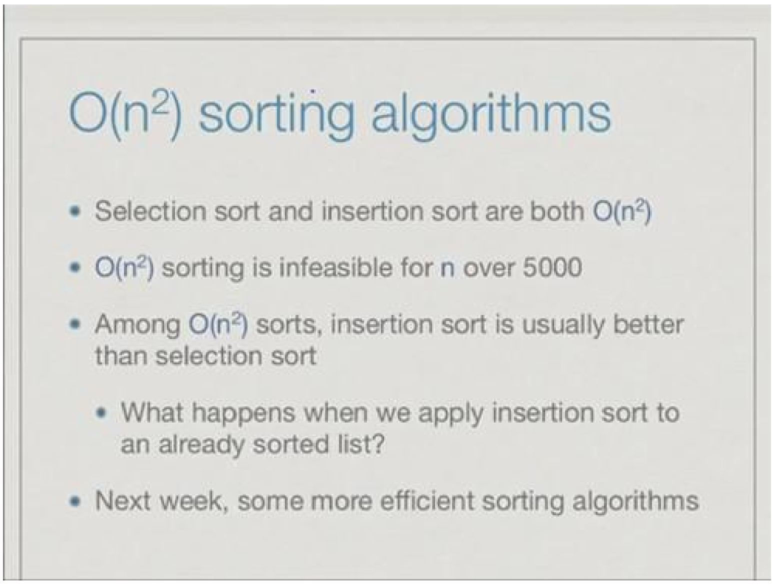
(Refer Slide Time: 15:00)




So how would we analyse the complexity of recursive insertion sort? So as before, we use T of n to denote the time it takes to run insertion sort on an input of length n. Insertion sort at the highest level consists of two steps. We first have to sort the initial slice of length n minus 1 and by definition, this will take time T of n minus 1 and then, we need n minus 1 steps in the worst case to insert the last position into the sorted slice. This gives rise to what we call a recurrence. We saw this when we were looking at how to analyse binary search which was also a recursive algorithm. We have that T n in general is n minus 1 plus T of n minus 1 and T of 1 when we come to the base case is 1.

As with the binary search, we can solve this by expanding or unwinding the recurrence. So, we have T of n is n minus 1 plus T n minus 1. If we take T n minus 1 and apply the same definitions, we get n minus 2 plus T n minus 2. Then, we take T n minus 2 and apply the same definition, we get n minus 3 and T n minus 3 and this will keep going on until we get T of n minus k is equal to T of 1. In other words when k becomes n after n steps, we will have 1. So, we will have n minus 1 plus n minus 2 down to 1 which is the same thing we had for the iterative version of insertion sort - n into n minus 1 by 2 and this is order n squared.

(Refer Slide Time: 16:41)




We have seen two order n squared sorting algorithms both of which are very natural intuitive algorithms which we do by hand when we are giving sorting tasks to perform - selection sort and insertion sort. We have also seen that both of these will not work in general if we have large values of n and not even so large; if we have values of n over 5000. But we also saw in the previous two lectures that insertion sort is actually slightly better than selection sort because selection sort, the worst case is always present because we always have to scan the entire slice in order to find the minimum value element to move into the correct position where insertion sort will stop as soon as it finds something which is in the correct order.

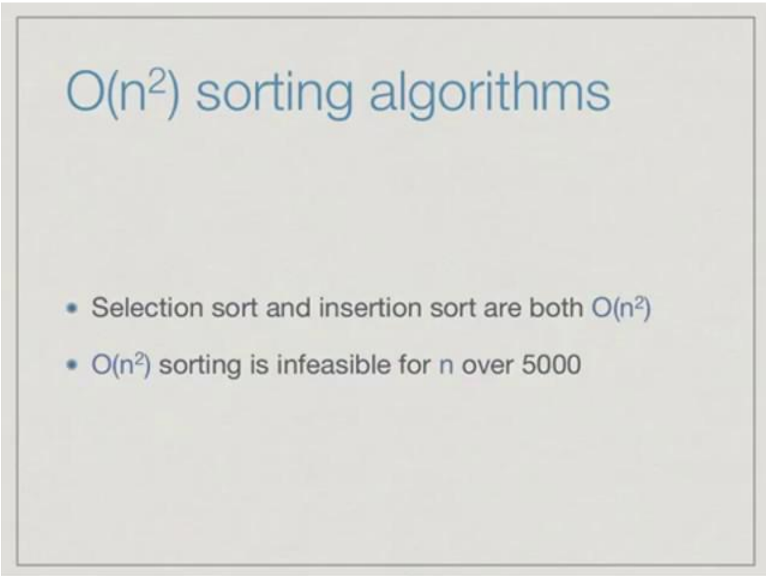
So if we have a list which is already sorted, then insertion sort will actually work much better than n squared, but we cannot rely on this, and anyway we are counting worst case complexity. So, we have to take the fact that both of these will in general not work very well for lists larger than 5000 elements.

What we will see next week is that there are substantially more efficient sorting algorithms which will allow us to sort much larger lists than we can sort using selection sort or insertion 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 - 01**
**Merge Sort**

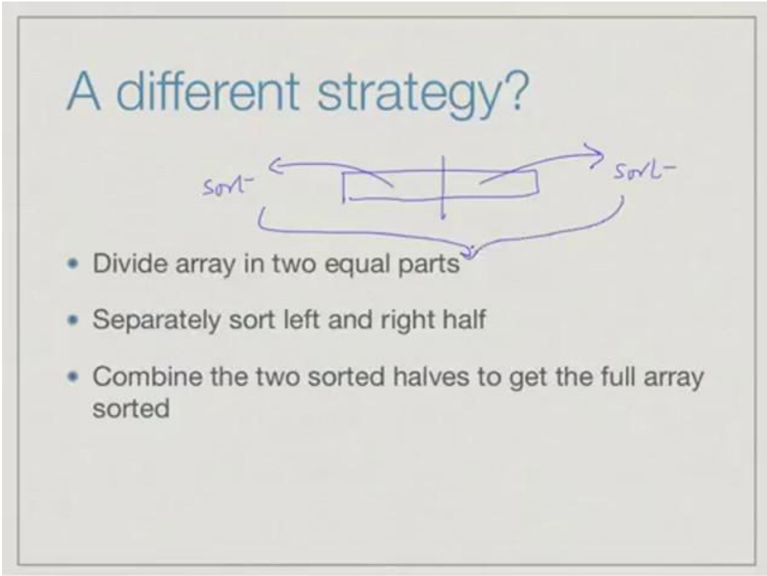
(Refer Slide Time: 00:02)




Last week, we saw two simple sorting algorithms, selection sort and insertion sort. These were attractive, because they corresponded to the manual way in which we would sort items by hand.

On the other hand, we analyzed these to see that the worst case complexity is order n squared where n is the length of the input list to be sorted. And unfortunately, n squared sorting algorithms are infeasible for n over about 5000, because it will just take too long and on the other hand, 5000 is the rather small number when we are dealing with real data.

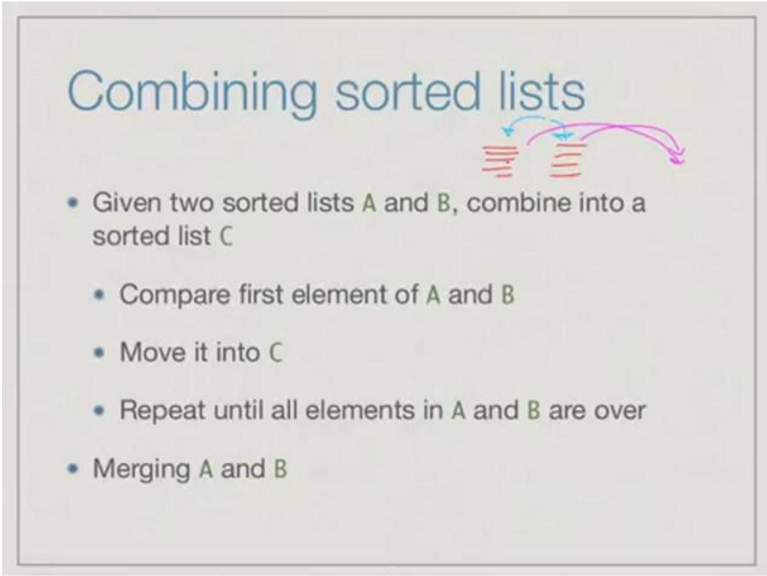
(Refer Slide Time: 00:37)




Let us examine a different strategy all together. Suppose we had the example where you were teaching assistant and you were supposed to sort the answer papers for the instructor and supposing the instructor had not one teaching assistant, but two teaching assistants. And the job is distributed to the two teaching assistants, so each one is told to go with halves the papers, sort them separately and come back and then the instructor has to put these two lists together.

In other words, you divide the array initially, the unsorted array or list into two parts and then you hand over these two parts to two different people or two different programs if you want to sort. So, you sort these two halves separately and now the key is to be able to combine these two sorted things efficiently in a single sorted list.

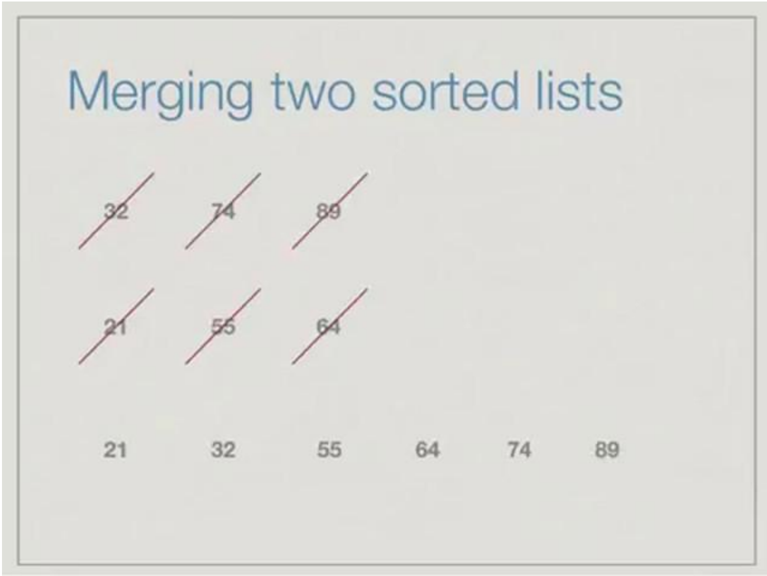
(Refer Slide Time: 01:27)




Let us focus on the last part, how we combine two sorted lists into a single sorted list. Now this is again something that you would do quite naturally. Supposing you have the two outputs from the two teaching assistants then what you would do is you would examine of course, the top paper in both. Now, this top paper on the left hand side is the highest mark on the left hand side. The top paper on the right hand side is the highest mark on the right hand side. The maximum among these two is a top overall. So, you could take the maximum say this one and move it aside.

Now you have the second highest on the right hand side and the first the highest on the left hand side. Again, you look at the bigger one and move that one here and so on. So, at each time, you look at the current head or top of each of the lists and move the bigger one to the output, right. And if you keep repeating this until all the elements are over, you will have merged them preserving the sorted order overall.

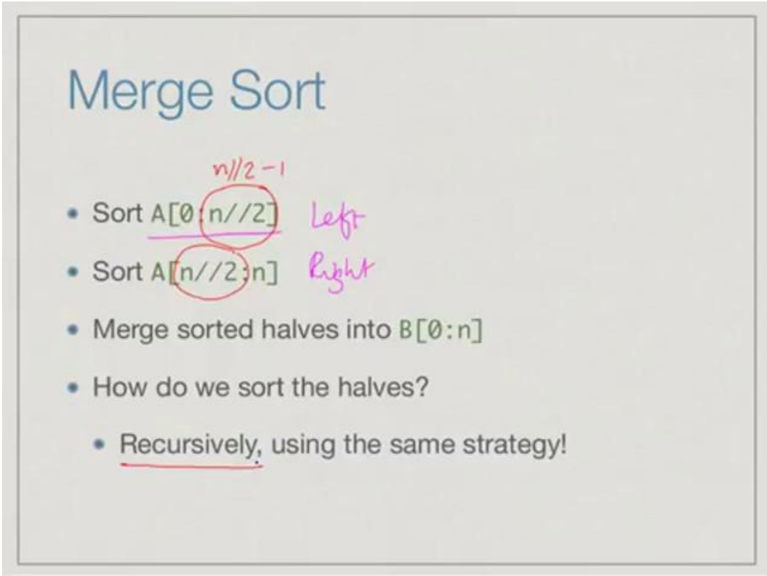
(Refer Slide Time: 02:24)




Let us examine how this will work in a simple example here. So, we have two sorted lists 32, 74, 89 and 21, 55, 64. So, we start from the left and we examine these two elements initially and pick the smaller of the two because we are sorting in ascending order. So, we pick the smaller of the two that is 21 and now the second list has reduced the two elements.

At the next step, we will examine the first element in the first list and the second element in the second list because that is what is left. Among these two 32 is smaller, so we will move 32. Now 55 is the smaller of the two at the end, now 64 is the smaller of the two at the end. Notice we have reached the situation where the second list is empty, so since the second list is empty we can just copy the first list as it is without having to compare anything because those are all the remaining elements that need to be merged.

(Refer Slide Time: 03:23)

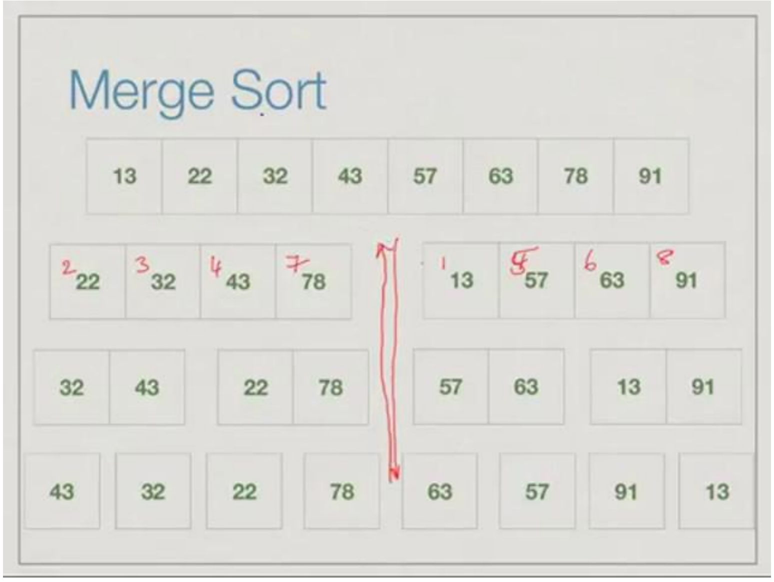



Having done this, now that we have a procedure to merge two-sorted list into a single-sorted list; we can now clarify exactly what we mean by merging the things using merge sort. So, merge sort is this algorithm which divides the list into two parts and then combines the sorted halves into a single part. So, what we do is we first sort the left hand side. So, we take the positions from 0 to n by 2 minus 1 where n is the length. This is left and this is the right.

Now one thing to note is in python notation, we use the same subscript here and here, because this takes us to the position n by 2 minus 1 and this will start at n by 2. So, we will not miss anything nor will we duplicate anything, so it is very convenient. This is another reason why python has its convention that the right hand side of a slice goes up to the slice minus 1.

If we write something like this we do not have to worry about whether we have to do plus 1, minus 1, we can just duplicate the index of the right hand side and the left hand side and it will correctly span the entire list. So, what we do is this is a naturally recursive algorithm; we recursively use this algorithm to sort the first half and the second half and then we merge these two sorted halves into the output. The important thing is we keep repeatedly doing the same thing; we keep halving, sort the half, sort the other half and merge and when do we reach a base case where when we reach a list which has only one element or zero elements there is nothing to sort. When such a situation, we can just return the list as it is and then rely on merging to go ahead.

(Refer Slide Time: 04:58)



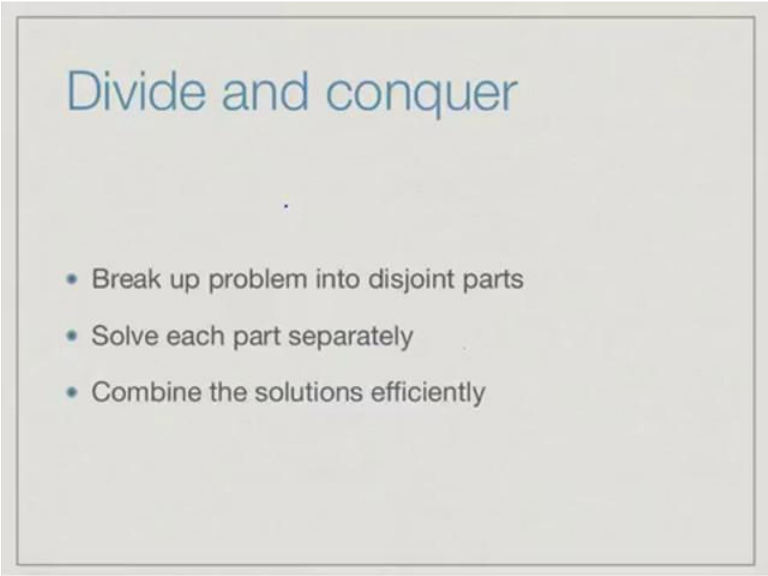

Once again let us look at an illustrative example. Supposing, we have eight items to sort which are organized like this. The first step is dividing it into two and sort each separately. So, we divide it into two groups; we have the left half and the right half. Now these are still things, which we do not know how to sort directly, so again we divide into two. The left half gets divided into two further subdivisions and so does the right.

Now we have list of length two, we could sort them by hand, but we say that we do not know how to sort anything except a list of length 1 or 0, so we further break it up. Now, we have trivial lists 43 and 32 on the left, 22 78 and so on, so we end up with eight lists of length one which are automatically sorted.

At this stage, the recursion comes back and says, you have sorted the sub list 43 for example, this list we have sorted the left into 43 and the right into 32 in a trivial way, so we need to combine them by merging. So, we merge 43 and 32 by applying a merge procedure and we get 32 before 43 when we merge 22 and 78 they remain in the same order. Here 57 come before 63 and finally, 13 comes before 91.

Now, at this level, we have two lists of lengths two which are sorted and so they must be merged and similarly here we have two lists of lengths two which are sorted and they must be merged. So, we merge the first pair, we get 22 followed by 32, followed by 43, followed by 78. And similarly, here we get 13 followed by 57 followed by 63 followed by 91. So, after doing these two merges we have now two lists of length four each of which are sorted. And now we will end up picking from this 13 and then so this is 13 and then we will pick 22 then we pick 32 and then we pick 43 then 57, then 63, then 78 and then 91 right. This is how this recursion goes, you first keep breaking it up, down till the base case and then you keep combining backwards using the merge.

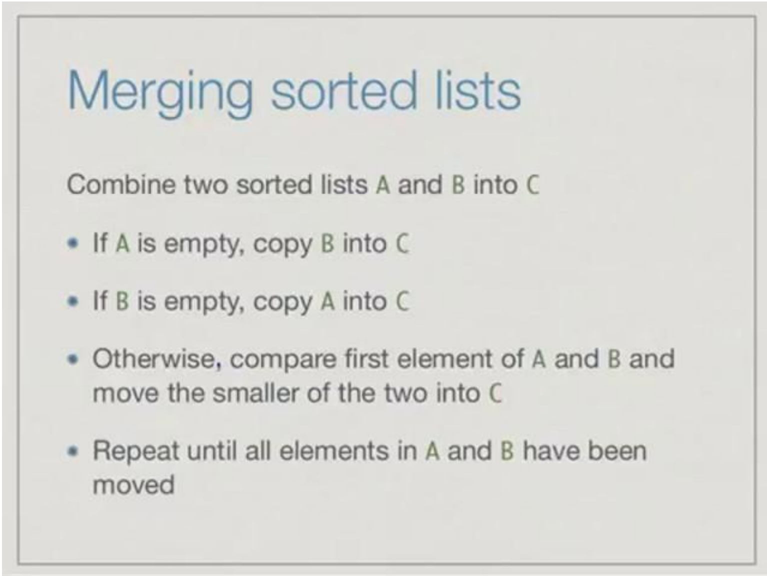
(Refer Slide Time: 07:08)




This strategy that we just outlined from merge sort is a general paradigm called divide and conquer. So, if you have a problem where you can break the problem up into sub problems, which do not have any interference with each other. So, here for instance, sorting the first half of the list and sorting the second half of the list can be done independently. You can take the papers assigned by the instructor, give them to two separate teaching assistants, ask them to go to two separate rooms; they do not need to communicate with each other to finish sorting their halves.

In such a situation, you break up the problem into independent sub problems and then you have efficient way to combine the solved sub problems. So that is the key there, how efficiently you can combine the problems. If you takes you a very long time to combine the problems then it is not going to help you at all. But, if you can do it in a simple way like this merge sort where we do the merging by just scanning the two lists from beginning to end and assigning each one of them to the final thing as we see it, then you can actually derive a lot of benefit from divide and conquer.

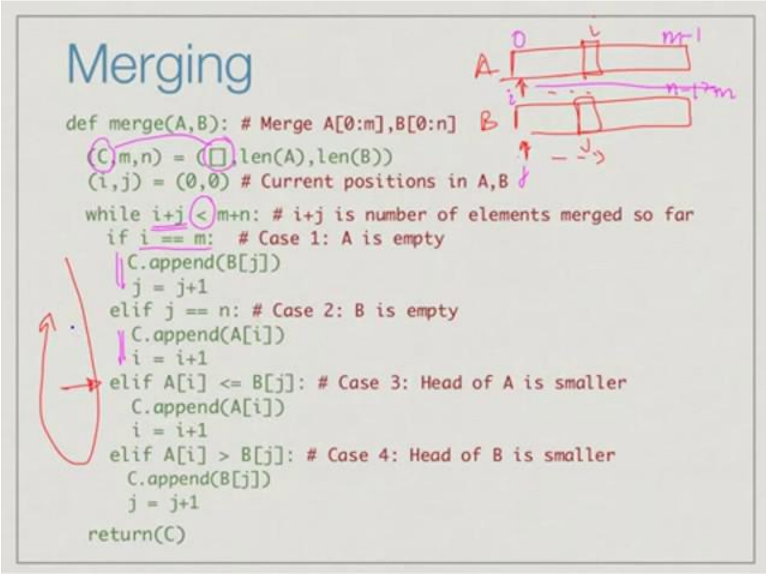
(Refer Slide Time: 08:08)




Let us look a little more in detail at the actual algorithmic aspect of how we are going to do this. First, since we looked at merging as the basic procedure, how do we merge two sorted list. As a base case, if either of them is empty as we saw in the example, we do not need to do anything; we just copy the other one. So, we are taking two input lists A and B, which are both sorted and we are trying to return a sorted list C.

So, if A is empty, we just copy B into C; if B is empty, we just copy A into C. Otherwise, what we do is if both are not empty, so we want to take the smaller one of the head of A and the head of B and move that to C, because that will be the smallest one overall in what is remaining. So, we compare the first element of A and B and we move the smaller one into C and we keep repeating this until all the elements in A and B has been moved.

(Refer Slide Time: 09:05)




This is a python implementation of this merge function in general the two lists need not be the same length. So, we are merging A of length m and B of length n into an output list of length C. Initially, we set up m and n, because we need to keep track of how many elements we have moved in order to decide when to terminate. So, we set up m and n to point to the lengths of A and B respectively and we initialize the output list to be the empty list, because remember that in python the type of C will only be known after it is assigned a value. So, we need to tell it that initially the output merge list is an empty list, so that we can then use append to keep adding items to it.

Now what we are going to do is essentially start from the left hand side of both A and B, so we are going to start here and walk to the right; as we go long we are going to move one of the two elements. So, we need an index to point here, so we use the index i and j to point into A and B respectively.

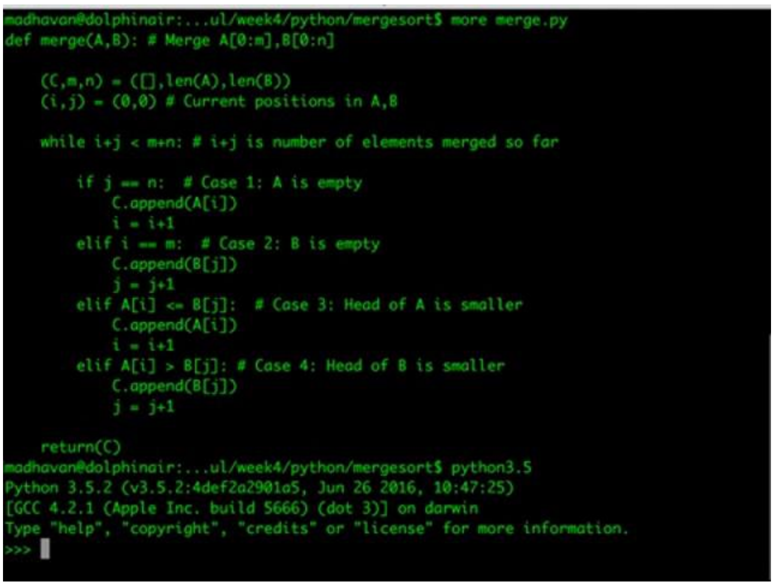
And initially, these indices point to the starting element which is 0. So, as we move along if we move i from 0 to 1 that means, we have processed one element in A; if we move it to 3; we have processed three elements in A and so on. So, at any given time i plus j will tell us how many elements have been moved so far to the output; eventually, everything in A and everything in B must be moved to the output. This will go on so long as i plus j is not reached the total m plus n which was the total number of elements we had to move to begin with.

While i plus j is less than m plus n, we have to look at different cases. The first two cases are where one of the two lists is empty; either we have reached the end of A, so i has actually reached. So, remember the indices go from 0 to m minus 1 and 1 to n minus 1. So, if i has actually gone to m; that means, that we have exhausted the elements in A. So, we append the next element in B and we keep going by incrementing the pointer in B or the index in B.

Similarly, if we have reached the end of B, we append the next element A and we go back. Now remember that at this point because i plus j is less than m plus n, if we have finished m elements, but m plus n is not been reached, there must be some element in B. Similarly, if we have finished I mean we have finished n elements in B, but i plus j is not yet m plus n, there must be at least one element left in it. These two things will definitely work just by checking the fact that we have not finished all the elements, but one of the lists is exhausted.

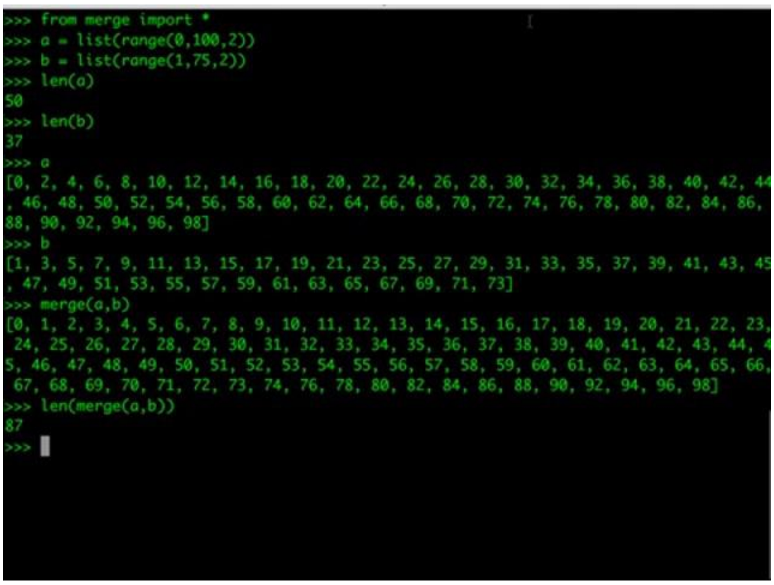
Now, if neither list is exhausted then we have to do a comparison. So, we come to this case and we check whether the element in A is smaller than or equal to the element in B. So, at this point we are in general looking at some A i and some B j. So, we have to decide which of these two goes into C next. The smaller of the two if it comes in A, we append that to C at the increment i pointer; otherwise, we append the B value increment the j pointer. At the end of this loop, what we would have done is to have transferred m plus n elements in the correct order from A and B into C.

(Refer Slide Time: 12:29)




Let us just verify that this code that we have described works in python as we expect. So, here is a file merge dot py in which we had exactly the same code as we had on the slide. So, you can check that the code is exactly the same it goes through this loop while i plus j is less than m plus n and it checks the four cases an according to that copy is either in element from A to C, or B to C and finally returns the list C.

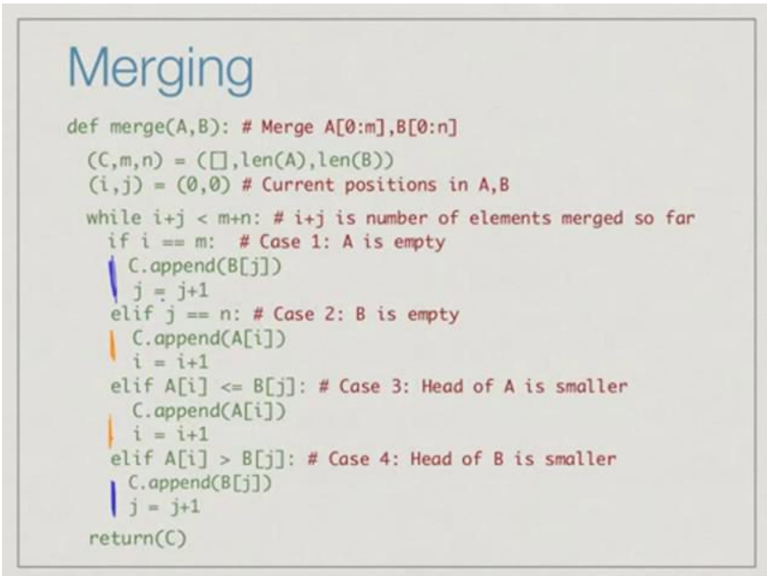
(Refer Slide Time: 13:00)




The simplest way to do this is to try and construct two lists; suppose, we take a list of numbers where the even numbers from 0 to 100. So, we start at 0 and go to 100 in steps of 2. And we might take the odd numbers from say 1 to 75, so we do not have the same length here right. The length of a is 50, the length of b is 37. And a has it's in ascending order 0 to 98 in steps of 2; B is 1 to 73 in steps of 1 and in steps of 2 again.

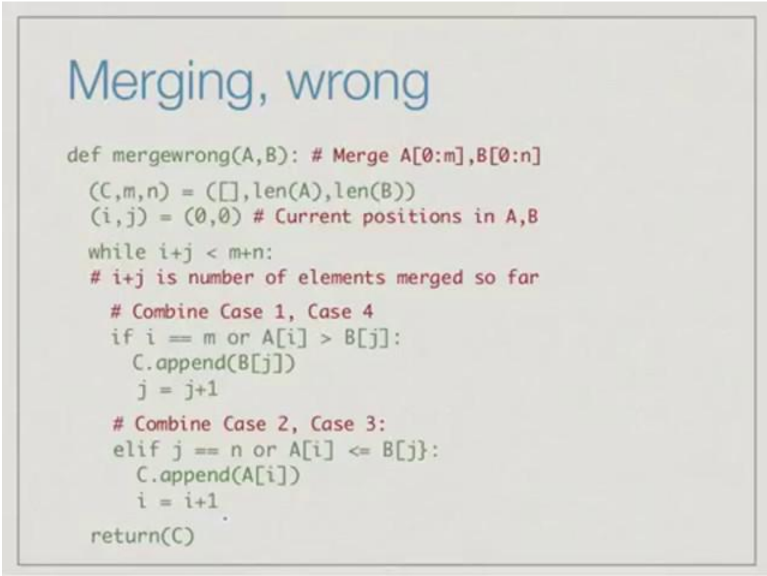
Now if we say merge a, b, and then we get something which actually returns this merge list. Notice that up to 73, which is the last element in b, we get all the numbers. And then from here, we get only the even numbers because we are only copying from a. And if you want to check the length of the merge list then it is correctly 37 plus 50 is 87.

(Refer Slide Time: 14:12)




If we go back and look at this code again, then it appears as though we have duplicated the code in a couple of places. So, we have two situations case 1 and case 4 where we are appending the element from B into C and we are incrementing j. And similarly, we have two different situations - case 2 and case 3, where we are appending the element from A into C and then incrementing i. So, it is tempting to argue that we would have a more compact version of this algorithm, if we combine these cases.

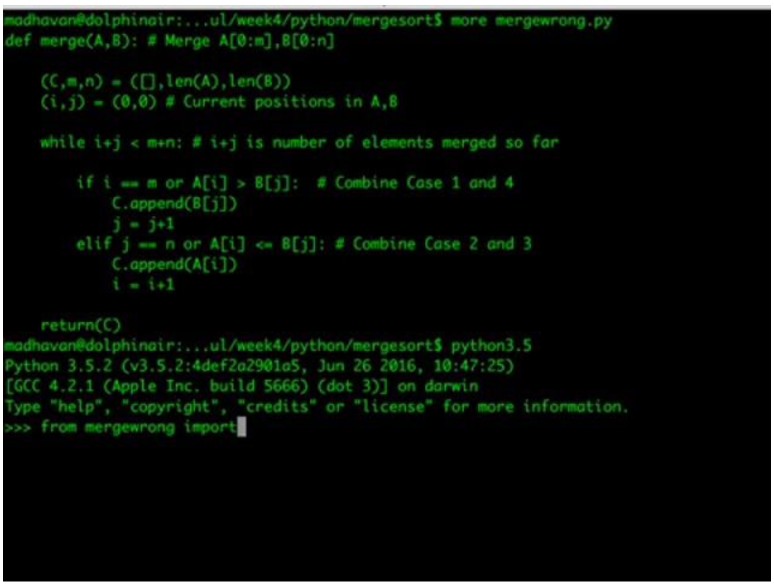
(Refer Slide Time: 14:57)




If we combine these cases then we can combine case 1 and 4. Remember one and four are the ones where we take the value from B. So, we combine 1 and 4 and say either if A is empty or if B has a smaller value then you take the value from B and append it to C and say j equal to j plus 1.

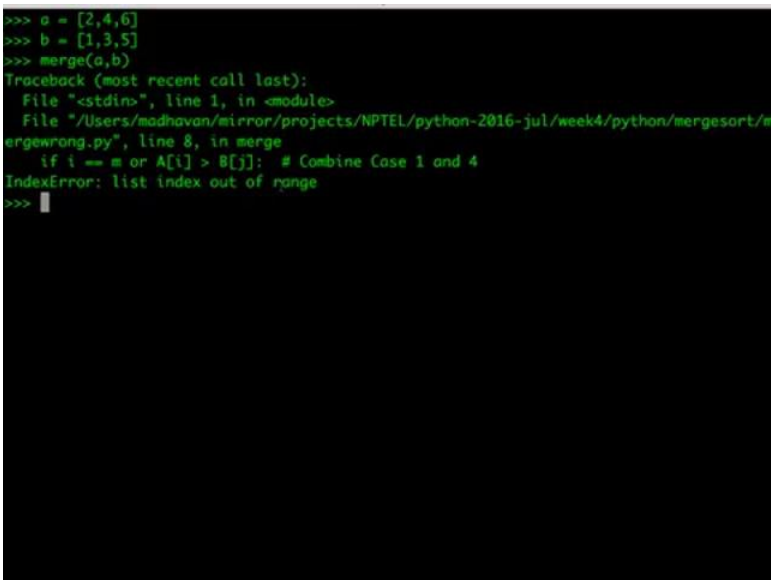
On the other hand, either if B is empty or A has a smaller value then you take the value from A and append the index in that right. Let us see what happens if we try to run this code.

(Refer Slide Time: 15:20)



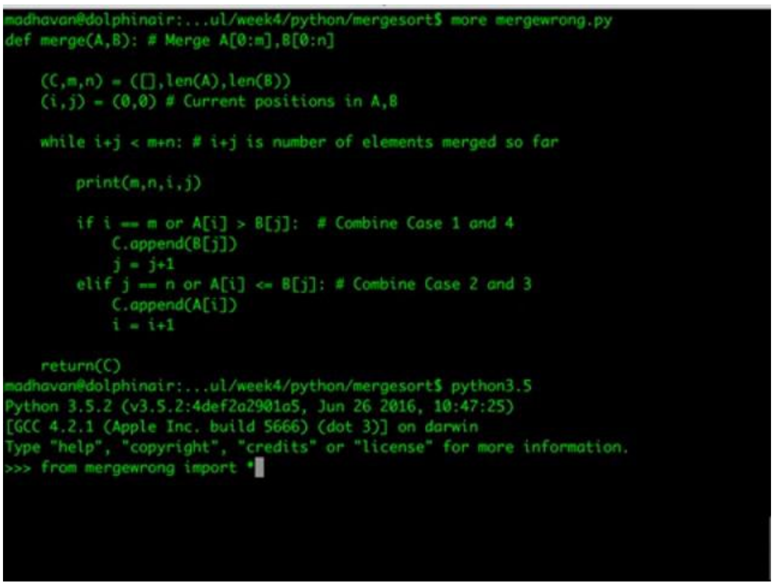
Here we have a file merge wrong dot py, the name suggests that is going to be a problem, where we have combined case 1 and 4, where we append the element from B into C and 2 and 3 where we combine the element append the element from A into C. Let us run this and see. Now, we take merge wrong at the starting point.

(Refer Slide Time: 15:46)



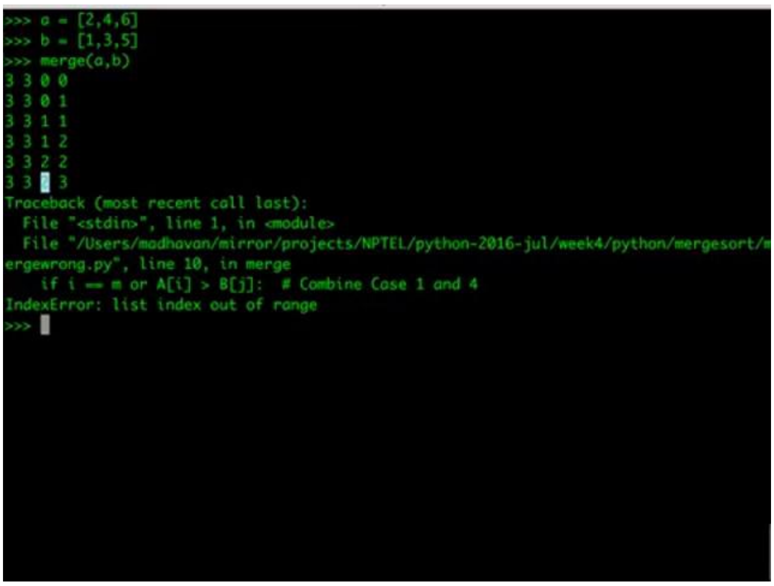

And let us just do a simple case. Supposing we take a as 2, 4, 6 and b as 1, 3, 5 then we have to expect 1, 2, 3, 4, 5, 6. Let us try to merge a and b right and now we get an error which says that we have a list index out of range. List index out of range suggests that we are trying to access some element which is not present and it so happens that this is in the case if i equal to m or a i equal to b j, greater than b j. Let us see if we can diagnose what is going on.

(Refer Slide Time: 16:25)




One simple way of diagnosing what is going on is to just insert some statements to printout the values of the names at some appropriate point. Now here since we are having an error at inside the while loop, what we have done is we have added the statement print which as I said we have not formally seen we will see it in the next week. But it does the intuitive thing it take the names m, n, i and j and prints them out in that sequence on the screen, so that we can see what is happening. Let us now run this again. So, we run the interpreter load this updated version of merge wrong.

(Refer Slide Time: 17:03)



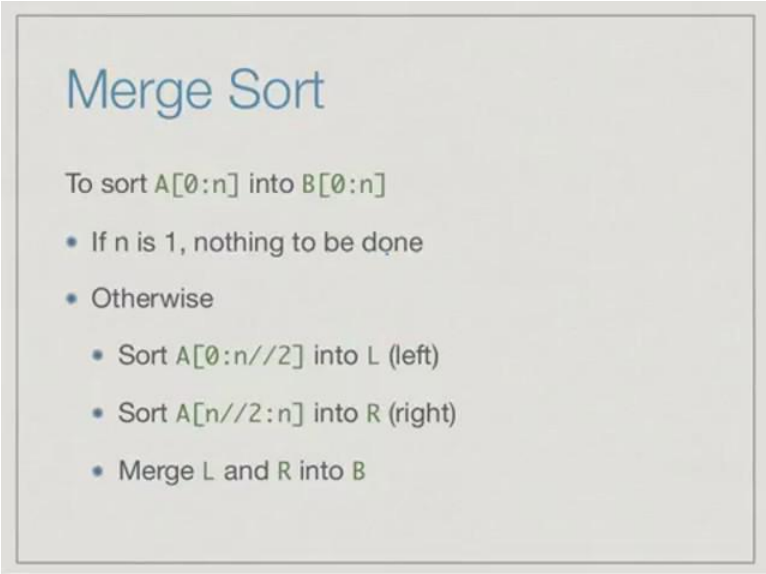
Setup a and b as before. So, a is 2, 4, 6; b is 1 3 5. And now we run merge. And now we see what is happening. So, m and n are the initial lengths 3 and 3. And these are the values 0 and 0 are i and j the pointers. So, j becomes 1 then i become 1 and so on.

At this point, this is where the problem is right. What we have found is that if i is equal to n or a i greater than b j, so i is not equal to m. So, i is not yet 3, so then it is trying to check whether a i is bigger than b j, but at this point unfortunately j is n. If we had had these cases in order we would have first checked if i is 3 then if j is 3 and only if neither of them are 3 would be a try to compare them. Because now we are only checking if i is 3, since i is not 3 we are going at and checking the value at a i against b j, but unfortunately j has become 3 already and we have not checked it yet.

By combining these two cases, we have allowed a situation where we are trying to compare a i and b j where one of them is a valid index and the other is not a valid index. Although it looks tempting to combine these two cases one has to be careful when doing so especially when we have these boundary conditions when we are indexing list, we must make sure that the index we are trying to get to is a valid index. And sometimes it is implicit and sometimes we have to be careful and this is one of those cases where you have to be careful and not optimize these things.

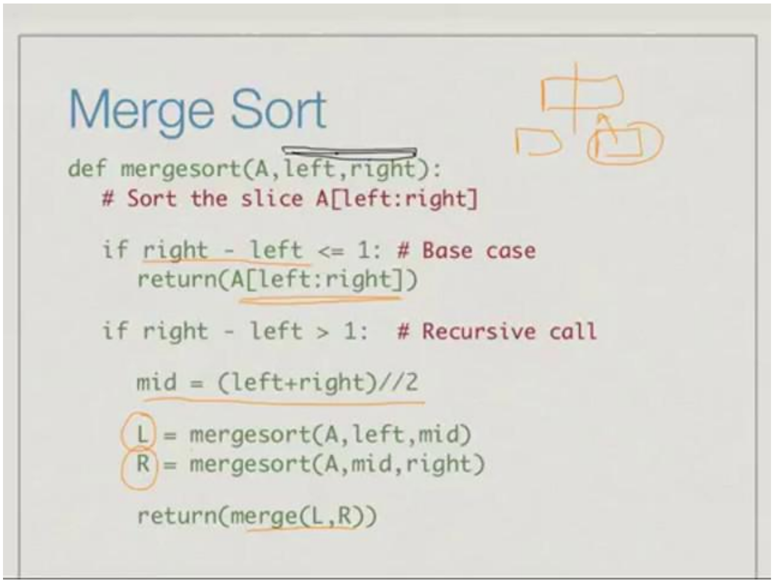
Otherwise, we have to have a separate condition saying if i is equal to m or j is less than n and which becomes more complicated than the version we had with four explicit cases. So, we may as well go back to the version with four explicit cases.

(Refer Slide Time: 18:52)




Now that we have seen how to merge the list, let us sort them. So, what we want to do is take a list of elements A and sort it into an output list B. So, if n is 1 or 0 actually, so if it is empty or it has got length 1, we have nothing to do; otherwise, we will sort the first half into a new list L and sort the second half into a new list R. L for left and R for right. And then we will apply the earlier merge function to obtain the output list B.

(Refer Slide Time: 19:31)

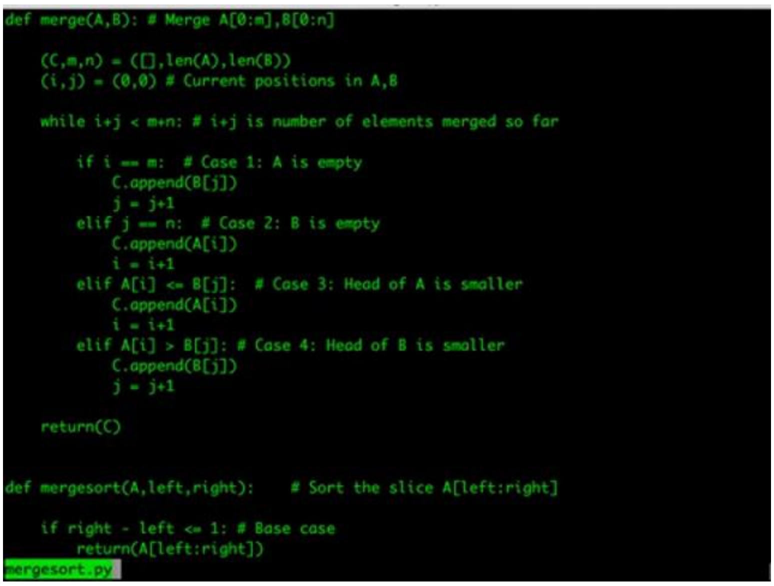



This is a very simple function except that we are going to be sorting different segments or slices of our list. So, we will actually have merge sort within input list and the left and right endpoints of the slice to be sorted. If the slice of length 1 or 0 then we just return the slice as it is. It is important that we have to return that part of the slice and not the entire part A, because they are only sorting.

Remember when we broke up something into two parts, for example, right, so then at this point we have to return the sorted version of this slice, not the entire slice. So, we have to return A from left to right if it is a base case; otherwise, we find the midpoint then we sort recursively sort the portion from the left hand side of the current slice to the midpoint, we put it in L. Then we take the midpoint to the right, put it in R and we use our earlier function merge to get a sorted list out of these two parts L and R and return this. This is a very straight forward implementation; there are no tricks or pitfalls here.

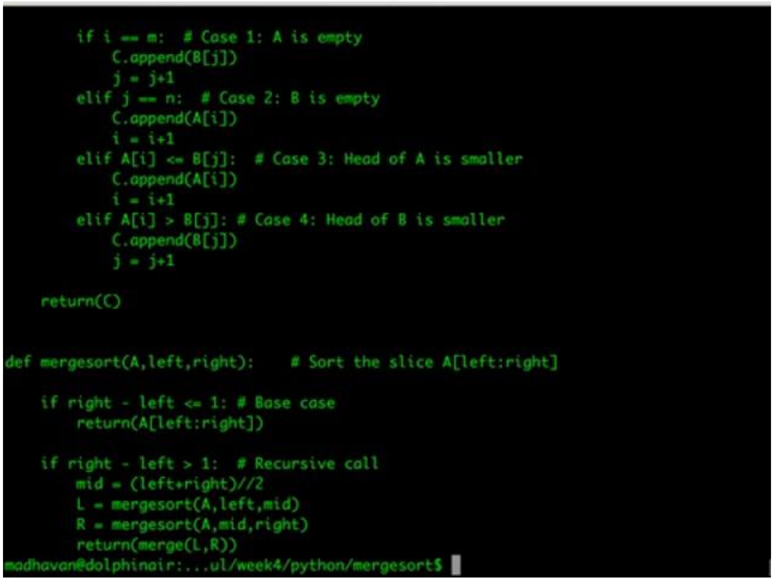
The only thing to remember is that we have to augment our merge sort function with these two things, the left point and the right point. We had a similar thing if you remember for binary search, where we recursively kept having the interval to search. So, we have to keep telling it in which interval we are searching.

(Refer Slide Time: 20:53)



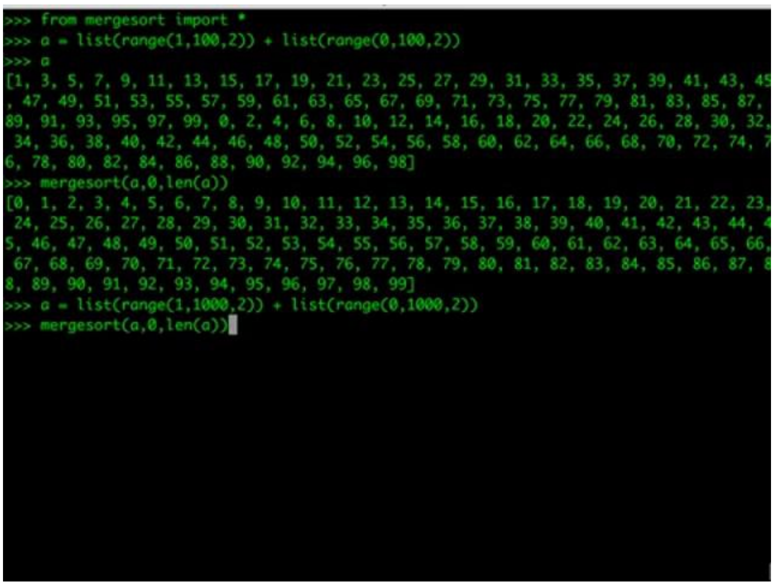

Let us look at a python implementation of merge sort. So, here we have a file merge sort dot py. We start with the function merge, which we saw before with a four way case split in order to shift elements from A and B to C.

(Refer Slide Time: 21:06)



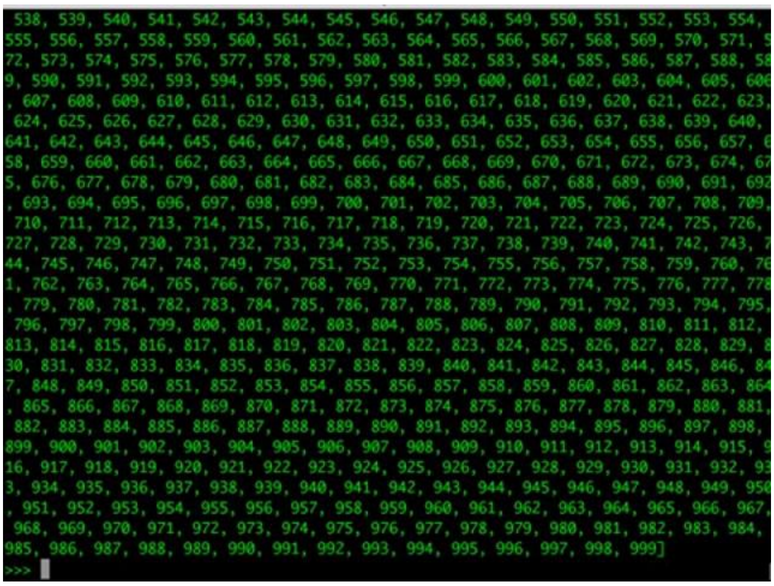

And then we add at the bottom of the file, the new function merge sort which we saw on the previous slide which takes a slice of A from left to right and sorts it. If it is a trivial slice, it returns the slice as it is. Otherwise, it breaks it into two parts and recursively sorts that. Let us see how this would actually run.

(Refer Slide Time: 21:25)




We take python and we say from merge sort, import all the functions. And now let us take a larger range. Suppose, if we take all the odd numbers followed by all the even numbers. So, we say range 1 to 100 in steps of 2. Those are the odd numbers and then all the even numbers in the same range say. So, A has now odd numbers followed by even numbers. You would imagine that if I sort this from 0 to the length of A, then you get the numbers sorted in sequence.

(Refer Slide Time: 22:12)

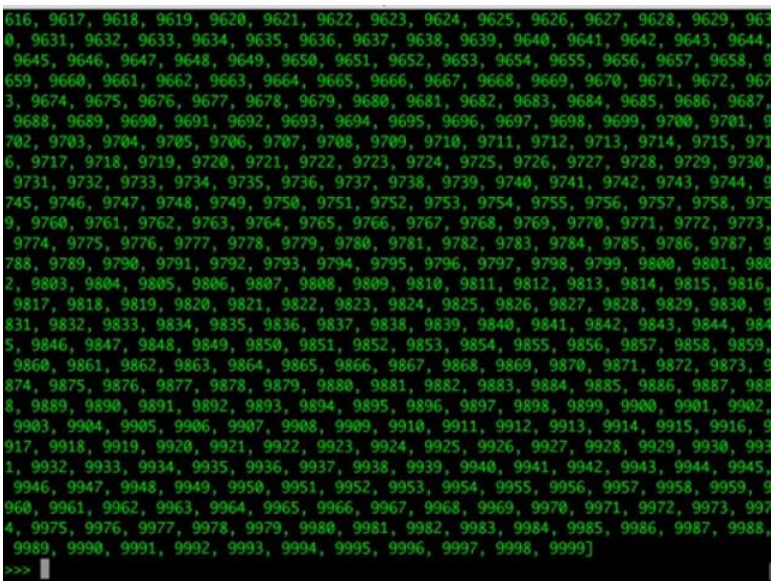



Now what if I take a larger list 1000 then I get, again everything sorted.

(Refer Slide Time: 22:16)


Now our claim is that this is an order n log n algorithm. It should work well for even bigger list. If I say 10000 which remember would take a very long time with insertion sort or  selection sort.

(Refer Slide Time: 22:26)

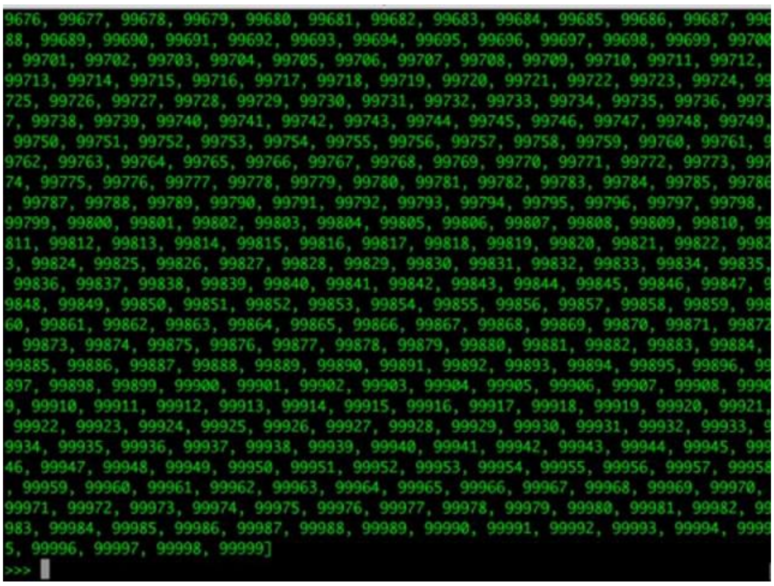



Question is how long it takes here and it comes out quite fast.

(Refer Slide Time: 22:29)


We can go further and say 100000 for example, and even here it comes reasonably fast.

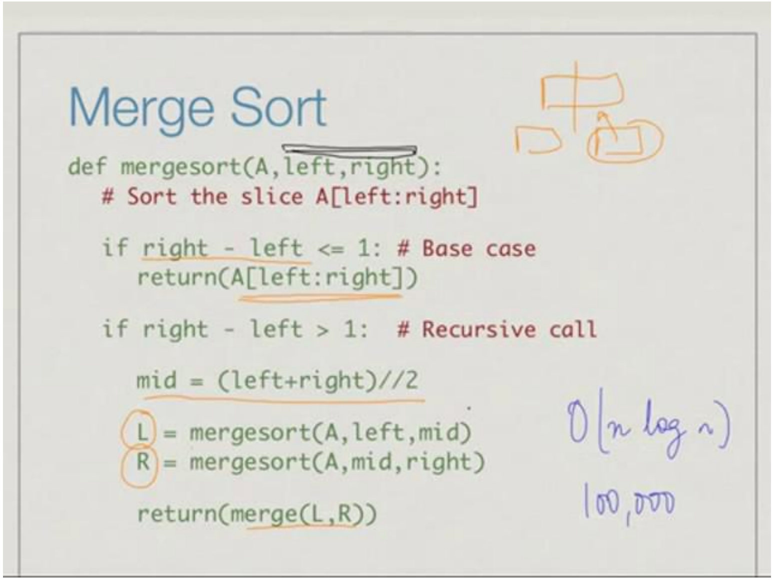
(Refer Slide Time: 22:37)




So, we can see that we are really greatly expanded the range of lists that we can sort when moving to n log n algorithm because now merge sort can handle things which are 100 times larger 100,000 as suppose to a few 1000 then insertion sort or selection sort. Another small point to keep in mind is notice that we did not run it to its recursion limit problem that we had with the insertion sort which we defined recursively. There for each element in the list, we were making a recursive call. If we had 1000 elements, we have making a 1000 recursive calls and then we have to increase the limit.

Now here even for 100,000 we do not have the problem and the reason is that the recursive calls here are not one per element, but one per half the list. So, we are only making log n recursive calls. So, 100,000 elements also requires only log 100,000. Remember a log 1000 is about 10. So, we are making less than 20 recursive calls, so we do not have a problem with the recursion limit, we do not have any pending recursions of that depth in this function.

(Refer Slide Time: 23:45)




We have seen merge sort in action and we have claimed without any argument that is actually order n log n and demonstrated that it works for inputs of size 100,000. In the next lecture, we will actually calculate why merge sort is order n log n.

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

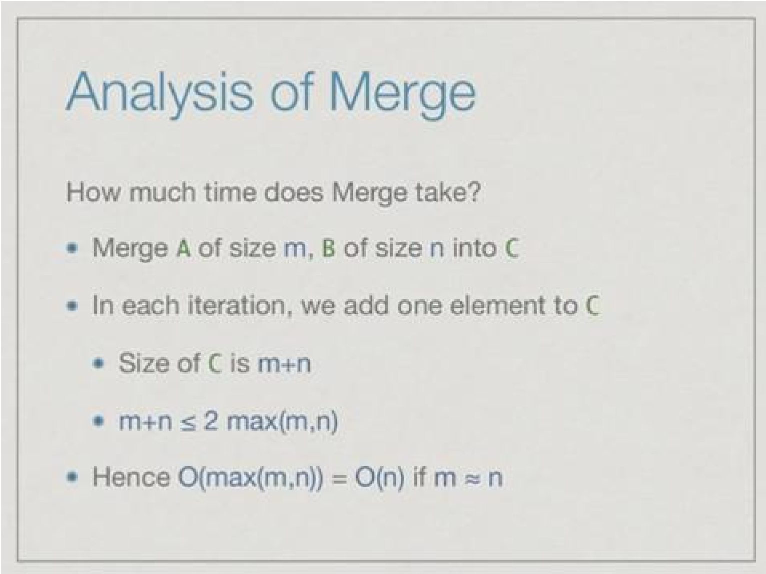
**Week - 04**
**Lecture - 02**
**Merge Sort, Analysis**

In the last lecture we looked at Merge Sort and we informally claimed that it was much more efficient than insertion sort or selection sort and we claimed also that it operates in time order n log n.

(Refer Slide Time: 00:03)


Recall that merge sort as its base a merging algorithm which takes two sorted lists, A and B and combines them one at a time by doing a scan over all elements.

(Refer Slide Time: 00:29)

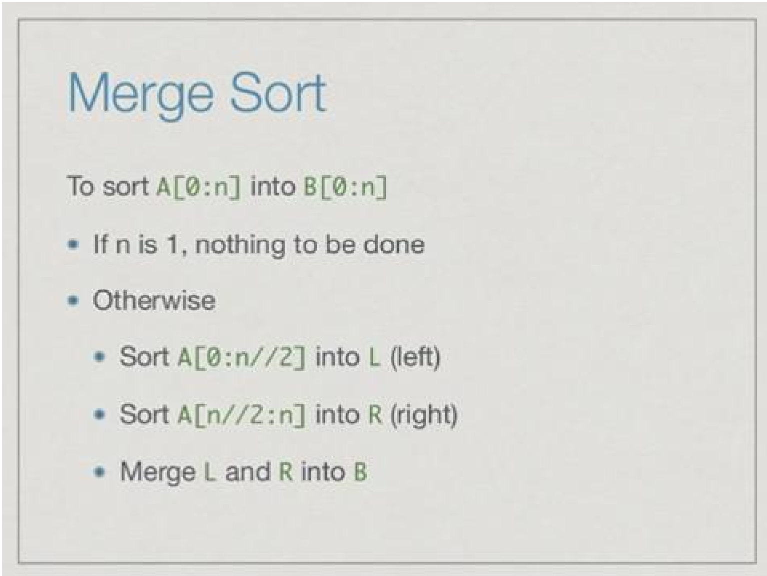



In order to analyze merge sort, the first thing we need to do is to give an analysis of the merge function itself. How much time does merge take in terms of the input sizes of the two lists, A and B. So, suppose A has m elements and B has n elements and we want to put all these elements together into a single sorted list in C. Remember that we had an iteration where in each step of the iteration we looked at, the first element in A and B and move the smaller of the two to C. So clearly C grows by one element with each iteration and since we have to move all m plus n elements from A and B to C, the size of C is m plus n.

What do we do in each iteration? Well we do a comparison and then, we do an assignment and then, we increment some indices. So, this is a fixed number of operations. So, it is a constant. So, the total amount of work is proportional to m plus n. Notice that m plus n is at most twice the maximum of m plus n. So, m is 7 and n is 15, then 5 plus 7 plus 15 will be less than two times 15.

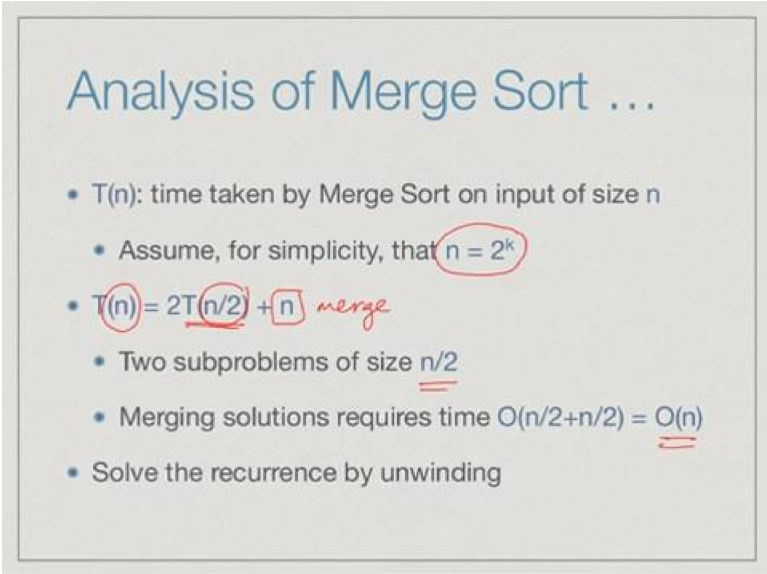
We can say that merge as a function takes time of the order of maximum of m and n and in particular very often like in merge sort, we are taking two lists of roughly the same size like we divide a list into two halves and then, we merge them. If both m and n are of the same, approximately the same size, then the max of m and n is just one of them in itself. Essentially merge is linear in the size of the input list.

(Refer Slide Time: 02:06)




Now, having analyzed merge, let us look at merge sort. So, merge sort says that if a list is small, has zero or one elements, nothing is to be done. Otherwise you have to solve the problem for two half lists and then, merge them.

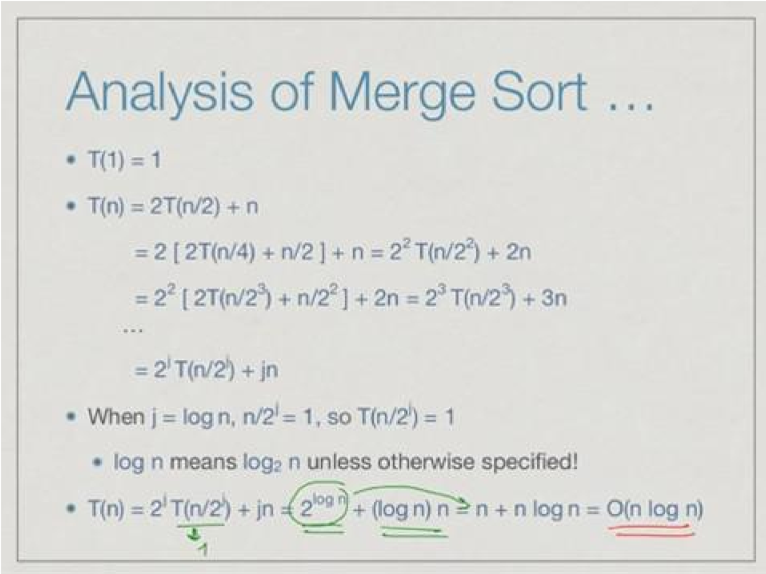
(Refer Slide Time: 02:23)




As with any recursive function, we have to describe the time taken by such a function in terms of a recurrence. So, let us assume for now since we are going to keep dividing by 2 that we can keep dividing by 2 without getting an odd number in between.

Let us assume that the input n is some perfect power of 2. It is n is 2 to the k. When we breakup merge sort into two lists, we have two lists of size n by 2. The time taken for n elements is two times time taken for two list of n by 2 and this is the merge component. We have an order n step requires us to merge two lists of size n by 2 and remember we just said that merge is linear in the size of the input. So, we have two sub problems of size n by 2, that is two times n by 2 and we have merging which requires order n. As with binary search and with recursive insertion sort, we can solve this recurrence by unwinding it.

(Refer Slide Time: 03:33)

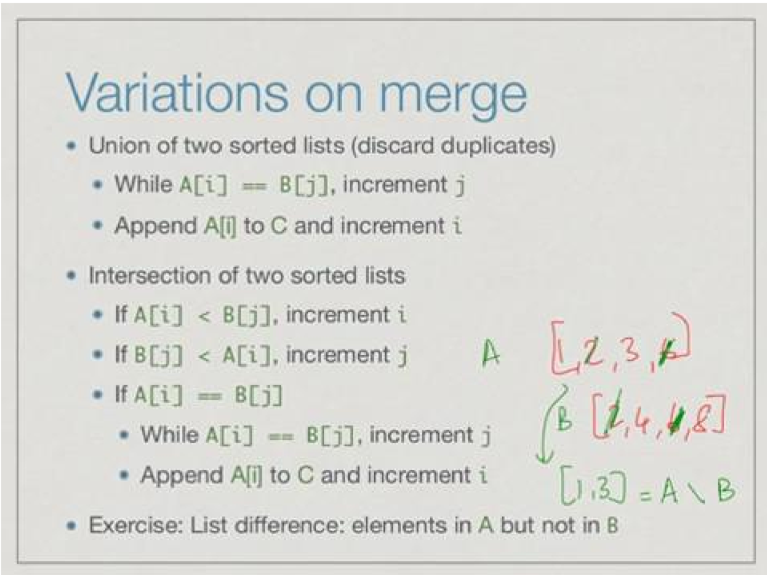



We start with the base case. If we have a list of size 1, then we have nothing to do. So, T of n is 1 and T of n in general is two times T n by 2 plus n. If we expand this out, we read substitute for T n by 2 and we get two times T n by 4 plus n by 2 because that if we take this as a new input, this expands using the same definition and if we rewrite this. So, we write two times 2 as 2 square and we write this 4 as two squared. We will find that this is equivalent to writing it in this form 2 into 2, 2 squared T n by 2 square and now notice that you have two times n by 2 over here. This 2 and this 2 will cancel. So, we have one factor n and the other factor of n. The important thing is that you have 2 here in the exponent and you have 2 here before the n.

Now, like wise what we will do in the next step is to expand this two times T n by 4. So, we expand two times T n by 4 and that will give us another n by 8 which you write as n 2 cube which used to be 2 squared 2 n by 2 square plus 2 n will turn out to be 2 cubed 2 n by 2 cube plus 3 n. So, notice that the two’s have become threes uniformly. So, in this way if we keep going after k steps or j steps, we will have 2 to the j times T n by 2 to the j plus j times n. Now, how long do we keep doing this? We keep doing this till we hit the base case. So, when j is log of n, where log by log we usually mean log to the base 2, then n by 2 to the j will be 1. So, T by T of n by 2 to the j will also be 1.

After log n steps, this expression simplifies to 2 to the log n plus log n times n everywhere we have a j, we put a log n and take this has become 1, so it has disappeared. We have 2 to the j is 2 to the log n plus this j has log n and then, we have n and this is 2 to the log n by definition is just n. So, 2 to the log n is n and we have n log n and by our rule that we keep the higher term when we do, we go n log n is bigger than n. We get a final value of O n log n for merge sort.

(Refer Slide Time: 06:00)




Merge turns out to be a very useful operation. What we saw was to combine two lists faithfully into a single sorted list inparticular our list if we had duplicates. So, if we merge say 1, 3 and 2, 3, then we end up with the list of the form 1, 2, 3, 3. This is how merge would work. It does not lose any information. It keeps duplicates and faithfully copies into the final list.

On the other hand, we might want to have a situation where we want the union. We do not want to keep multiple copies and we want to only keep single copy. In the union case here is what we would do. Let us assume that we have two lists and in general, we could have already duplicates within the lists. Let us suppose that we have 1, 2, 2, 6 and 2, 3, 5 then we do the normal merge. So, we move one here and now, when we hit two elements which are equal, right then we need to basically scan till we finish this equal thing and copy one copy of it and then finally, we will put 3 and then, 5 and then 6.

When A and A i is equal to B j will increment both sides and make sure that we go to the end of that block. The other option is to do intersection. Supposing we want to take 1, 2, 6 and 2, 6, 8 and come out with the answer 2, 6 as the common elements, then if one side is smaller than the other side, we can skip that element because it is not there in both lists. So, if A i is less than B j we increment i, if B j is less than A i, we increment j and if they are equal we will take union, we keep one copy of the common element.

So, merge can be used to implement various combinations, combination operations on this. It can be used to take the union of two lists and discard duplicates. It can be used to take the intersection of two lists and finally, as an exercise to test that you understand it and see if you can use merge to do list difference.

List difference is a following operation. If I have say 1, 2, 3, 6 and I have 2, 4, 6, 8 then this difference is all the elements in the first list which are not there in the second list. Two is there here and it is here, you remove 2, 6 is there here and here remove 2. So, you should get 1, 3. So, if this is A, and this is B, then this is so-called list difference A minus B. So, see if you can write a version of merge which gives you all the elements in A which are not also in B, also known as list difference.

(Refer Slide Time: 08:49)


Now, merge sort is clearly superior to insertion sort and selection sort because it is order n log n, it can handle lists as we saw of size 100,000 as opposed to a few thousand, but it does not mean that merge sort does not have limitations. One of the limitations of merge sort is that we are forced to create a new array, every time we merge two lists. There is no obvious way to efficiently merge two lists without creating a new list. So, there is a penalty in terms of extra storage. We have to double this space that we use when we start with lists and we want to sort it within merge sort.

The other problem with merge sort is that it is inherently recursive and so, merge sort calls itself on the first half and the second half. Now, this is conceptually very nice. We saw that recursive definitions rec recursive functions are very naturally related to inductive definitions and they help us to understand the structure of a problem in terms of smaller problems of the same type.

Now, unfortunately a recursive call in a programming language involves suspending the current function, doing a new computation and then, restoring the values that we had suspended for the current function. So, if we currently had values for local names like i, j, k, we have to store them somewhere and then, retreat them and continue with the old values when the recursive call is done. This requires a certain amount of extra work.

Recursive calls and returns turn out to be expensive on their own time limits. So, it could be nice if we could have both the order n log n behavior of merge sort. And we could do away with this recursive thing, but this is only a minor comment. But conceptually merge sort is the basic order n log n sorting algorithm and it is very useful to know because it plays a role in many other things indirectly or directly.

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

**Week - 04**
**Lecture - 03**

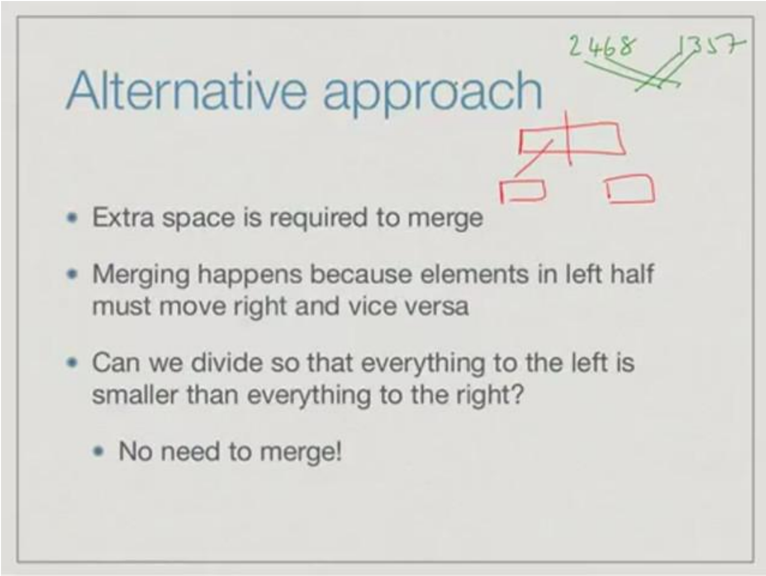
# Quicksort

We saw that merge sort is an order n log n sorting algorithm.

(Refer Slide Time: 00:01)


But it has a couple of deficiencies which make it sometimes impractical. The main problem is that it requires extra space for merging them. We also saw that it is difficult to implement merge sort without using recursion and recursion carries its own cost in programming language.

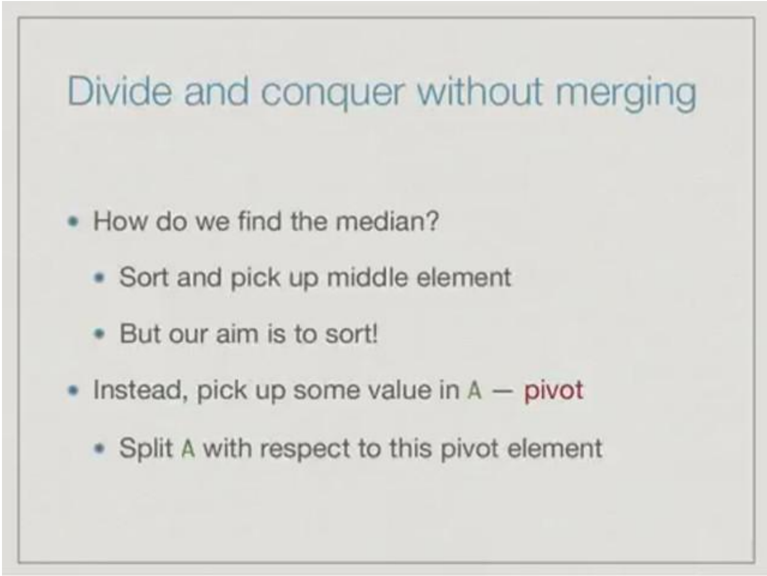
(Refer Slide Time: 00:23)




Let us address the space problem. The extra space required by merge sort is actually required in order to implement the merge function and why do we need to merge? The reason we need to merge is that when we do a merge sort, we have the initial list and then we split it into two parts, but in general there may be items in the left which are bigger than items in the right.

For instance, if we had say even numbers in the left and the odd numbers on the right then we have to merge by taking numbers alternatively from either side. So, if we could arrange that everything that is on the left side of our divided problem is smaller than everything on the right side of the divided problem, then we would not need to merge at all and this perhaps could save us this problem of requiring extra space for the merge.

(Refer Slide Time: 01:19)



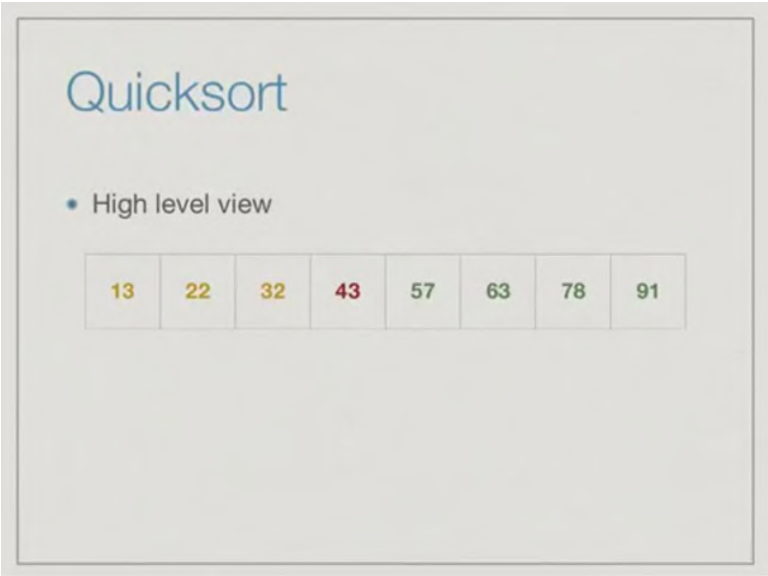

How would we do divide and conquer without merging. Assume that we knew the median value; remember the median value in a set is the value such that half the elements are smaller and half are bigger. We could move all the values smaller than the median to the left half and all of those bigger than the median to the right half. As we will see this can be done without creating a new array in time proportional to the length of the list.

Having done this rearrangement moving all the smaller values to the left half and the bigger values to the right half then we can recursively apply this divide and conquer strategy and sort the right and the left half separately and since we have guaranteed that everything in the left half is smaller than everything in the right half, this automatically means that after this divide and conquer step we do not need to combine the answers in any non trivial way because the left half is already below the right half. So, we do not need to merge.

If we apply this strategy then we would get a recursive equation exactly like merge sort. It would say that the time required to sort a list of length n requires us to first sort two lists of size n by 2 and we do order n not for merging, but in order to decompose the list so that all the smaller values are in the left and in the right. So, rearranging step before we do the recursive step is what is order n, whereas merge was the step after the recursive step which was order n in the previous case, but if we solve the recurrence, its the same one, we get another order n log n algorithm.

The big bottleneck with this approach is to find the median. Remember that we said earlier that one of the benefits of sorting a list is that we can identify the median as the middle element after sorting. Now here, we are asking for the median before sorting, but our aim is to sort, it is kind of paradoxical. If we are requiring the output of the sorting to be the input to the sorting. This means that we have to try the strategy out with a more simplistic choice of element to split the list. Instead of looking for the median we just pick up some value in the list A, and use that as what is called a pivot element. We split A with respect to this pivot so that all the smaller elements are to the left and all the bigger elements are to the right.

(Refer Slide Time: 03:58)

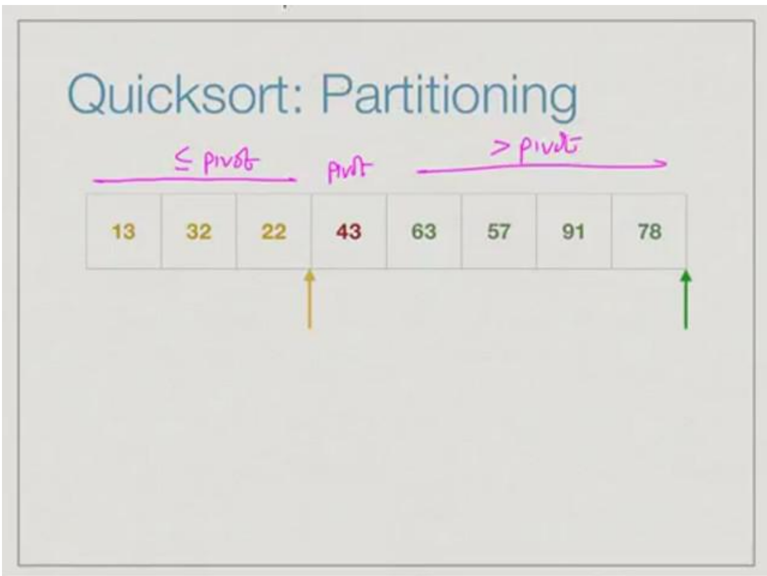



This algorithm is called Quicksort, it was invented by a person called C.A.R Hoare in the 1960s and is one of the most famous sorting algorithms. So, we choose a pivot element which is usually the first element in the list of the array. We partition A, into the lower part and the upper part with respect to this pivot element. So, we move all the smaller elements to the left and all the bigger elements to the right with respect to the choice of pivot element, and we make sure the pivot comes between the two because we have picked up the first element in the array to pivot. So, after this we want to move it to the center between the lower and the upper part and then, we recursively sort two partitions.

Here is a high level view of how quicksort will work on a typical list. Suppose this is our list, we first identify the beginning of the list, the first element as the pivot element. Now, for the remaining elements we have to figure out which ones are smaller and which ones are bigger. So, without going into how we will do this, we end up identifying 32, 22 and 13 as three elements which are smaller and marked in yellow and the other four elements which are marked in green are larger.

The first step is to actually partition with respect to this criterion. So, we have to move these elements around so that they come into two blocks. So that, 13, 32 and 22 come to the left; 63, 57, 91 and 78 come to the right and the pivot element 43 comes in middle. This is the rearranging step and now we recursively sort the yellow bits and the green bits, then assuming we can do that, we have a sorted array and notice that since all the yellow things are smaller than 43 and all the green things are bigger than 43, no further merging is required.

(Refer Slide Time: 05:51)



So let us look at how partitioning works. Here, we have the earlier list and we have marked 43 as our pivot element and we want to do a scan of the remaining elements and divide them into two groups; those smaller than 43, the yellow ones; those bigger than 43, the green ones and rearrange them. What we will do is we will keep two pointers; a yellow pointer and a green pointer and the general rule will be that at any given point we will have at some distance, the yellow pointer which I will draw in orange to make it more visible and the green pointer.

These will move in this order; the orange pointer or the yellow pointer will always be behind the green pointer and the inductive property that we will maintain is that these elements are smaller than or equal to 43, these elements are bigger than 43 and these elements are unknown. What we are trying to do is, we are trying to move from left to right and classify all the unknown elements; each time we see an unknown element we will shift the two pointers so that we maintain this property that between 43 and the first pointer we have the elements smaller than or equal to 43; between the first pointer and the second pointer we have the element strictly greater than 43 and to the right of the green pointer, we have those which are yet to be scanned.

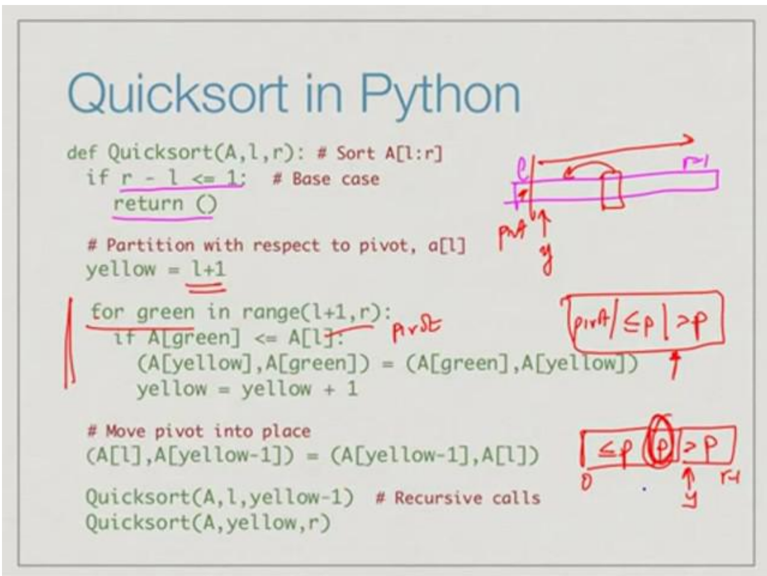
Initially nothing is known then we look at 32, since 32 is smaller than the 43, we move the yellow pointer and we also push the green pointer along. So, the unknown things start from 22, and there is nothing between the yellow and the green pointer indicating we have not yet found the value bigger than 43, same happens for 22. Now, when we see 78, we notice that 78 is bigger than 43. Now, we move only the green pointer and not yellow pointer, we have these three intervals as before. Remember that this is the part which is less than equal to 43; this is the part that is greater than 43 and this part is unknown. We continue in this way.

Now, we look at 63, again 63 extends the green zone, 57 extends the green zone, 91 extends the green zone. Now, we have to do something when we find 13. So, 13 is an element which has to be put into the yellow zone, one strategy would be to do a lot of shifting. We move 13 to where 22 is or after 22 and we push everything from 78 onwards to the right, but actually a cleverer strategy is to say that 13 must go here. So, we need to make space, but instead of making space we can say, it does not matter to us, we are eventually going to sort the green things anyway.

How does it matter which way we sort that, we will take this 78 and just move it to 13. So, instead of doing any shifting, we just exchange the first element in the green zone with the element we are seeing so far, that automatically will extend both yellow zone and the green zone correctly. So, our next step is to identify 13 as smaller than 43 and swap it with 78. Now, we have reached an intermediate stage where to the right of the pivot we have scanned everything and we have classified them into those which are the smaller ones and those which are the bigger ones.

Now, it remains to push the yellow things to the left of 43. Once again we have the same problem we saw when we included 13 in the yellow zone. If we move 43 to the correct place then we have to move everything here to the left, but instead we can just take this 13 in the last element to the yellow zone and replace it there and not shift 32 and 22. This disturbs the order, but anyway this is unsorted, it just remains unsorted. So, we do this and now we have the array rearrange as we wanted, all of these things to the left are smaller than the pivot the pivot is in the middle and everything to the right is bigger than the pivot.

(Refer Slide Time: 09:47)



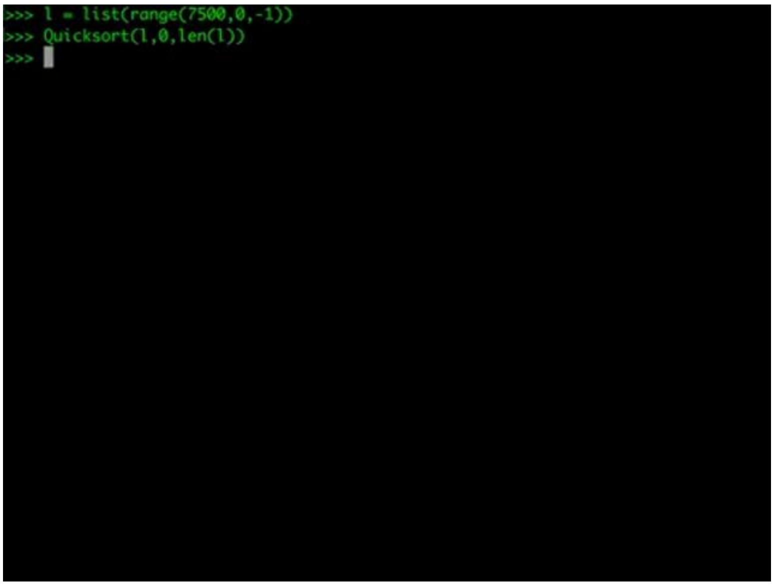
Here is an implementation in Python. So, remember that quicksort is going to have like merge sort and like binary search, we repeatedly apply it in smaller and smaller segments. In general, we have to pass to it the list which we call A, and the end points to the segment the left and the right. If we have something that we are doing a slice l to r minus 1, if this slice is 1 or 0 in length, we do nothing otherwise we follow this partitioning strategy we had before, which is that we are sorting from l to r minus 1. The position l, this is the pivot.

We will initially put the yellow pointer here, saying that the end of the yellow zone is actually just the pivot, there is nothing there. So, yellow is l plus 1 and now we let green proceed and every time you see an element in the green the new green one which is smaller than the one which is the pivot. Remember this is the pivot, if ever we see a green the next value to be checked is smaller than or equal to A[l] we exchange so that we bring this value to the end of the yellow zone.

This is what we did to 13 and then we move the yellow pointer as well, otherwise if we see a value which is strictly bigger, we move only the green pointer which is implicitly done by the for loop and we do not move the yellow. At the end of this, we have the pivot then we have the less than equal to pivot and then we have the greater than. So, this is that intermediate stage that we have achieved at the end of this loop. Now, we have to find the pivot and move it to the correct place.

Remember that the yellow, yellow is pointing to the position beyond the last element smaller than that. So, yellow is always one value before, beyond this. So, we take the yellow minus 1 value and exchange it with the left value and now what we need to do is we have now less than p, p, greater than p and this is where yellow is. So, we need to go from 0 to yellow minus 1, we do not want to sort p again. Because p is already put in the correct place, so we quicksort from l to yellow minus 1 and from yellow to the right end.

(Refer Slide Time: 12:11)




Here, we have written the Python code that we saw in the slide in a file. You can check that it is exactly the same code that we had in the slide. We can try and run it and verify that it works. So, we call Python and we import this function. Remember that this is again a function which sorts in place. If you want to sort something and see the effect we have to assign it a name and then sort that name and check the name afterwards. Let us, for instance, take a range of values from say 500 down to 0 then if we say quicksort(l) then we have to of course, give it the end then l gets correctly sorted.

So, you cannot see all of it, but you can see from 83, 84 up to 102 up to 500. Now, we have the same problem that we had with insertion sort. If we say 1000 and then we try to quicksort this, we will get this recursion depth because as we will see, in the worst case actually, quicksort behaves a bit like insertion sort and this is a bad case. So, to get around this we would have to do the usual thing - we have to import the sys module and set the recursion limit to something superbly large, say 10000, maybe 100,000 and then if we ask it to quicksort there is no problem.

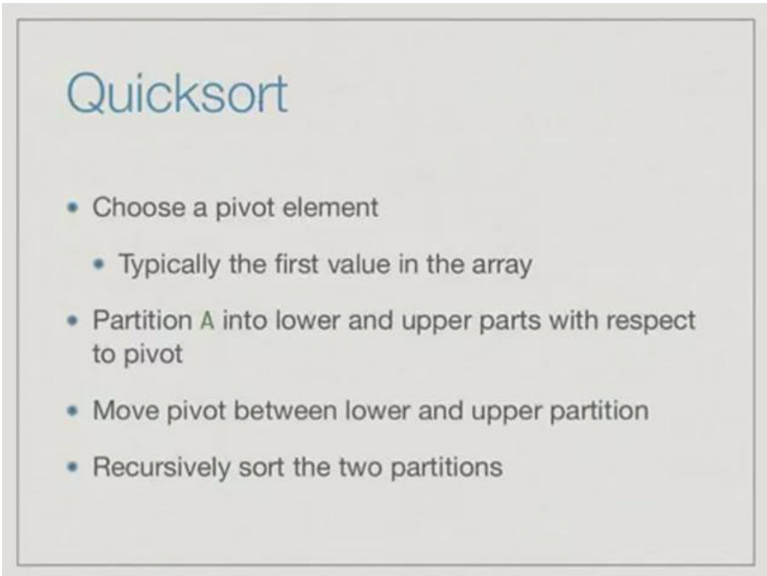
This is another case where this recursion limit in python has to be manually set and one thing we can see actually is that quicksort is not as good as we believe because if we were to, for instance, sort something of size say 7500 then it takes a visible amount of time. We saw that merge sort which was n log n could do 5000 and 10000 and even 100,000 instantaneously.

So, clearly quicksort is not behaving as well as merge sort and we will see in fact, that quicksort does not have an order n log n behavior as we would have liked and that is because we are not using the median, but the first value to speak. We will see that in the next lecture as to why quicksort is actually not a worst case order n log algorithm.

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

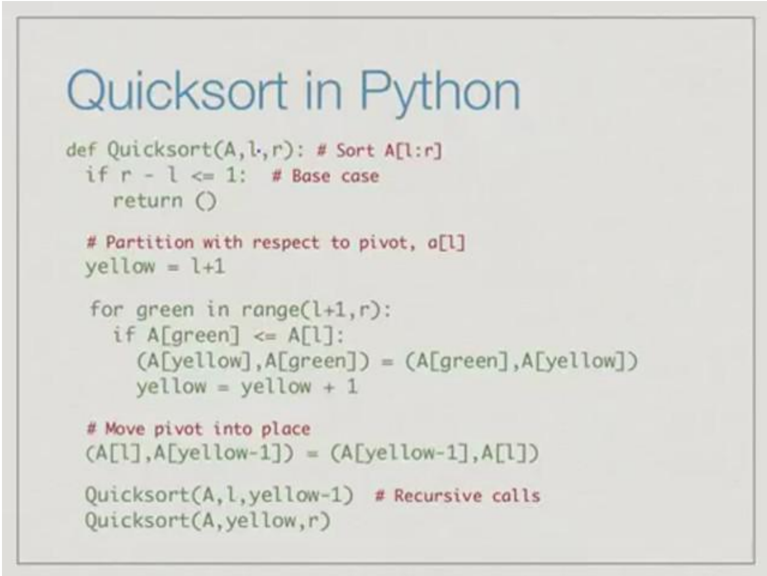
**Week - 04**
**Lecture - 04**
**Quicksort Analysis**

(Refer Slide Time: 00:01)



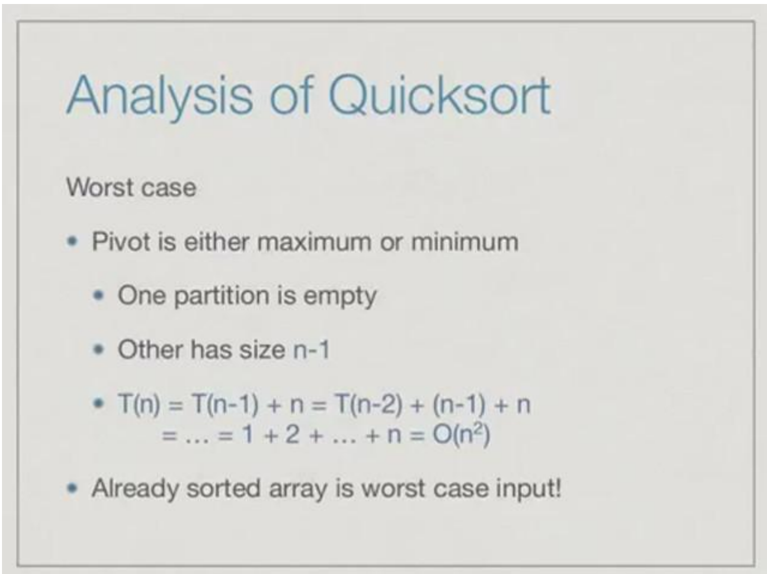

In the previous lecture, we saw quicksort which works as follows. You first rearrange the elements with respect to a pivot element which could be, well, say the first value in the array. You partition A, into the lower and upper parts, those which are smaller than the pivot, and those which are bigger than the pivot. You rearrange the array so that pivot lies between the smaller and the bigger parts and then you recursively sort the two partitions.

(Refer Slide Time: 00:31)




And this was the actual python code which we ran and we saw that it actually behaved not as well as merge sort even for the list of size 7500, we saw it took an appreciatively long time.

(Refer Slide Time: 00:42)



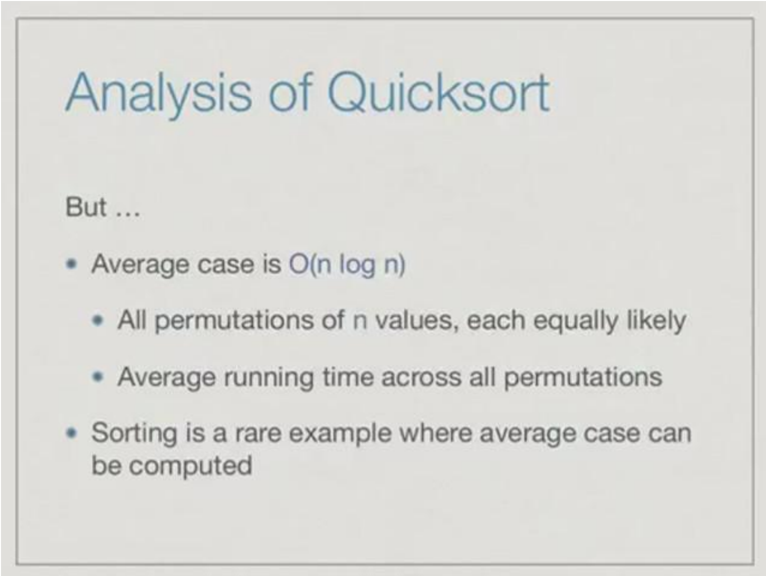

What is the worst case behavior of quicksort? The worst case behavior of quicksort comes when the pivot is very far from the ideal value we want. The ideal value we want is median, the median would split the list into two equal parts and there by divide the work into two equal parts, but if the pivot happens to be either the minimum or the maximum value in the overall array, then supposing it is the maximum then every other value will go into the lower part and nothing will go into higher part. So, the recursive call it consists of sorting of n minus 1 elements.

If one partition is empty, the other one has size n minus 1, this would happen if that the pivot is one of the extreme elements and if this happens and this is the worst case then in order to sort n elements, we have to recursively sort n minus 1 elements and we are still doing this order n work in order to rearrange the array because we do not find this until we have sorted out, gone through the whole array and done the partition.

This says t n is t n minus 1 plus n and if we expand this this comes out to be exactly the same recurrence that we saw for insertion sort and selection sort. So, t n would be 1 plus 2 up to n and this summation is just n into n plus 1 by 2 which would be order n square. The even more paradoxical thing about quicksort is that this would happen, if the array is sorted either in the correct order or in the wrong order.

Supposing you are trying to sort in ascending order and we already have an array in ascending order then the first element will be the smallest element. The split will give us an array of n minus 1 on one side and put 0 on the other side and this keep happening. The worst case of quicksort is actually an already sorted array. Remember, we saw that for insertion sort and already sorted array works well because the insert steps stops very fast. So, quicksort is actually in the worst case doing even worse than insertion sort and specifically on already sorted arrays.

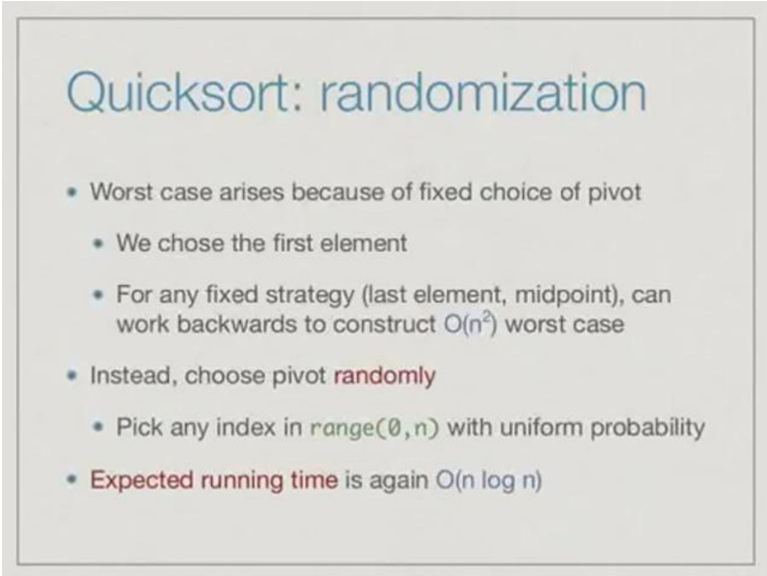
(Refer Slide Time: 02:40)




However, it turns out that this is a very limited kind of worst case and one can actually try and quantify the behavior of quicksort over every possible permutation. So, if we take an input array with end distinct values, we can assume that any permutation of these n values is equally likely and we can compute how much time quicksort takes in each of these different n permutations and assuming all are equally likely, if we average out over n permutations we can compute an average value.

Now, this sound simple but mathematically it is sophisticated and sorting is one of the rare examples where you can meaningfully enumerate all the possible inputs and probabilities of those inputs, but if you do this calculation it turns out that in a precise mathematical sense quicksort actually works in order n log n in the average. So, the worst case though it is order n square actually happens very rare

(Refer Slide Time: 03:41)

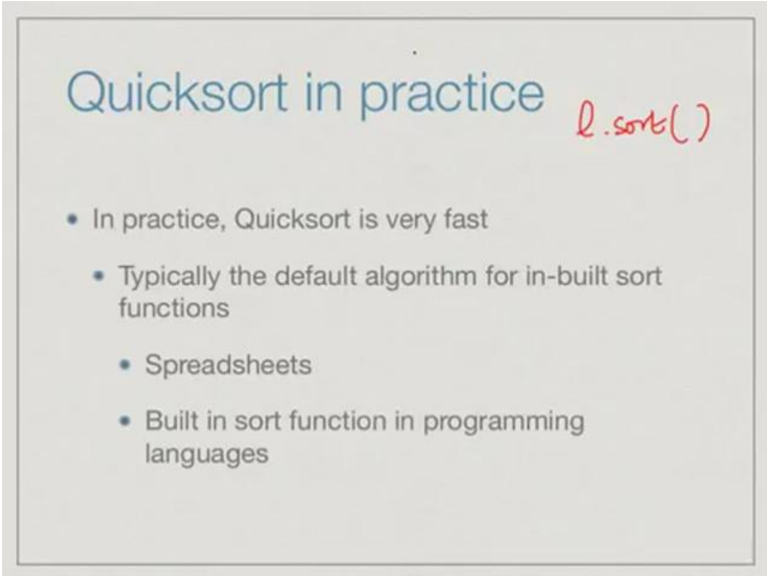



The worst case actually arises because of a fixed choice of the pivot element. So, we choose the first element as the pivot in our algorithm and so in order to construct the worst case input, if we always put the smallest or the largest element at the first position we get a worst case input. This tells us that a sorted input either ascending or descending is worst case for our choice object.

Supposing, instead we wanted to choose the midpoint we take the middle element in the array as a random then again we can construct a worst case input by always putting the smallest value and working backward so that at every stage, the middle value is the smallest or largest value. So, for any fixed choice of pivot, if we have a pivot, which is picked in a fixed way in every iteration, we can always reconstruct the worst case.

However, if we change our strategy and say that each time we call quicksort we randomly choose a value within the range of elements and pick that as the pivot, then it turns out that we can beat this order n squared worst case and get an expected running time, again an average running time probabilistically of order n log n.

(Refer Slide Time: 04:56)

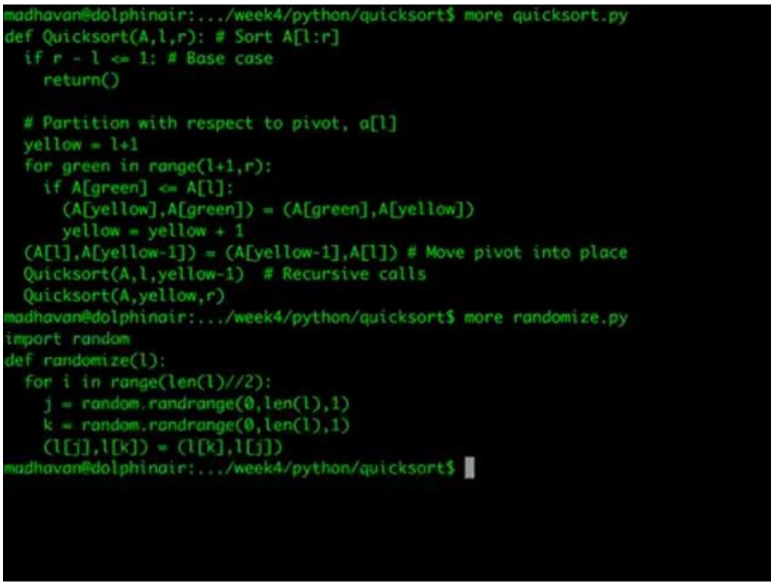



As a result of this because though it is worst case order n squared, but an average order n log n, quicksort is actually very fast. What we saw is it addresses one of the issues with merge sort because by sorting the rearranging in place we do not create extra space. What we have not seen in which you 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 quciksort.

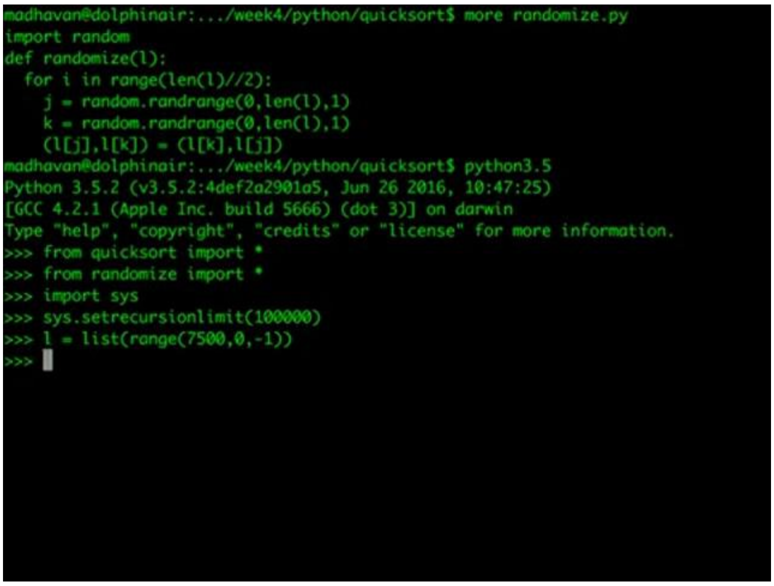
So, before we precede let us try and validate our claim that quicksort's worst case behavior is actually tied to the description of the worst case input as in already sorted list.

(Refer Slide Time: 06:22)




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.

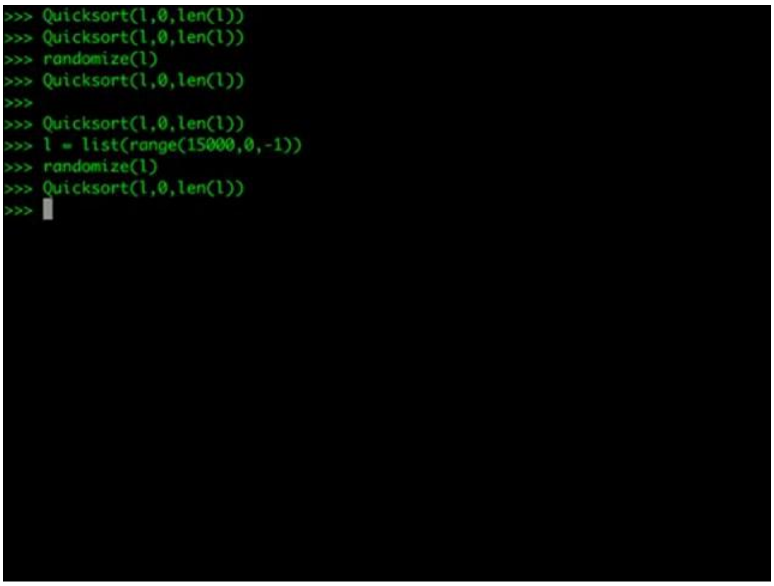
(Refer Slide Time: 06:52)



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)

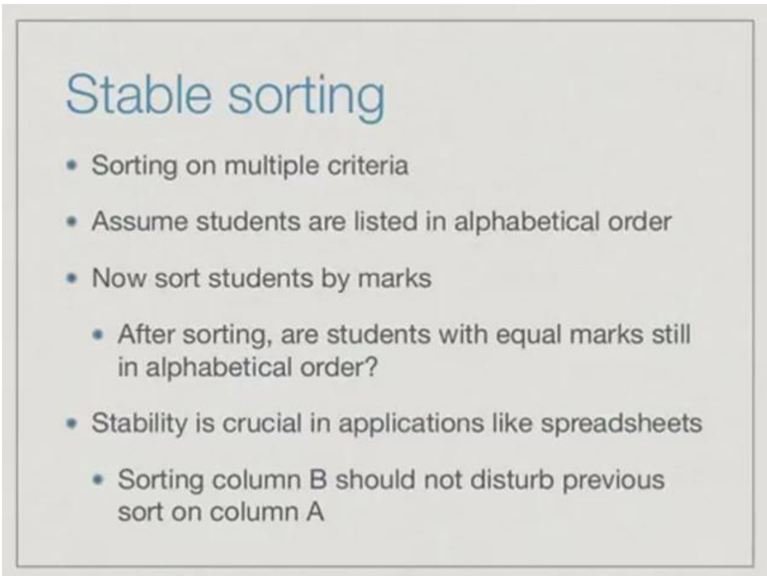



And what we saw was that, if we try to quick sort this list it takes a long time because it is 7500 and it is a worst case in. What we are going to try and do now is and the same thing will happen even after it is sorted because even after it is sorted is still a worst case input expect now it is an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

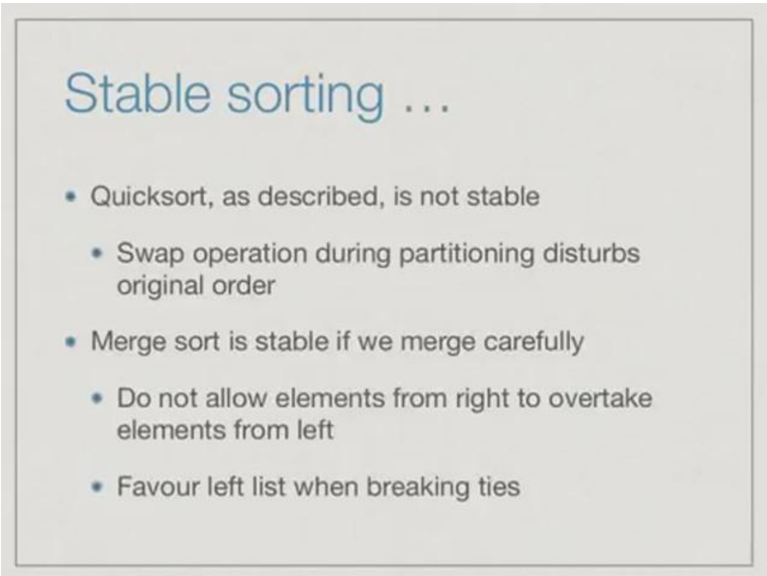
(Refer Slide Time: 09:54)




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)




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.

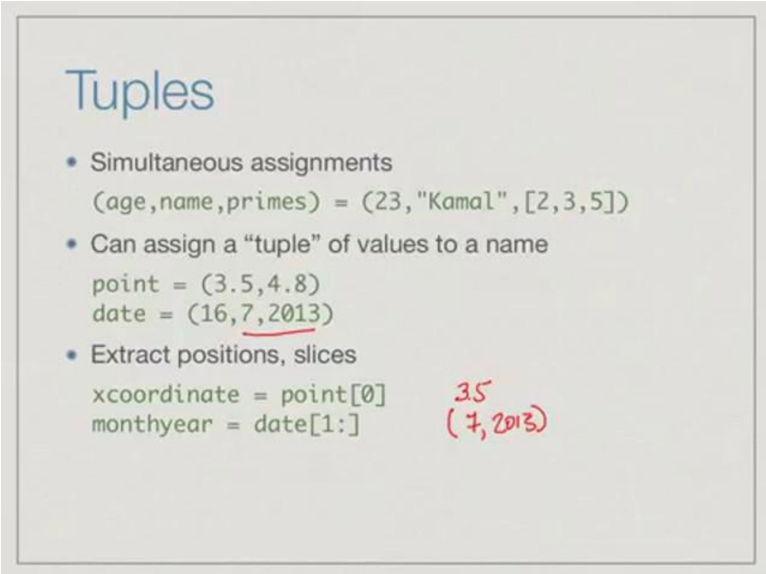
On the other hand, merge sort we can see is actually stable if you are careful to make sure that we always pick from one side consistently if the values are equal. So, when we are merging left and right when we have the equal to case we have to either put the element from left into the final list or 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)

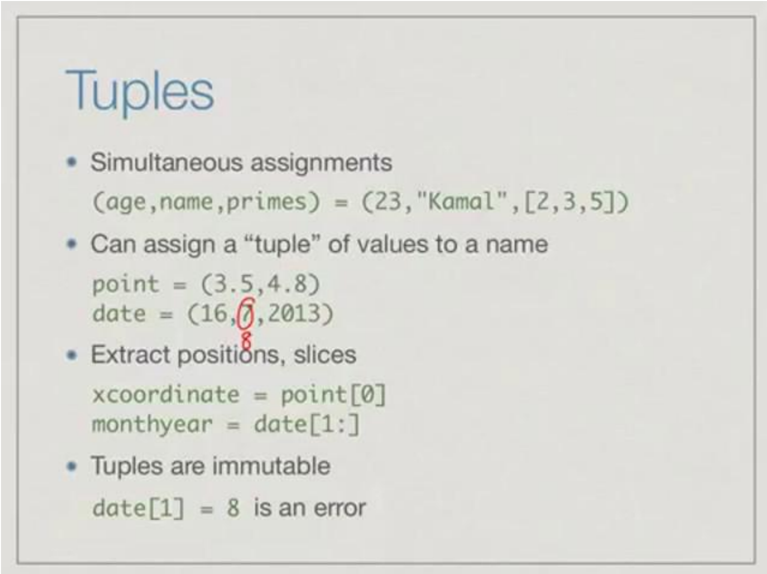



We have seen this kind of simultaneous assignment, where we take three names on the left and assign them to three values in the right, and we enclose these in these round brackets. So, this kind of a sequence of values with the round bracket is called a Tuple.

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.

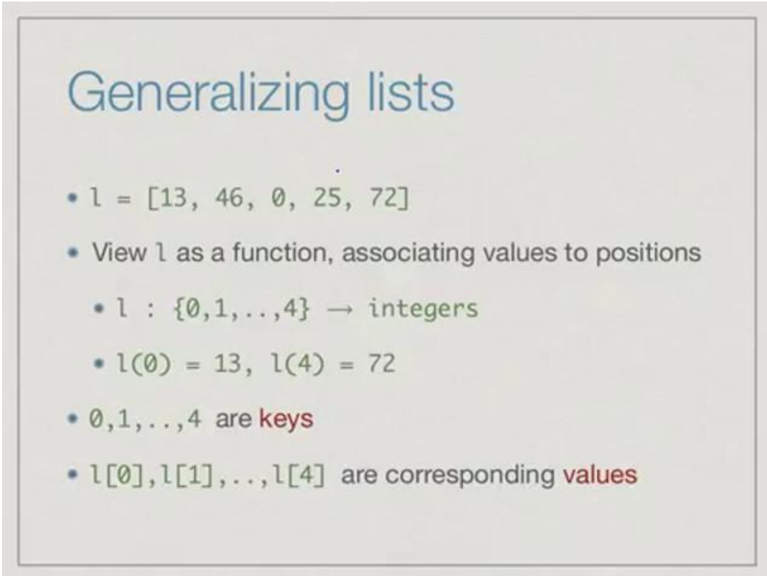
Similarly, we can say that a date is made up of three parts a day, a month, and a year; and we can encloses into a three value or triple. So, tuple behaves like a list, so it is a kind of sequence. So, like strings and list, in a tuple you can extract one element of a sequence. So, we can say that the 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)




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.

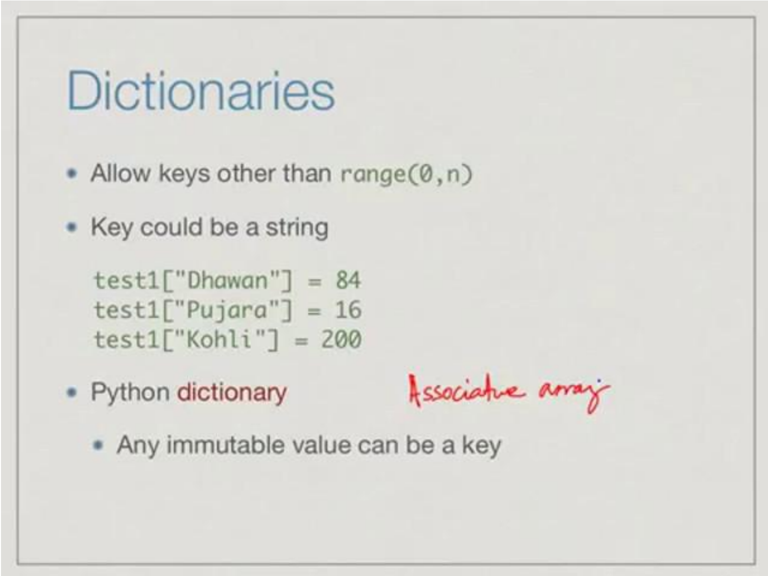
(Refer Slide Time: 02:10)




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.

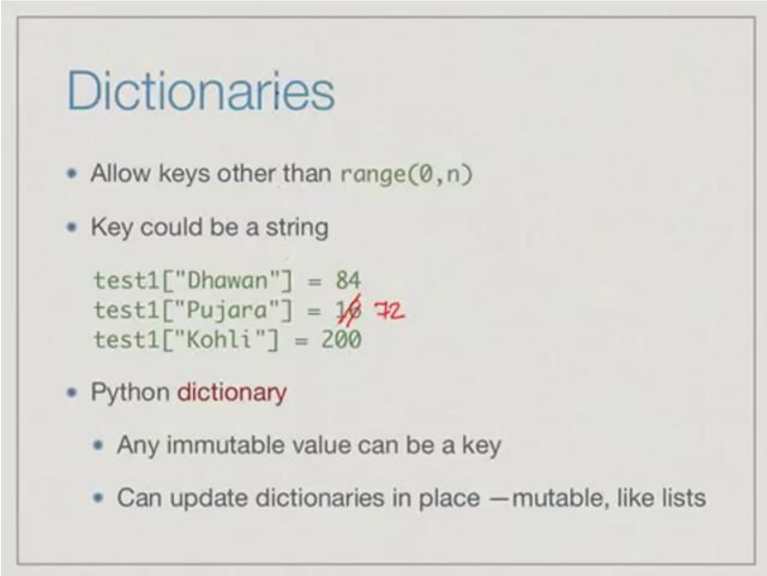
(Refer Slide Time: 03:19)




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.

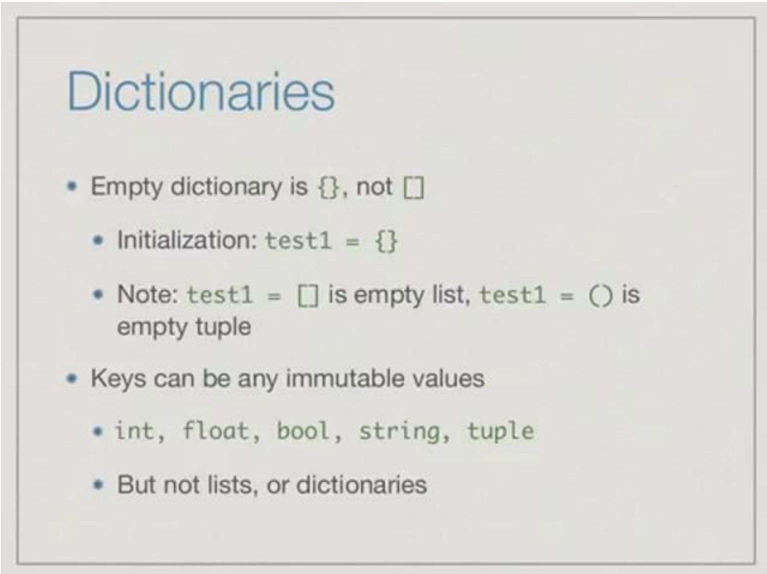
This is what python calls a dictionary, in some other programming languages this is also called an associative array. So, you might see this in the literature. So, here is a store of values which are accessed through a key which is not just a position, but some arbitrary index and python's rule is that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

(Refer Slide Time: 04:26)




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.

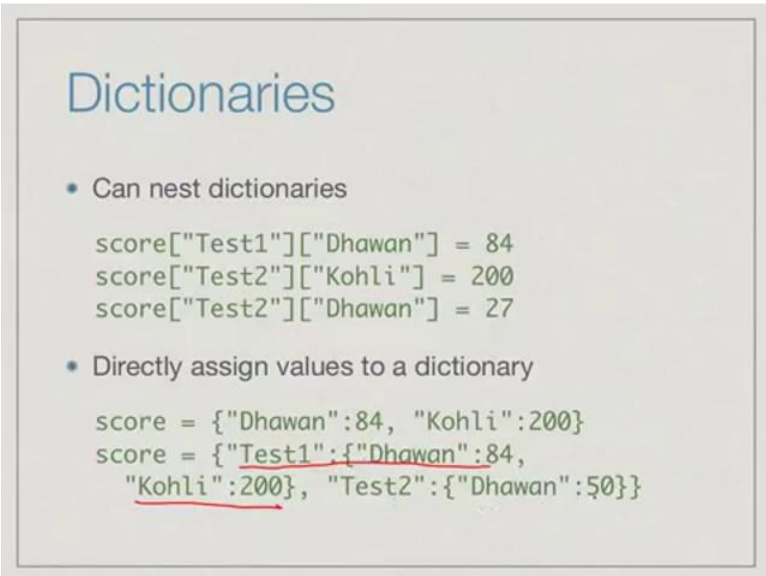
(Refer Slide Time: 04:59)



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)

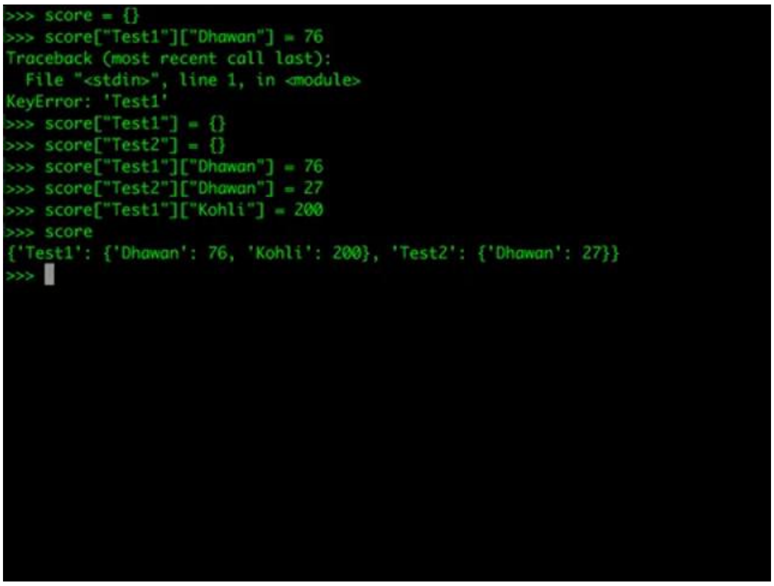



So, we can have multiple just like we have nested list where we can have a list containing list and then we have two indices take the 0th list and then their first position in the 0 list, we can have two levels of keys. If you want to keep track of scores across multiple test matches, instead of having 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.

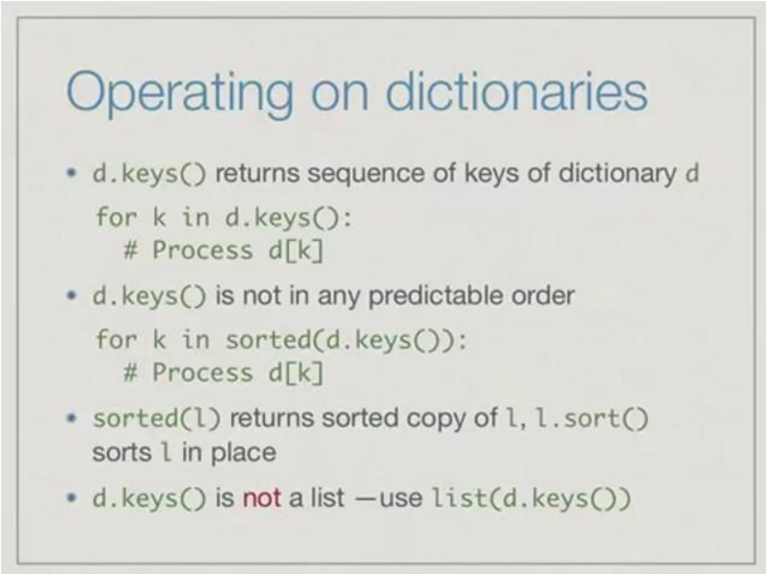
(Refer Slide Time: 07:39)




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 may be 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)




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 l takes an input list, leaves it unchanged, but it returns a sorted version.

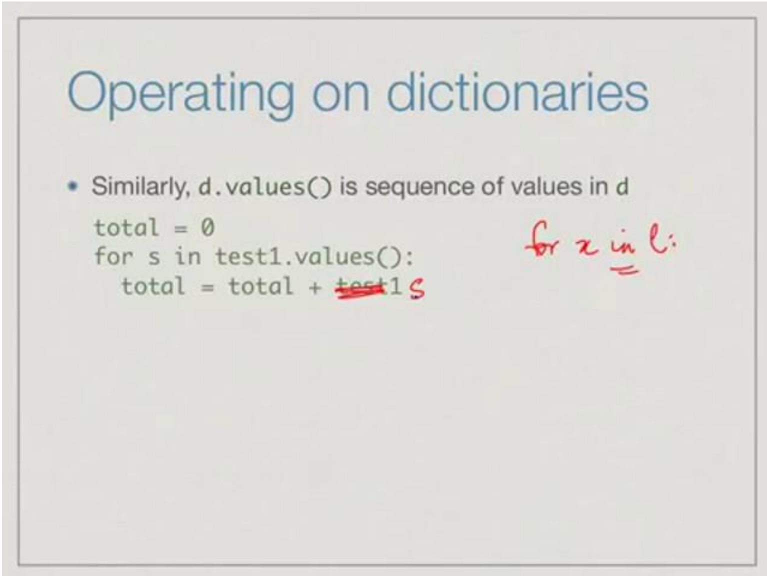
The other thing to keep in mind is that though it is tempting to believe that d dot keys is a list, it is not a list; it is like range and other things. It is just a sequence of values that you can use inside of for, so we must use the list property to actually create a list out of d dot keys.

(Refer Slide Time: 10:46)


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.

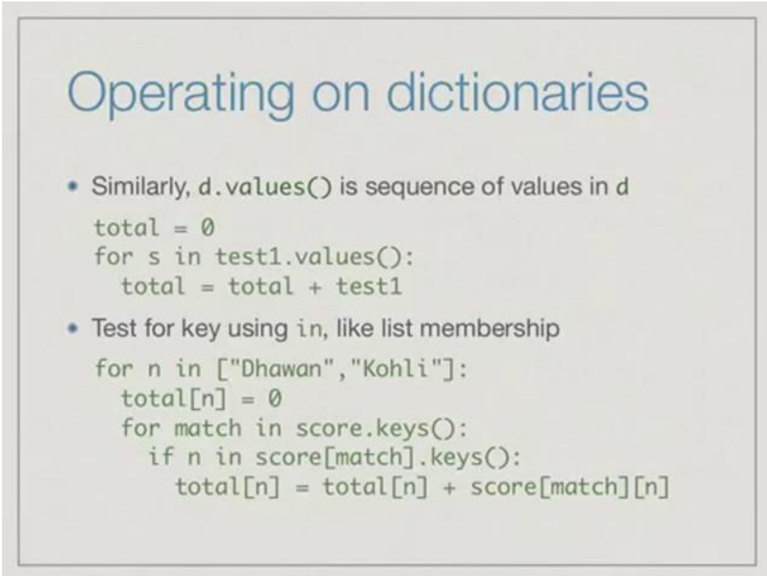
Now notice that the keys are inserted in the order a, b, c, d, e, f, g, h, i but if I ask for d dot keys it produces it in some very random order. So, e is first and a is way down and so on. There is no specific order that you can get from this. So, this is just to emphasize that the order in which keys 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)




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 that 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.

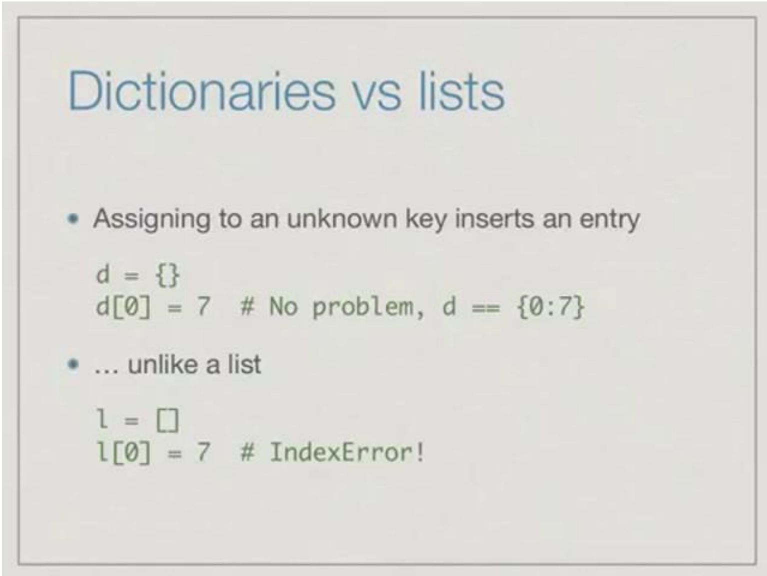
(Refer Slide Time: 12:50)




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.

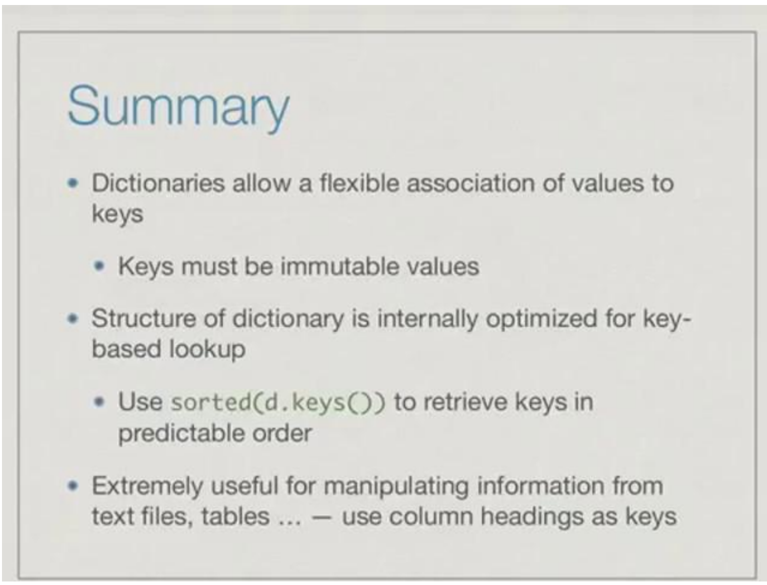
Now for each match in our nested dictionary, if Dhawan is entered as a batsman in that match, so if a name Dhawan appears as the key in score for that match then and only then you add a score, because if it does not appear it is illegal to access that match. So, this is one way to make sure that when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

(Refer Slide Time: 14:00)




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)



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 list 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 in to your programming skills, because this is what makes python really a very powerful language for writing scripts to manipulate it.

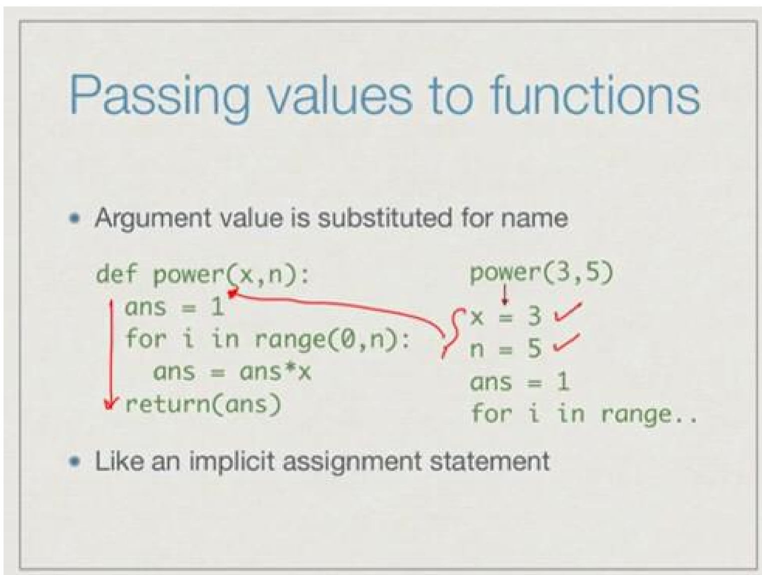
**Programming Data Structures and Algorithms in Python**

**Prof. Madhavan Mukund**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Madras**

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

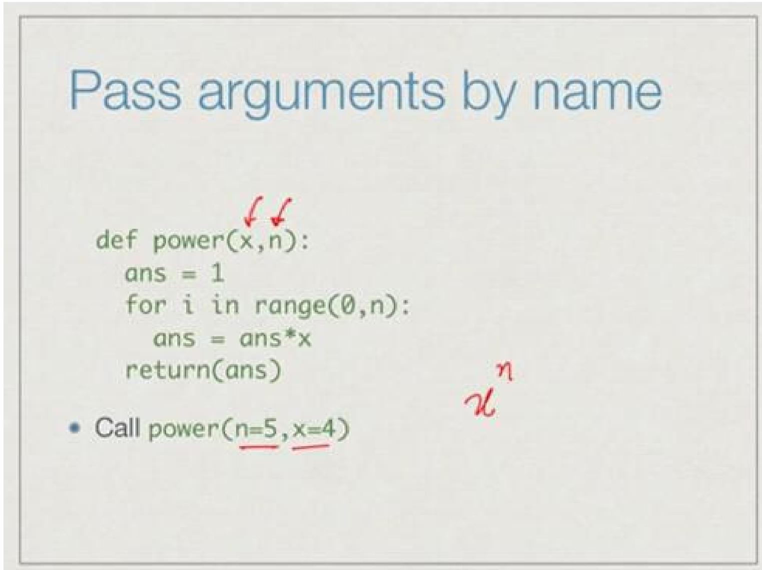
We have seen that we pass values to functions by substituting values for the argument set when defining the function.

(Refer Slide Time: 00:02)




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.

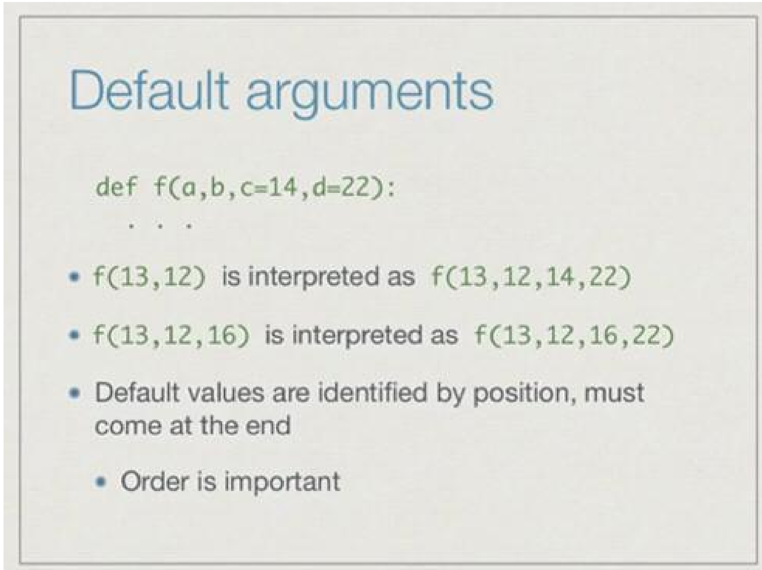
(Refer Slide Time: 00:02)




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)




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.

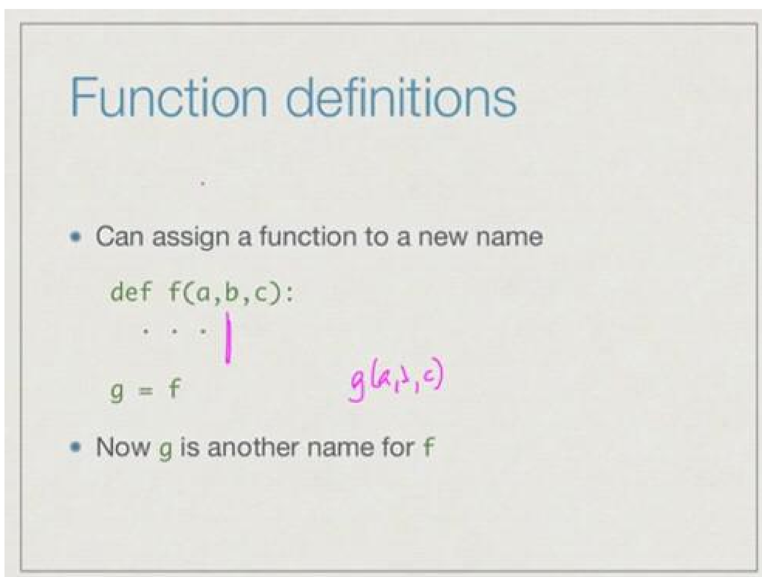
But, the problem is that, this quantity, the length of A, depends on A itself. So, when the function is defined, there will be, or may not be a value for A and whatever value you have chosen for A, if there is one, that length will be taken as a default. It will not be dynamically computed each time 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 pick 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)

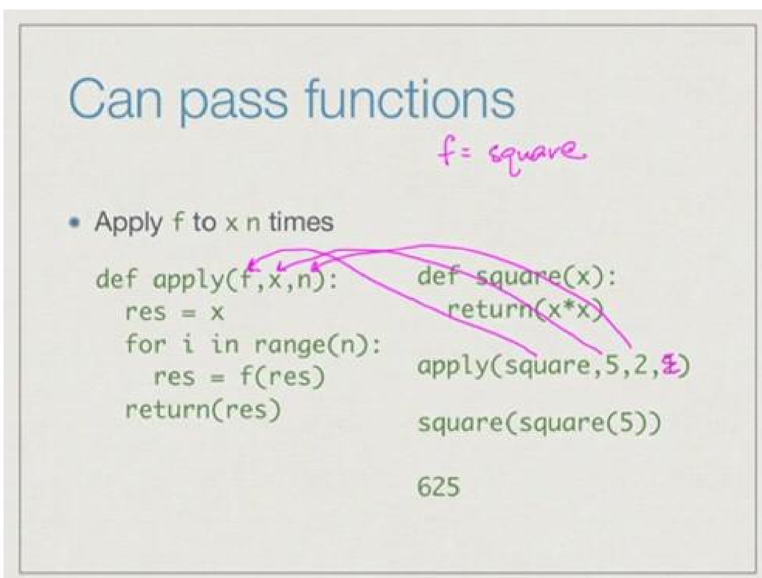



A function definition associates a function body with a name. It says, the name f will be interpreted as a function which takes some arguments and does something. In many ways, python interprets this like any other assignment of a value to a name. For instance, this value could be defined in different 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.

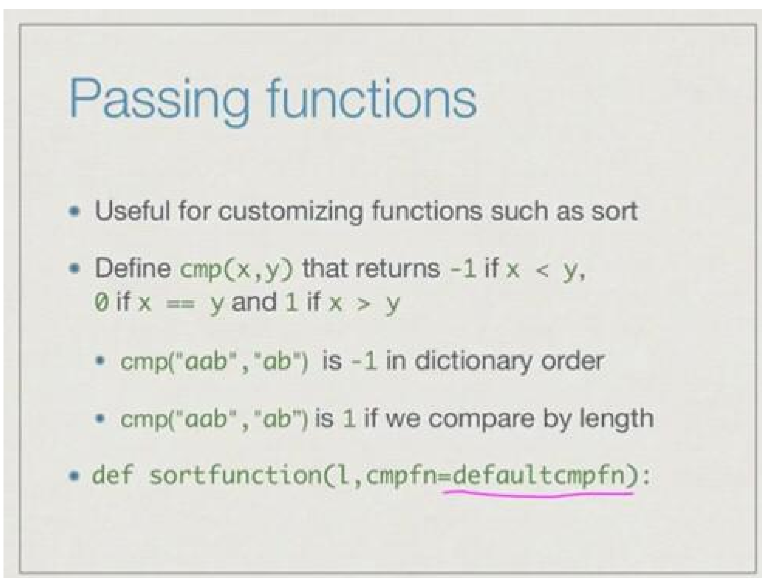
(Refer Slide Time: 08:40)



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)

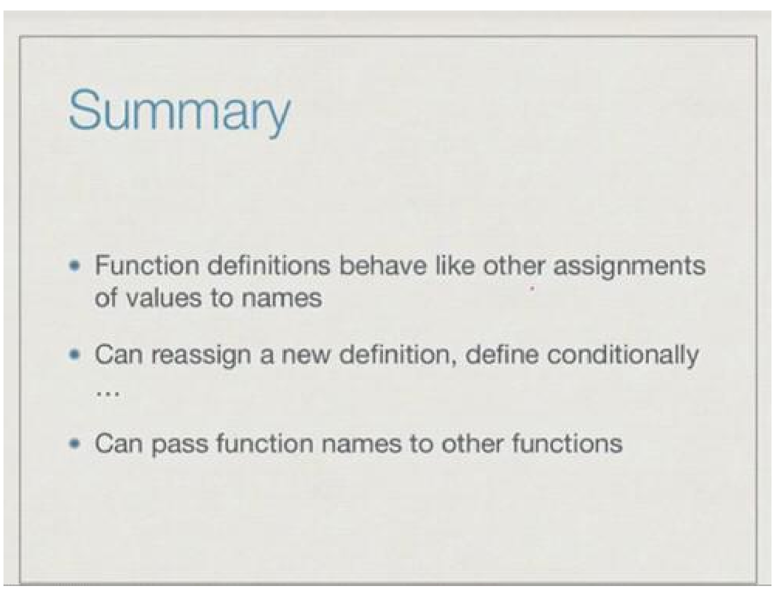



One practical use of this is to customize functions such as sort. Sometimes, we need to sort values based on different criteria. So, we might have an abstract compare function, which returns minus 1 if the first argument is smaller, zero if the 2 arguments are equal, and plus 1 if the first argument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



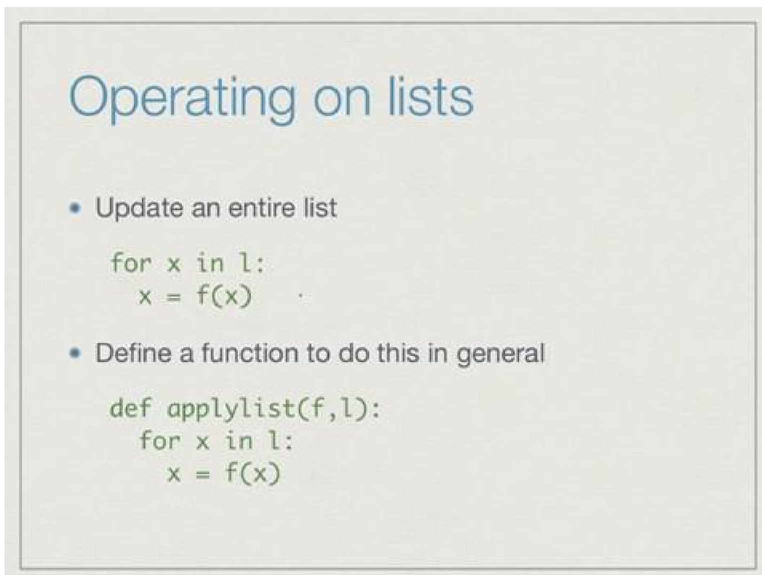
To summarize, function definitions behave just like other assignments of values to names. You can reassign a new definition to a function. You can define it conditionally and so on. Crucially, you can use one function and make it point, name point to another function, and this is implicitly used when 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**

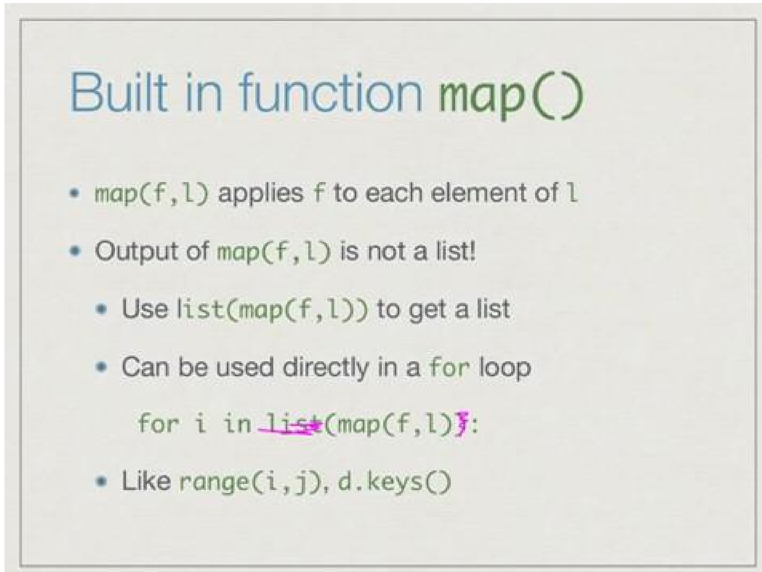
Quite often, we want to do something to an entire list.

(Refer Slide Time: 00:02)




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 apply list, 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.

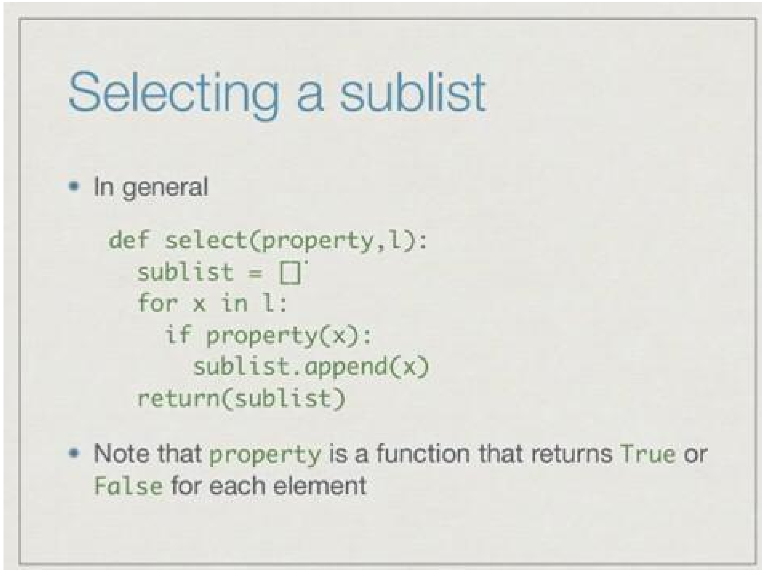
(Refer Slide Time: 00:45)




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.

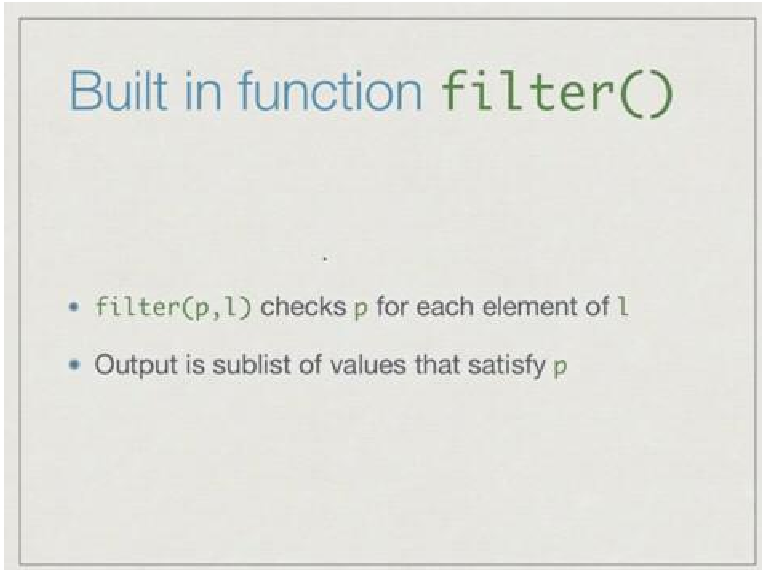
(Refer Slide Time: 02:10)




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.

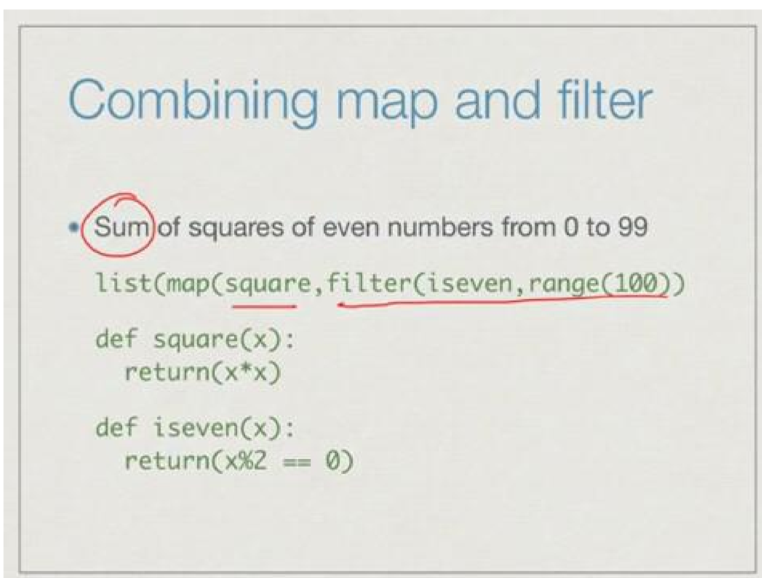
And then, we apply the test and some of them will pass, and some of them will succeed, some of them will fail, and at the end, wherever the things pass, those items will emerge in the output. So, in general, we could write a select function which takes the property and a list, and it creates a sub list 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 l 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)




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.

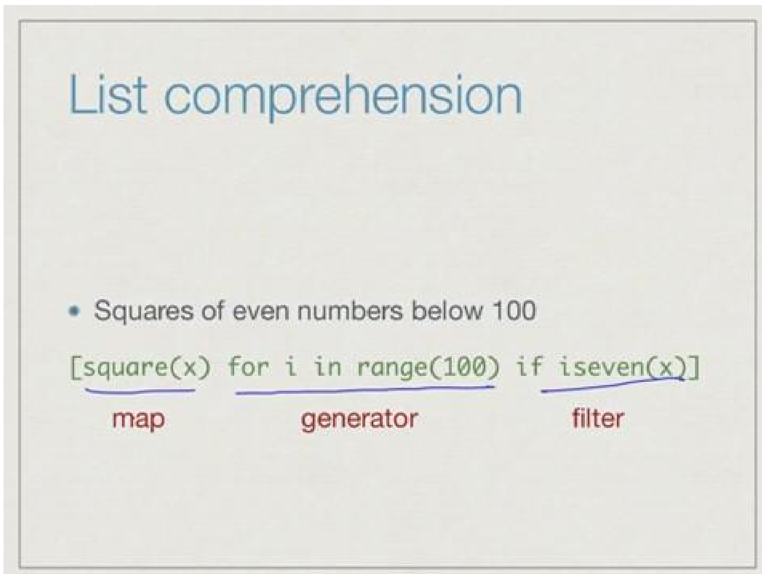
(Refer Slide Time: 04:00)




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)




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.

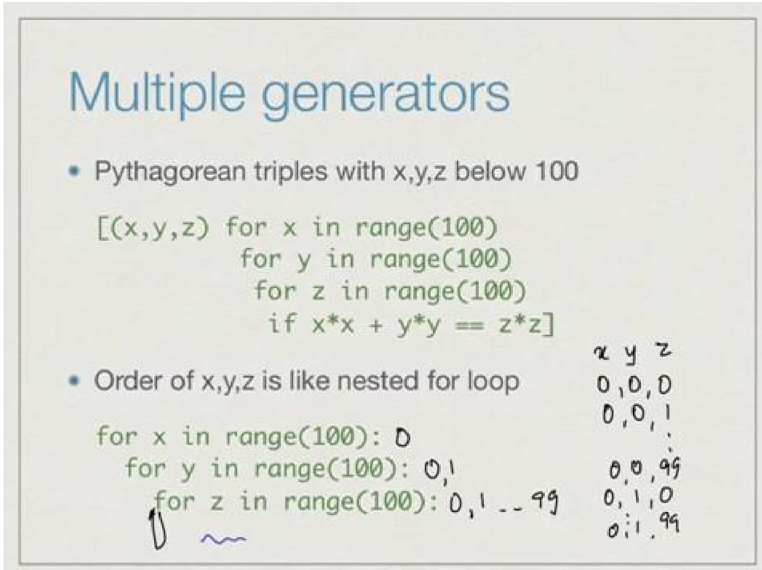
In set theory, this is called set comprehension. This is the way of building a new set by applying some conditional things to an old set. This is also implicitly applying a kind of a tripling operator; it takes 3 separate sets, x from 1 to n, y from 1 to n, z from 1 to n, combines them into triples. 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)

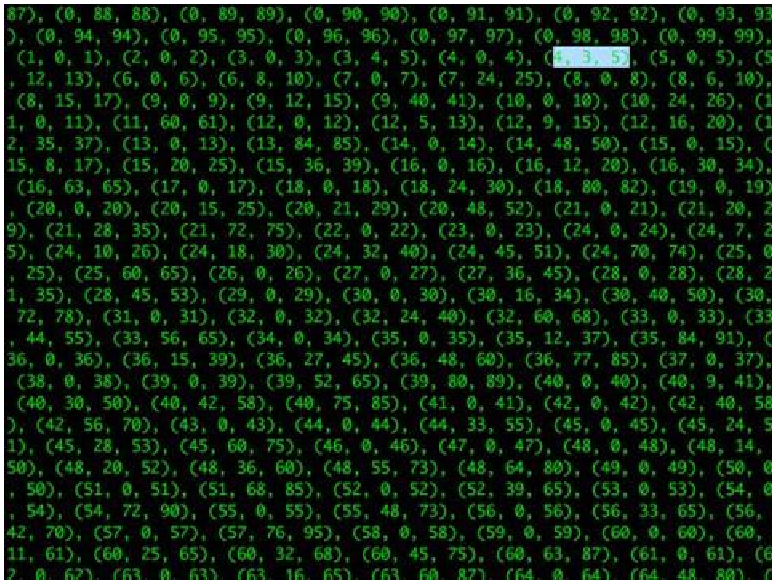



Let us go back to the Pythagorean triple example. We want all Pythagorean triples with x, y, z below 100. This, as we said, requires us to cycle through all values of x, y, and z in that range. it is a little bit more complicated than the one we did before, where we only had a single generator, all the 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)

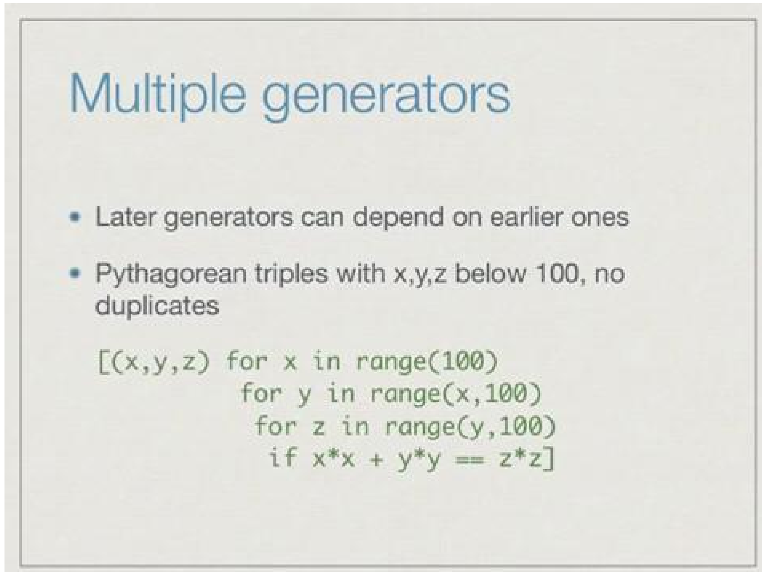



Let us see how this works in python. Let us first begin by defining square; a square of x return x times x; then, we can define iseven x, to check that the remainder of x divided by 2 is 0. So, we have square 8, 64; iseven 67 should be false; iseven 68 should be true and so on. Now, we have list comprehension. 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)

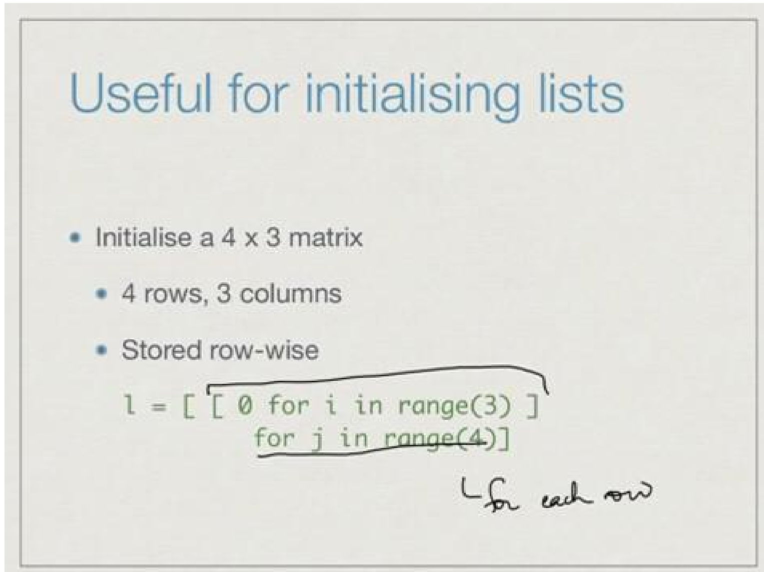



So, we can have a situation, just like we have in a 'for loop', where the later loop can depend on an earlier loop; if the outer loop says, i to some, i goes from something to something, the later loop can say that, j starts from i, and goes forward. For instance, we can now rewrite our Pythagorean triples 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)


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.

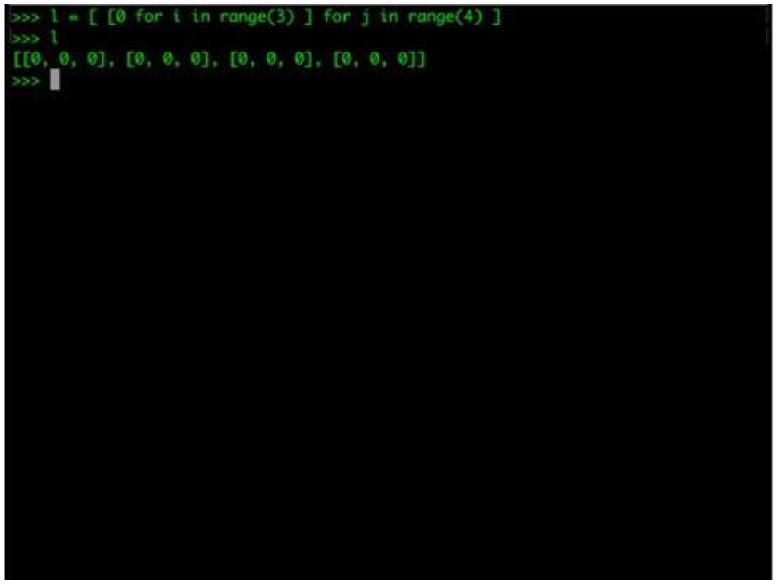
(Refer Slide Time: 14:06)




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.

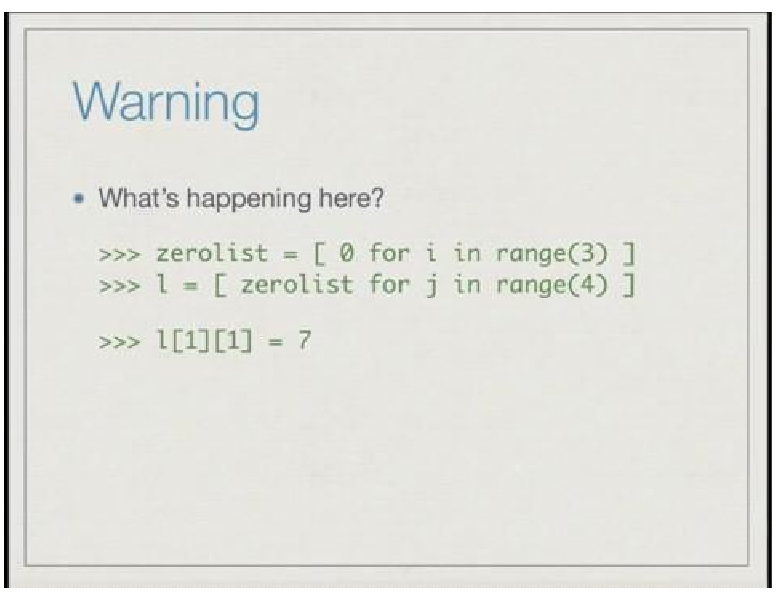
Here is an initialization, which says, l consists of something for the outer things says for, this is for each row. It is something for each row. For 4 rows 0, 1, 2, 3, I do something; and what is that something? I create a list of zeros of that size 3. Each row j, from 0 to 3, consists of columns 0, 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)



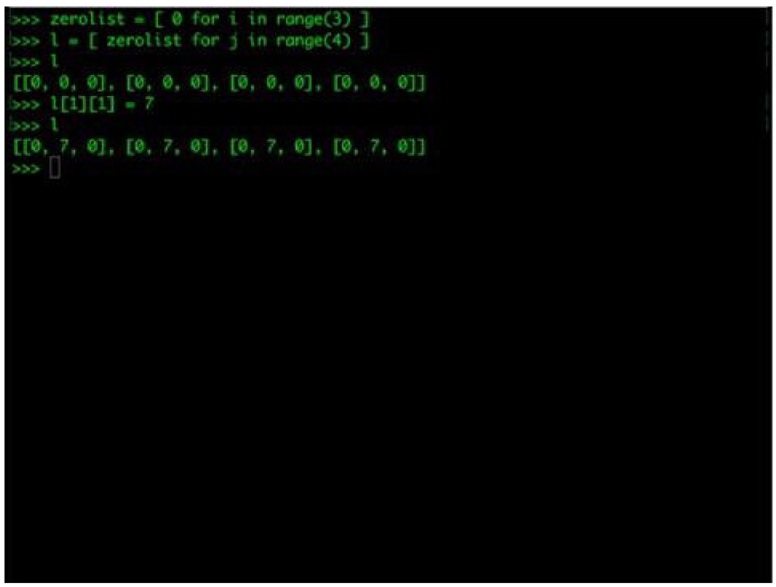

Here is that list comprehension notation for initializing the matrix. So, it says, for every j in range 4, right, we create a list, and that list itself has zero for i in range 3, and if you do this, and look at it, then, correctly it has 3 zeros, and 3 zeros, and 3 zeros, 4 times. These are the four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

(Refer Slide Time: 15:37)



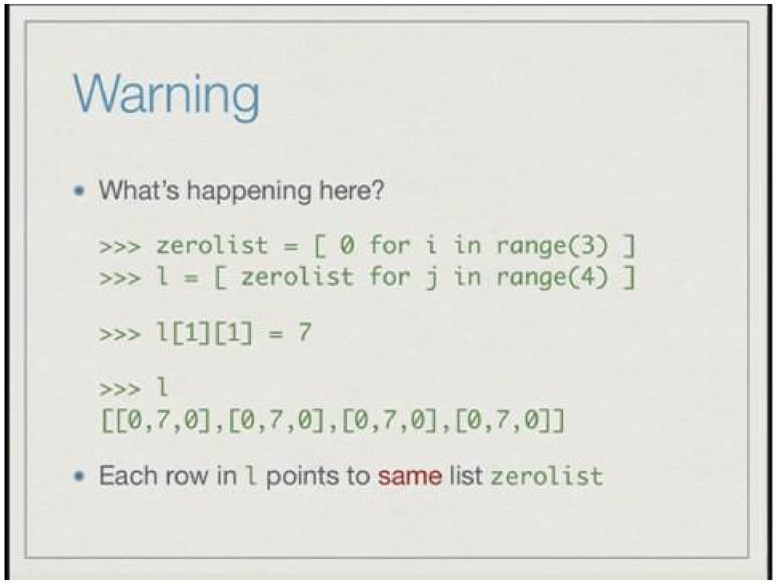
Suppose, instead, we split this initialization into 2 steps; we first create a list of 3 zeroes called zerolist, which says zero for i in range 3. This creates a list of 3 zeros; and then, we copy this list 4 times, in the four rows. We say that the actual matrix l has 4 copies of zerolist. Now, we go 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)




There we have the zero lists, and then, we say, l is 4 copies of zerolists, 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.

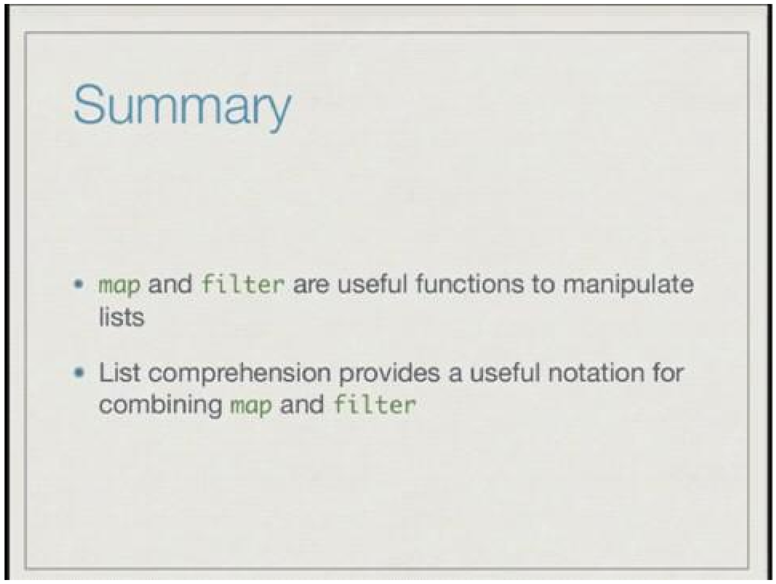
(Refer Slide Time: 16:56)




The output after l 1 1 is equal to 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 un-intentionally combine 2 rows into copies of the same thing, and updates to one row will also update another row.

(Refer Slide Time: 17:45)

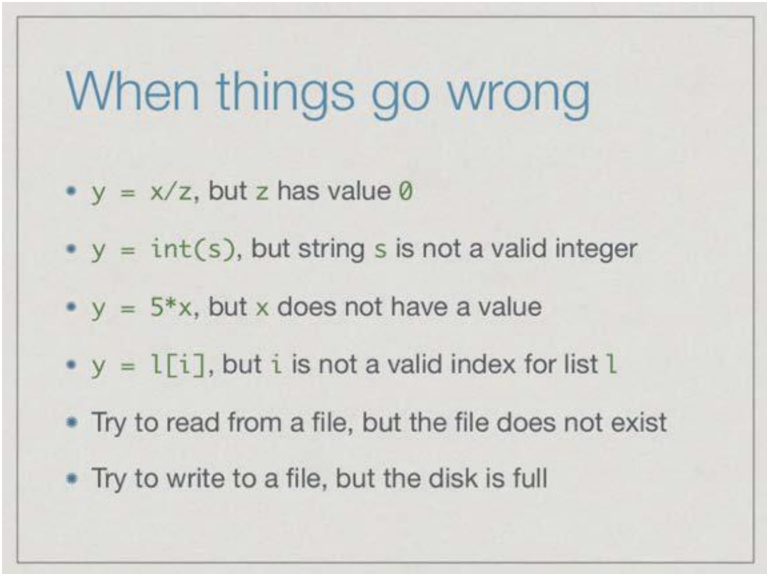



To summarize, map and filter are very useful functions to manipulating lists, and python provides, like many other programming languages, based on the function programming, the notation called list comprehension, to combine map and filter. And, one of the uses that we saw for list comprehension is to 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**

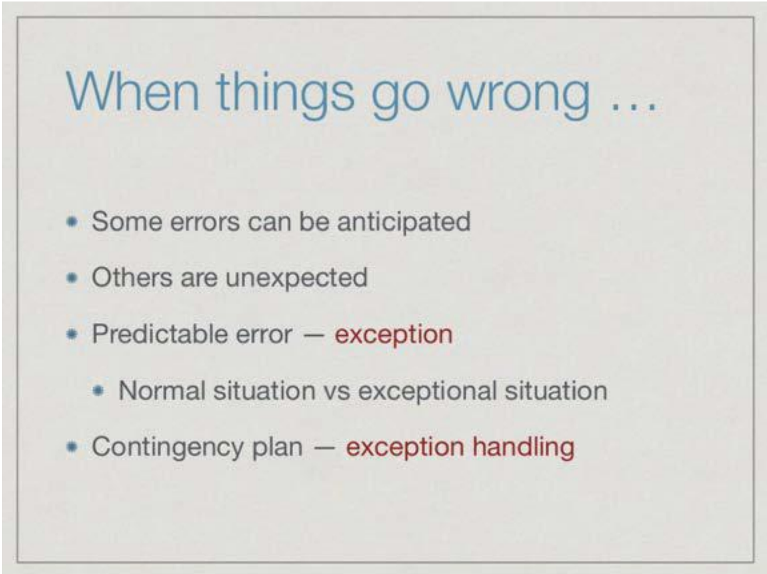
(Refer Slide Time: 00:03)




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.

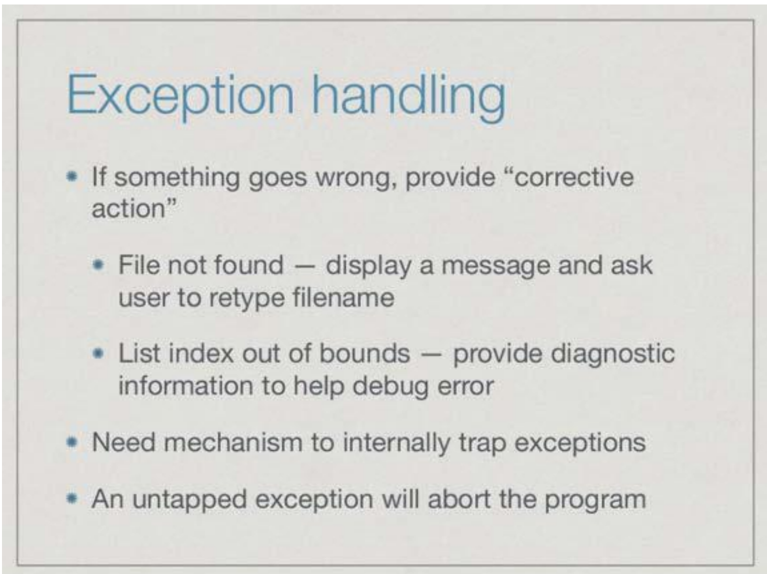
We could also be trying to compute an expression, using a name whose value has not been defined, or we could try to index a position in a list which does not exist. As we go forward we will be looking at how to read and write from files on the disc. So, we may be trying to read from a file, but perhaps 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)




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.

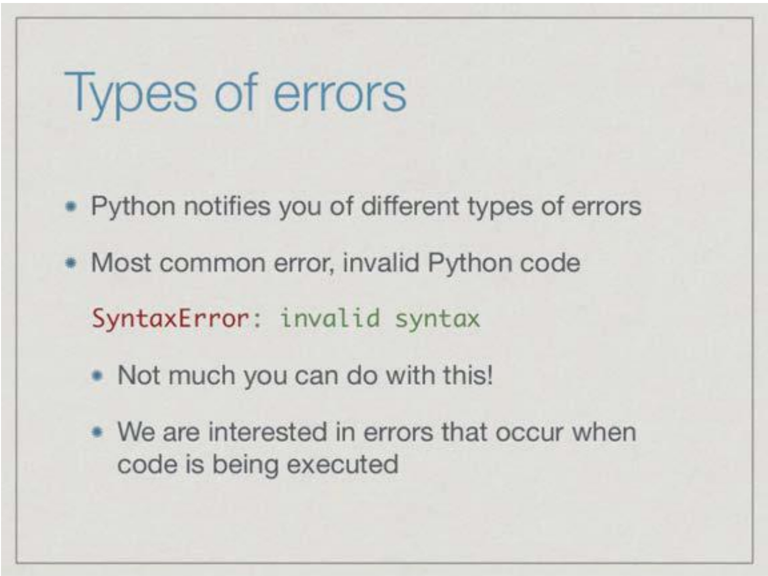
(Refer Slide Time: 01:36)



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.

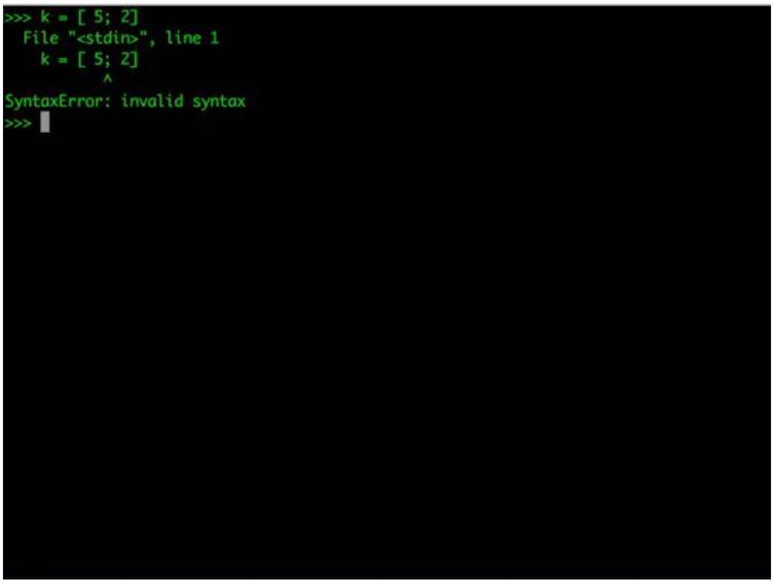
On the other hand if a list is being indexed out of bounds there is probably an error in our program, and we might want to print out this value the value of the index to try and diagnose what is going wrong with our program. Sometimes the error handling might just be debugging our error prone program. 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)




Now, there are many different types of errors and some of these we have seen, but we may not have noticed the subtelty 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.

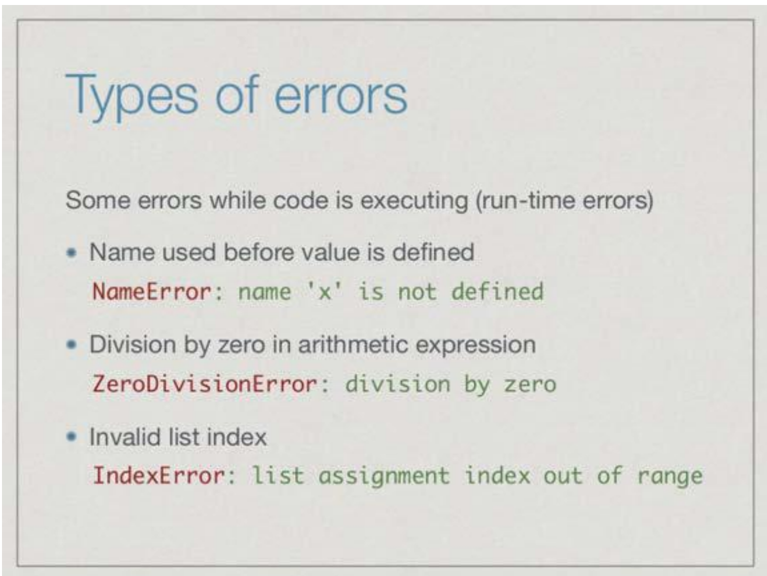
(Refer Slide Time: 03:19)




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.

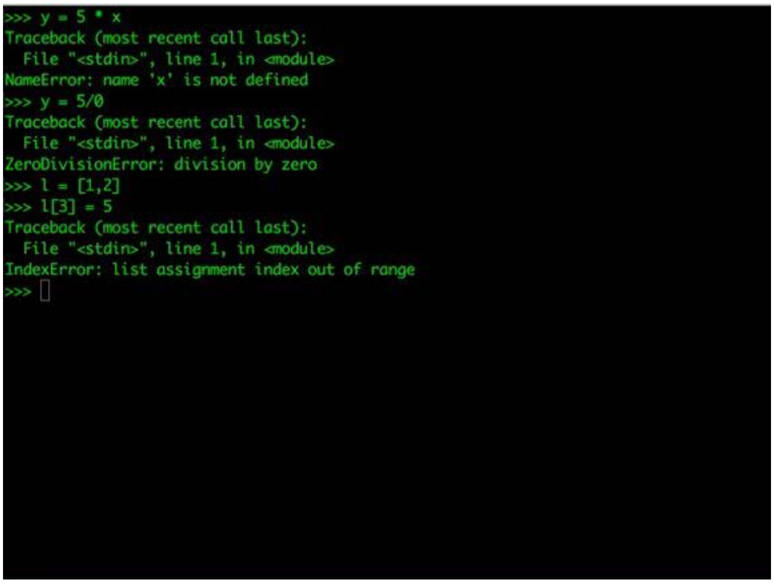
(Refer Slide Time: 04:01)



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.

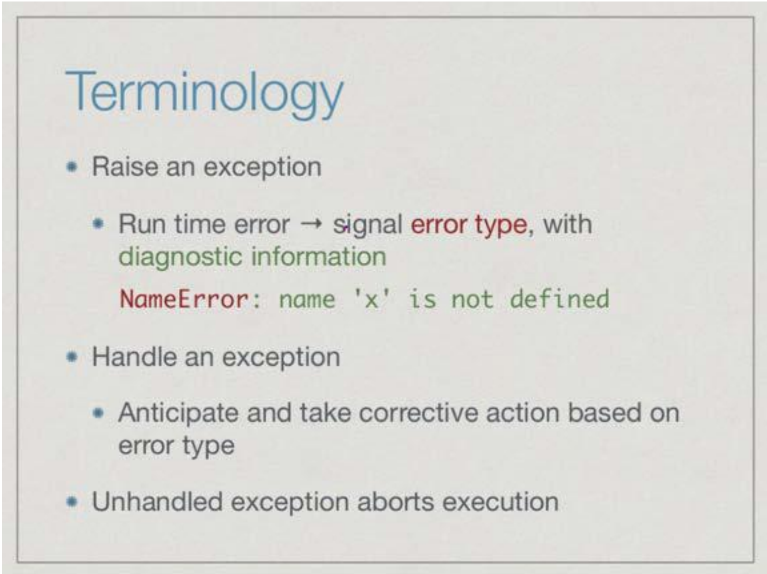
(Refer Slide Time: 04:54)




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)

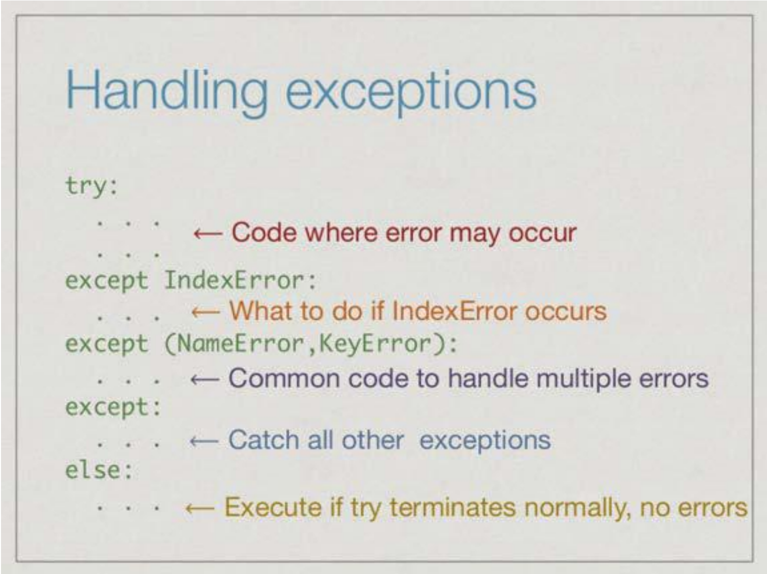



Let us first quickly settle on some terminology. So, usually the act of signalling an error is called raising an exception. So, when the python interpreter detects an error it gives us information about this error and as we saw it comes in two parts, there is the type of the error give what kind of error 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)




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.

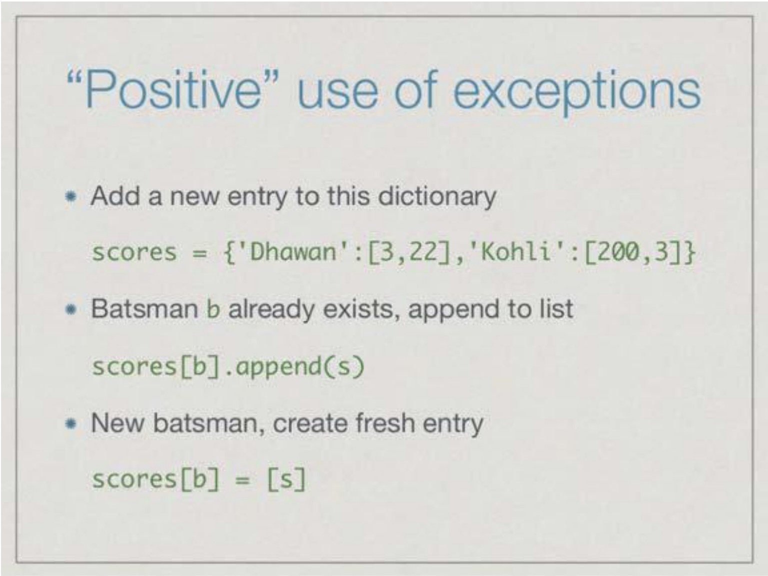
So, kind of a naked except block in which you do not specify the type of error and by default such an except statement would catch all other exceptions. The important thing to remember is that this happens in sequence. If I have three errors for example, if I have an index error and a name error and 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 it will 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.

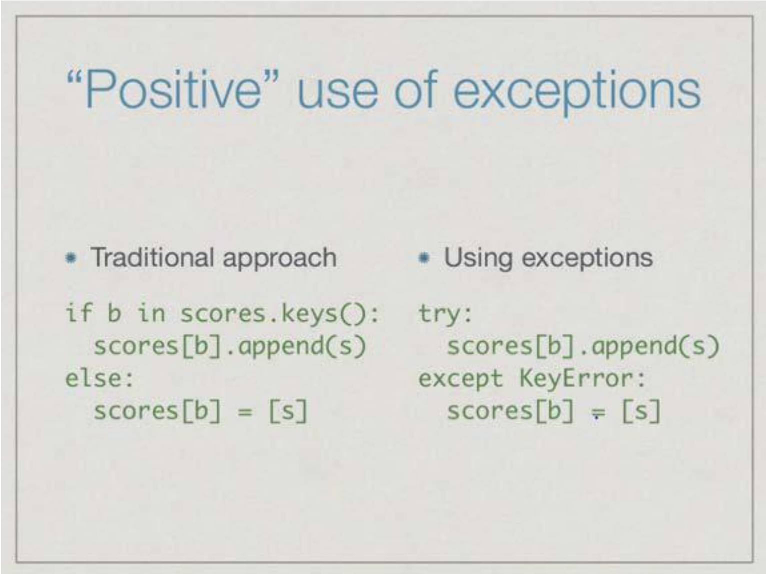
(Refer Slide Time: 10:58)




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 of 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.

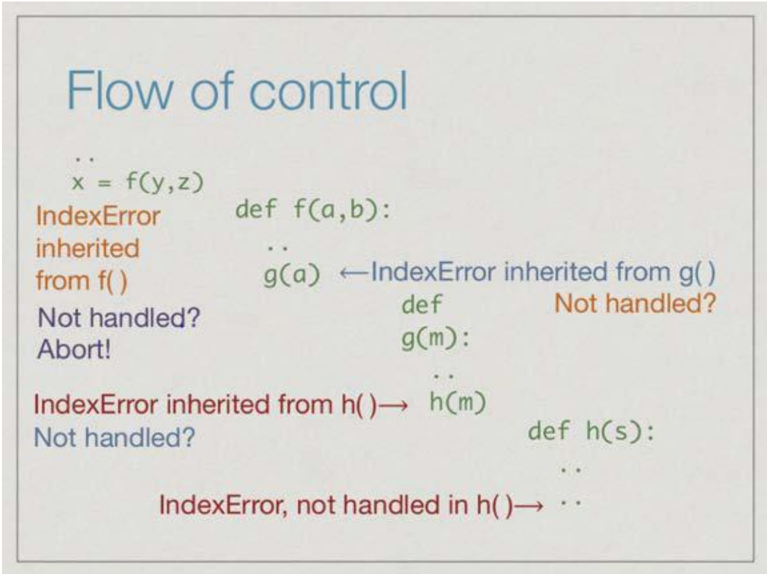
(Refer Slide Time: 12:21)




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 dot 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 of b then create an entry scores of b is equal to 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.

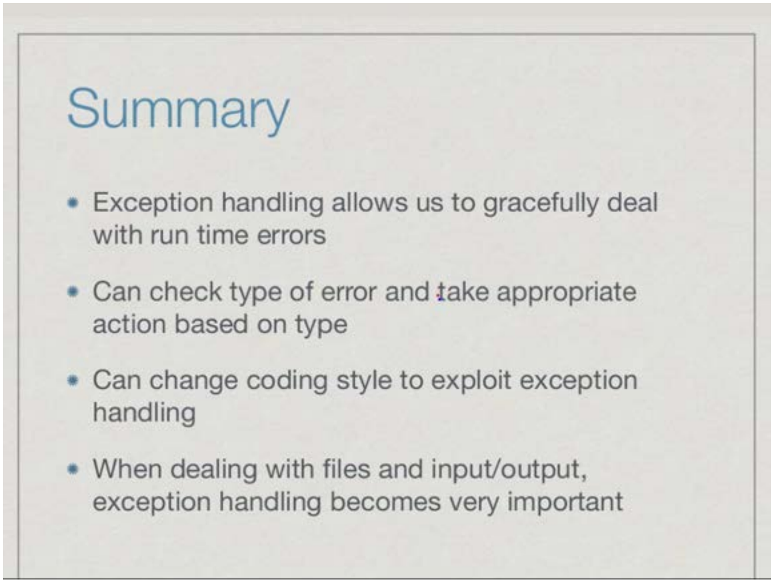
Somewhere inside h perhaps there is an index error and where we used this list for example, in h we did not put it on a try block and so, the error is not handled. So, what happens we said is when an error is not handled the program aborts, but the program does not directly abort; this function will 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)




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.

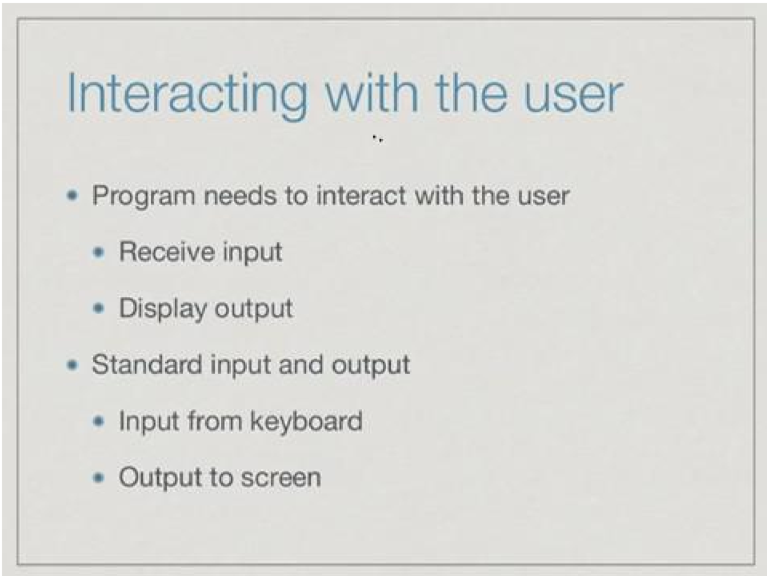
Input and output inherently involves a lot of interaction with outside things outside the program and hence is much more prone to errors and therefore, is useful to be able have this mechanism within our bag of tricks.

**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**

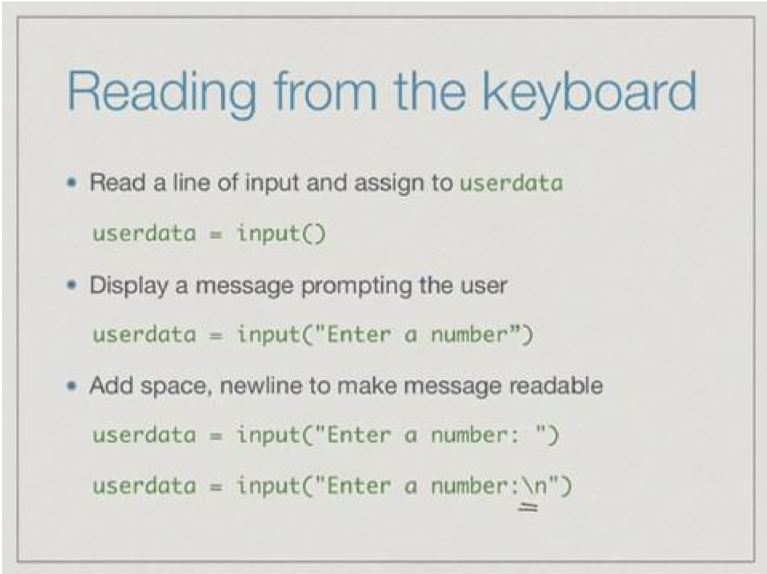
Till now, all the programs that you have been asked to write in your assignments have actually been just functions.

(Refer Slide Time: 00:02)




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)

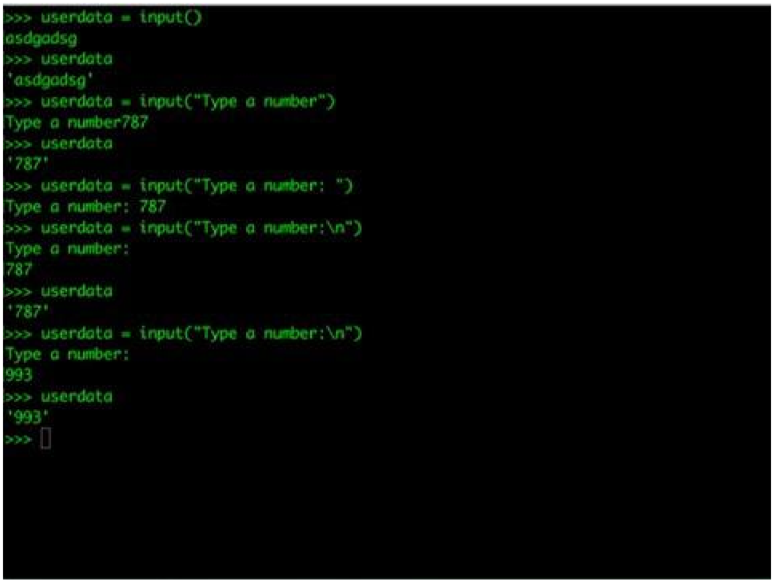



Traditionally, these modes are called standard input and standard output. So, standard input just means, take the input from the keyboard or any standard input device like that and standard output just means display the output directly on the screen. The basic command in python to read from the keyboard 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.

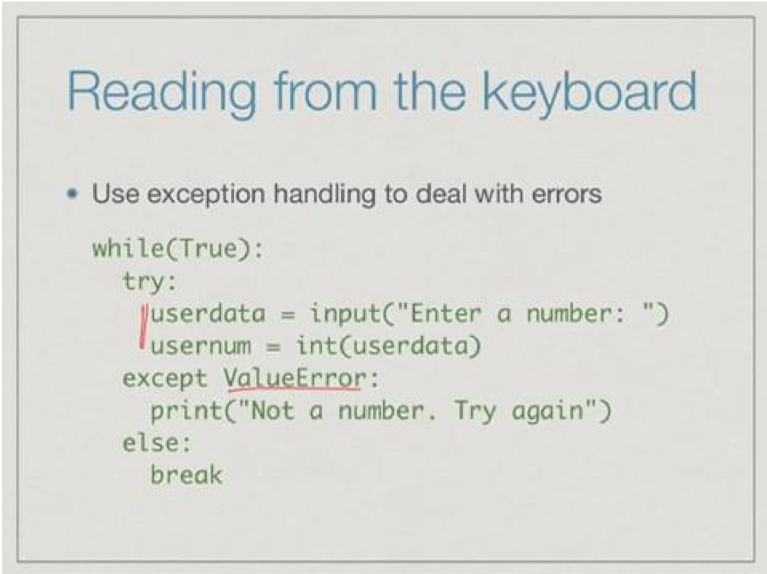
(Refer Slide Time: 02:30)




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)



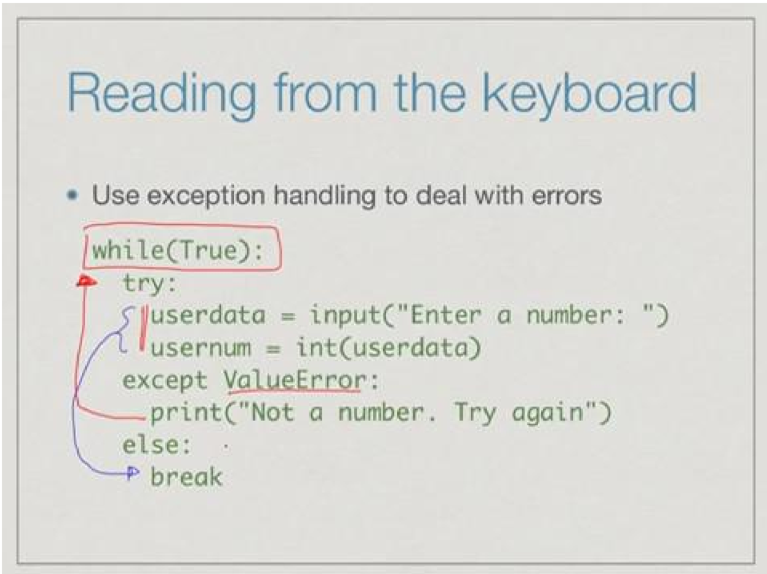

As we saw, when we were playing with the interpreter, the input that is read by the function input is always a string. Even if you say, enter a number and the user types in a number, this is not actually a number. If you want to use it as a number, you have to use this type conversion. Remember, we have 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)


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)



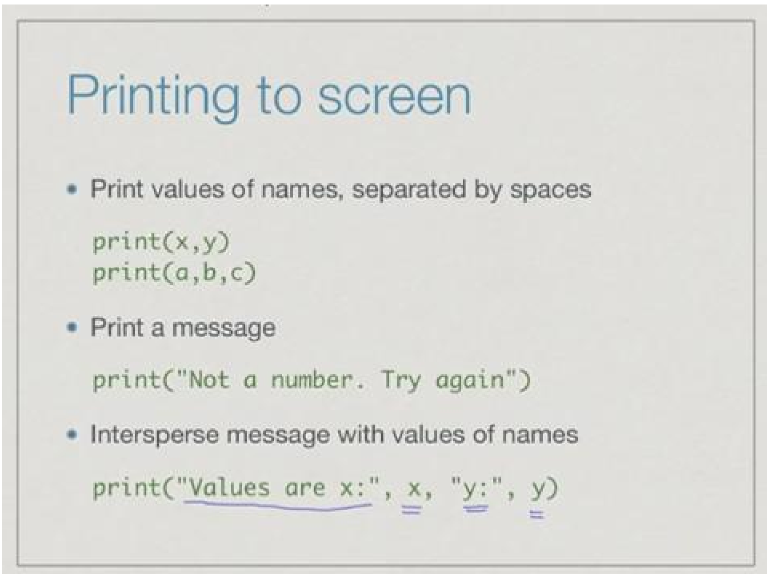

So, what we are trying to say is that, if we get a value error, we want to take appropriate action. So, we have this try block and if we see a value error, what we do is, we print a message. Now, we are going to see print just after this, but we print a message to the user, saying this is not a number, 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)

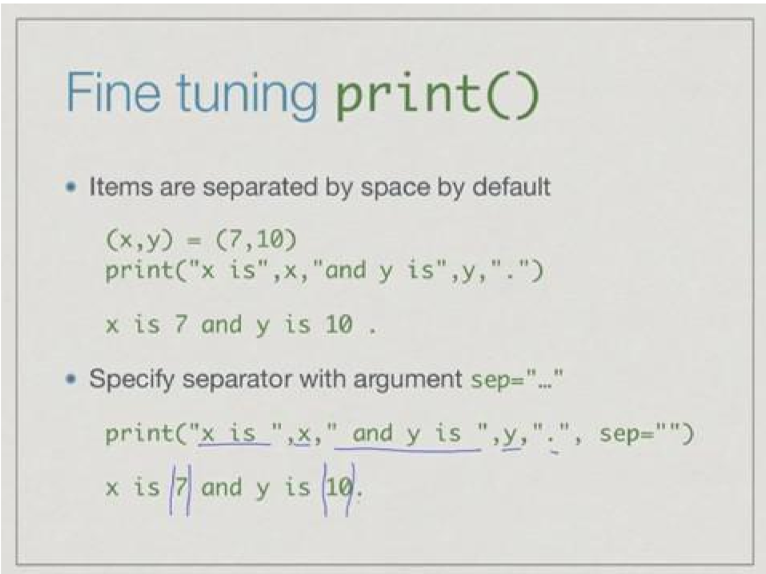



And, this is achieved using the print statement, which we have seen occasionally, informally, without formally having defined it. The basic form of the print statement is to give a sequence of values, separated by commas. So, print x, y will display the value of x, then, a space, then, the value of y.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




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.

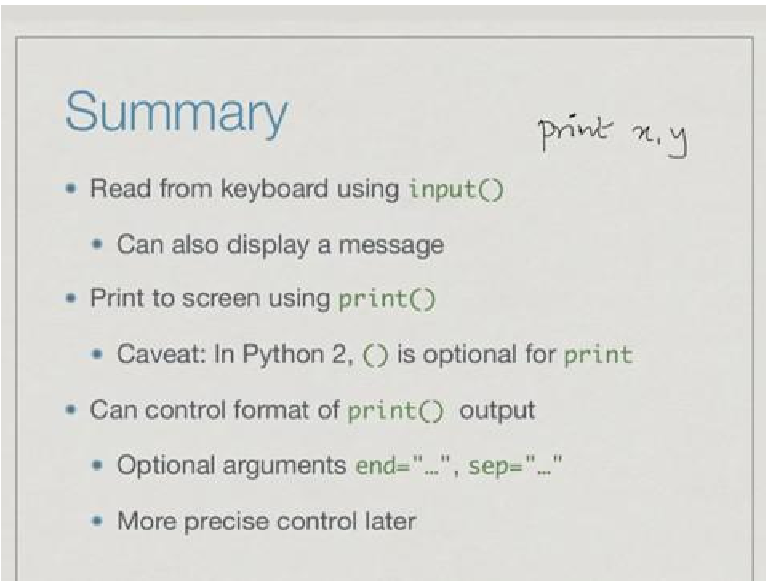
(Refer Slide Time: 13:28)

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)



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.

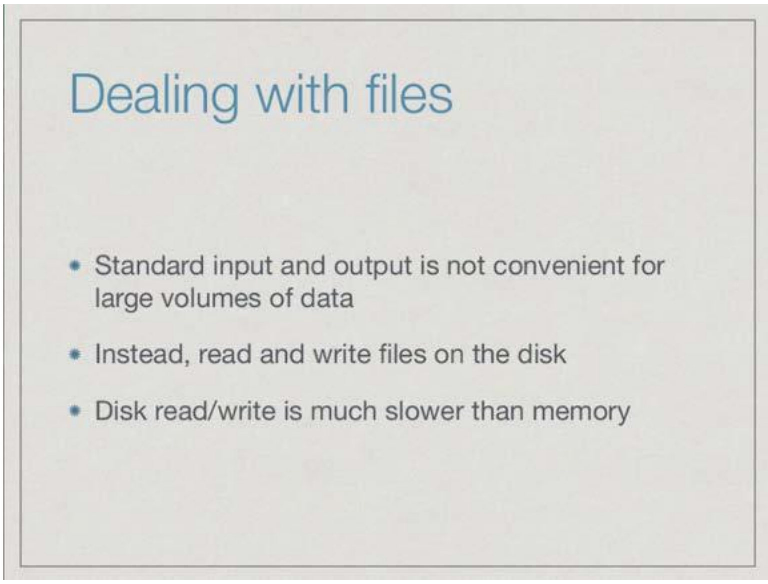
In python 2, you can say print space and then, give the, what is given as the arguments to print in python 3. Python 3 insists on it being called like a function, with brackets; in python 2 the brackets are optional. So, you will very often see, in python 2 code, something that looks like print x, y, 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**

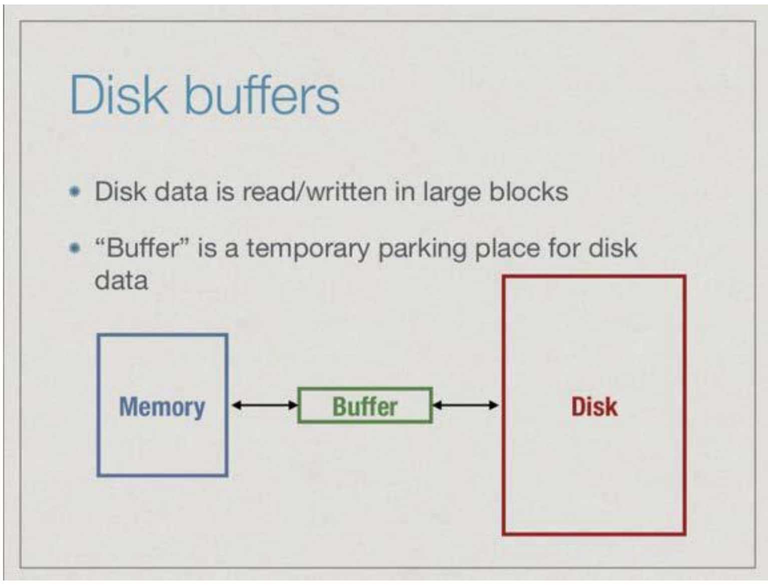
(Refer Slide Time: 00:02)




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.

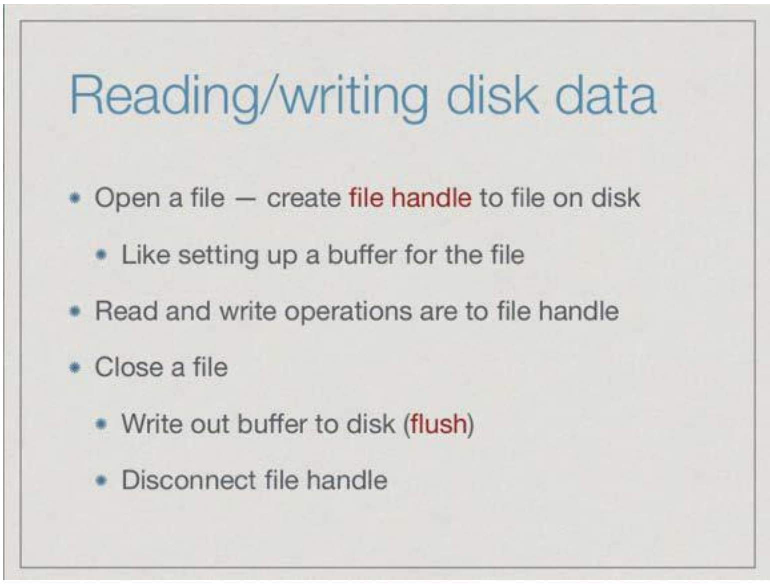
(Refer Slide Time: 00:48)




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)

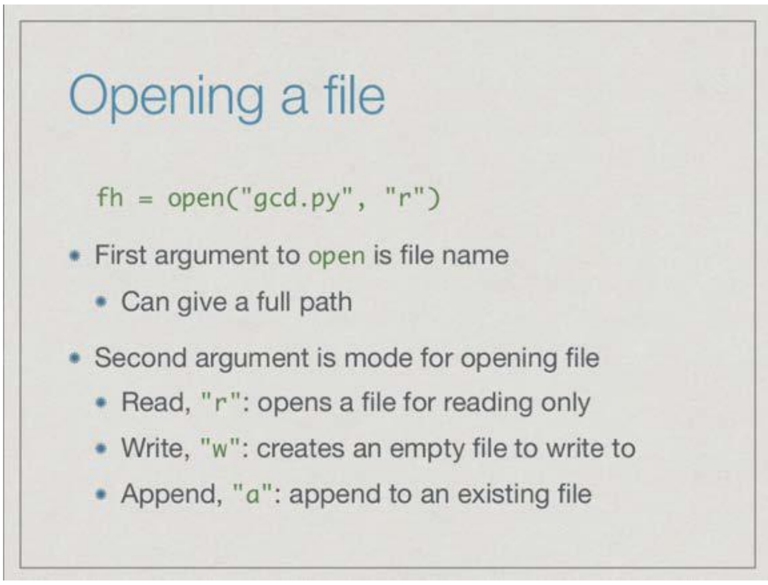



When we read and write from a disk the first thing we need to do is connect to this buffer. This is called opening a file. So, when we open a file we create something called a file handle and you can imagine that this is like getting access to a buffer from which data from that file can read into memory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

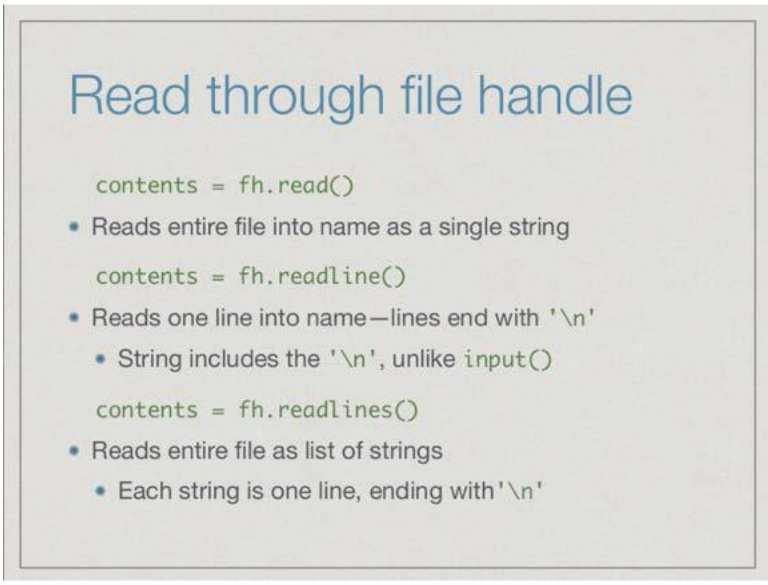



The command to open a file is just ‘open’. The first argument that you give open is the actual file name on your disk. Now, this will depend a little bit on what system you are using, but usually it has a first part and an extension. This commands, for instance, to open the file gcd dot py. Now implicitly, 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 exits 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)




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.

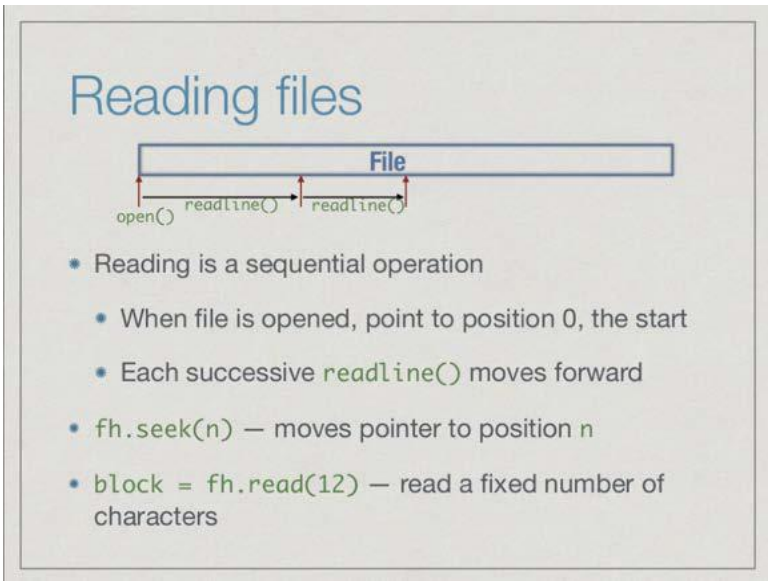
So, readline is like that, but the difference between the readline and input is that, when you read a line you get the last new line character along with the input string. When you say input you only get the characters which come before the last new line the new line is not included, but in readline 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 srings. 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)



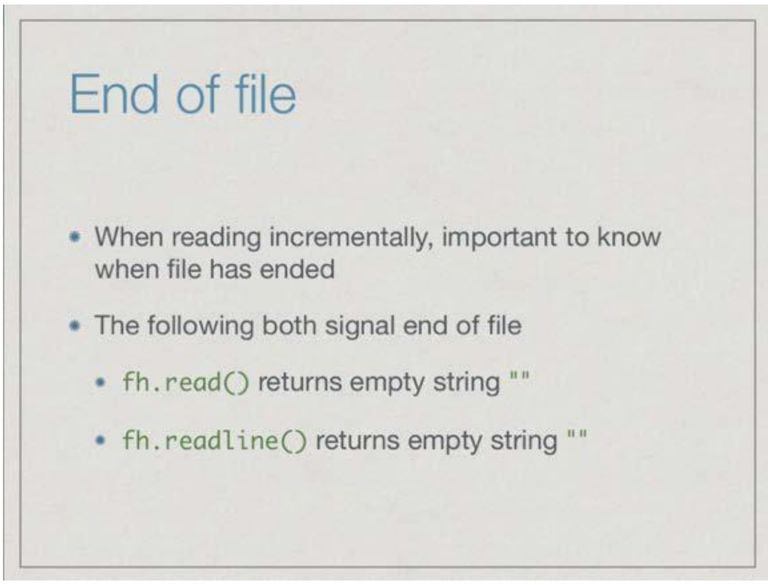
Reading files is inherently a sequential operation. Now, of course, if we use the basic command read, it reads the entire content. So obviously, it reads from beginning to the end, but if you are reading one line at a time then the way it works is that when we open the file we are initially at the beginning 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.

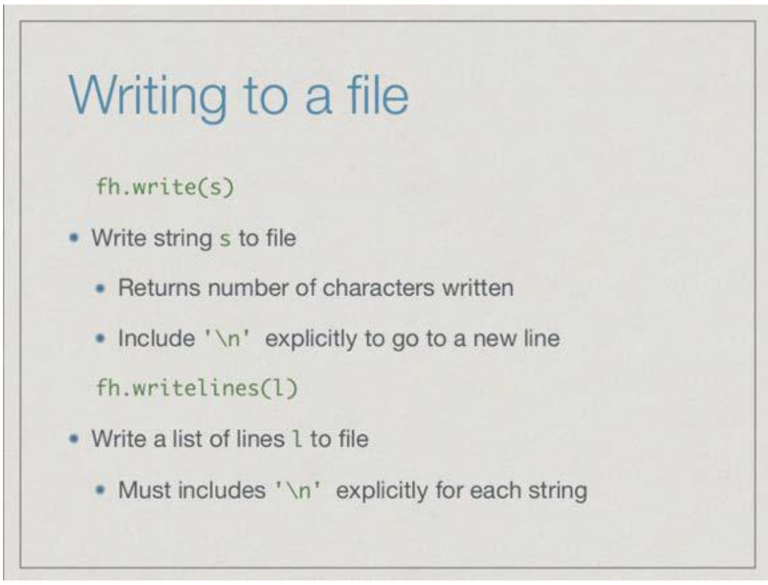
(Refer Slide Time: 09:56)




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)



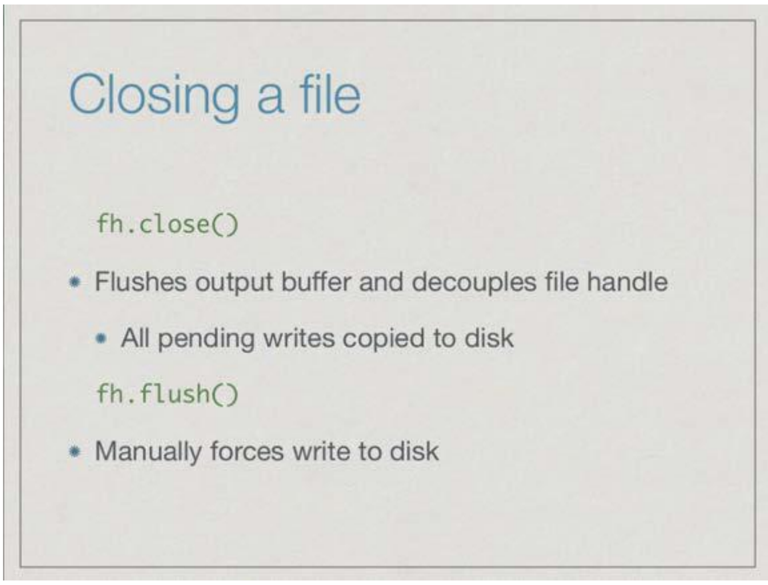

Having read from a file then the other thing that we would like to do is to write to a file. So, here is how you write to a file just like you have read a command you have a write command, but now unlike read which implicitly takes something from the file and gives to you, here you have to provide it 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.

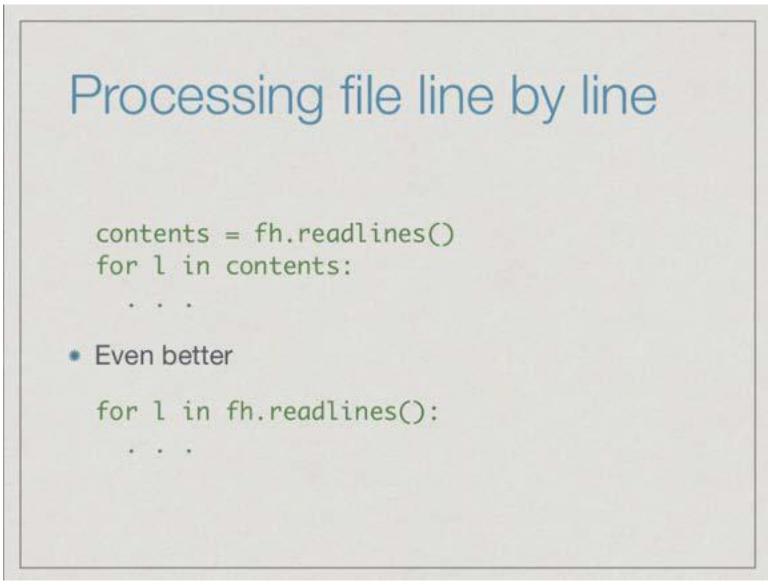
(Refer Slide Time: 12:40)




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.

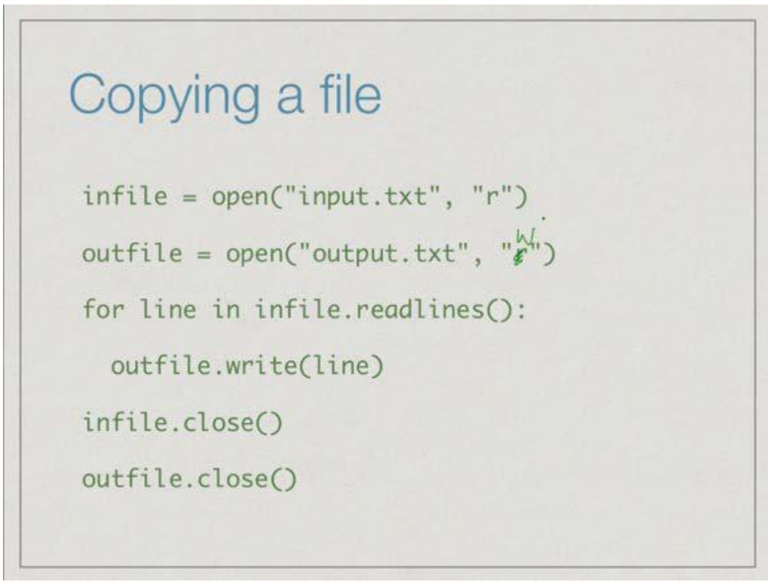
Now, sometimes there are situations where we might want to flush the buffer without closing the file. We might just want to make sure that all writes up to this point have been actually reflected on the disk. So, there is a command flush which does this. In case we say flush, it just say if there are 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)




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 do 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)



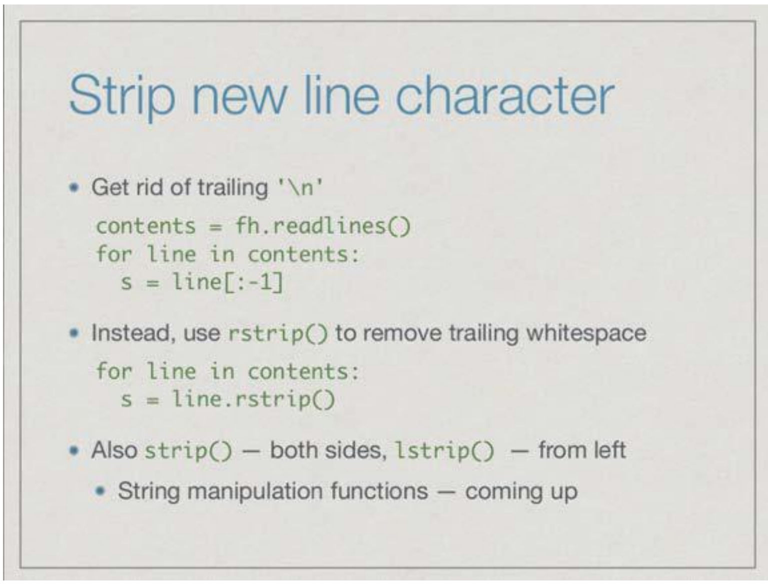
As an example, for how to use this line by line processing, let us imagine that we want to copy the contents of a file input dot txt to a file output dot txt.

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)


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 infiles I can just write it directly out.

(Refer Slide Time: 16:02)



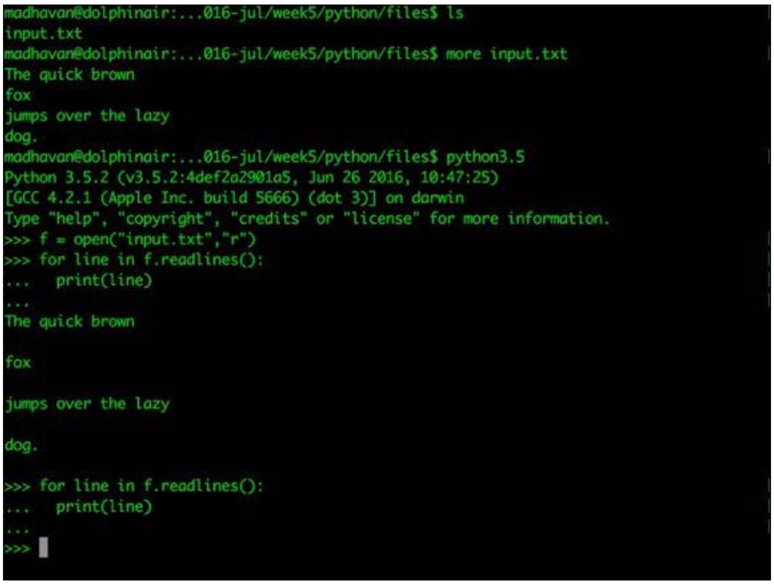

One of the things we are talking about is this new line character which is a bit of annoyance. If we want to get with a new line character, remember this is only a string and the new line character is going to be a last character in this string. So, one way to get is just to take slice of the string 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)




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.

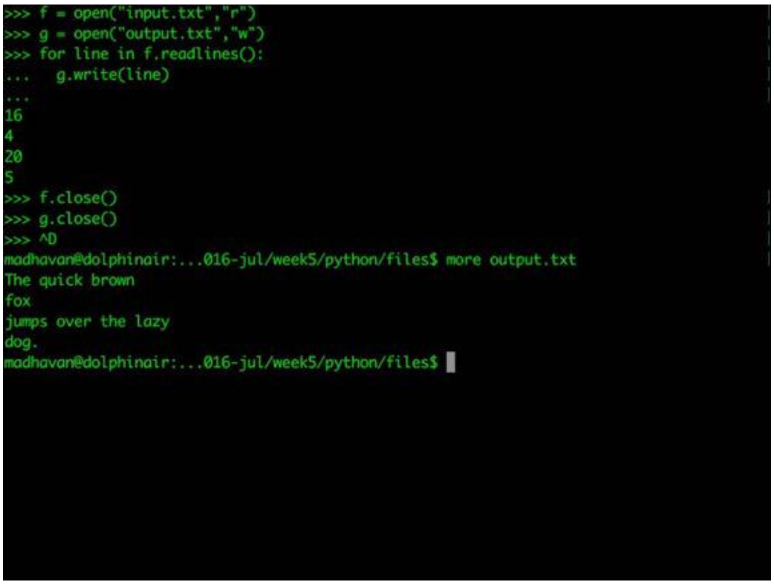
Now, let us try and do this again, supposing I repeat this thing and now I do this again, now nothing happens the reason nothing happens is because we had this sequential reading of the file. So, the first time we did f dot readlines, it read one line at a time and now we are actually pointing to the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

(Refer Slide Time: 19:49)


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.

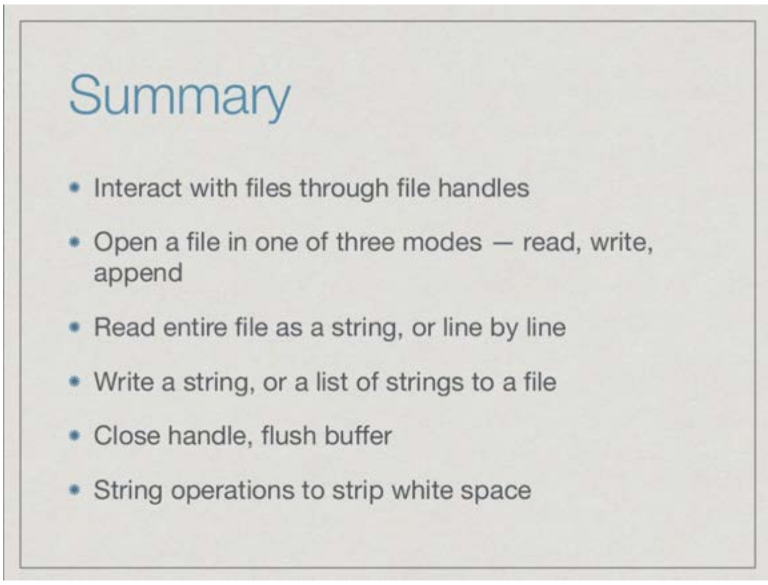
(Refer Slide Time: 21:10)




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)




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.

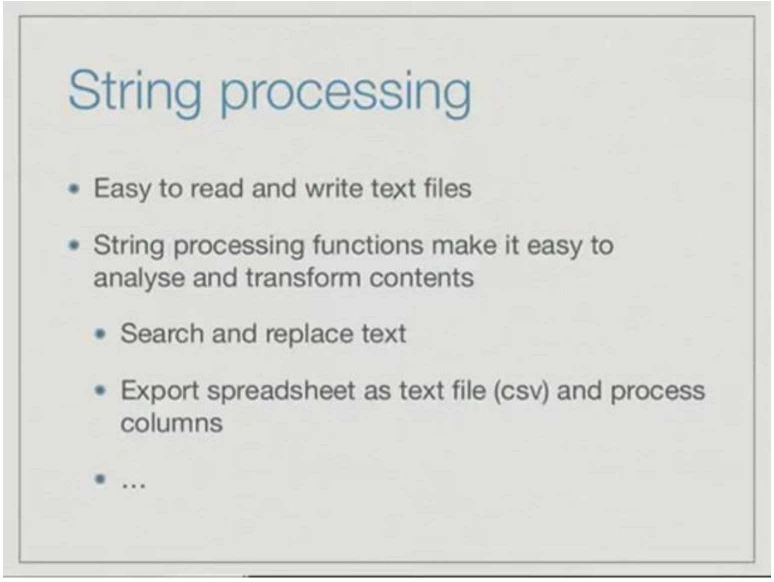
We saw that read, readline and readlines, using this we can read the entire file in one shot of the string or read it line by line. Similarly, we can either write a string or we write a list of strings too. So, we have a write command in a writelines and writelines are more correctly to be interpreters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

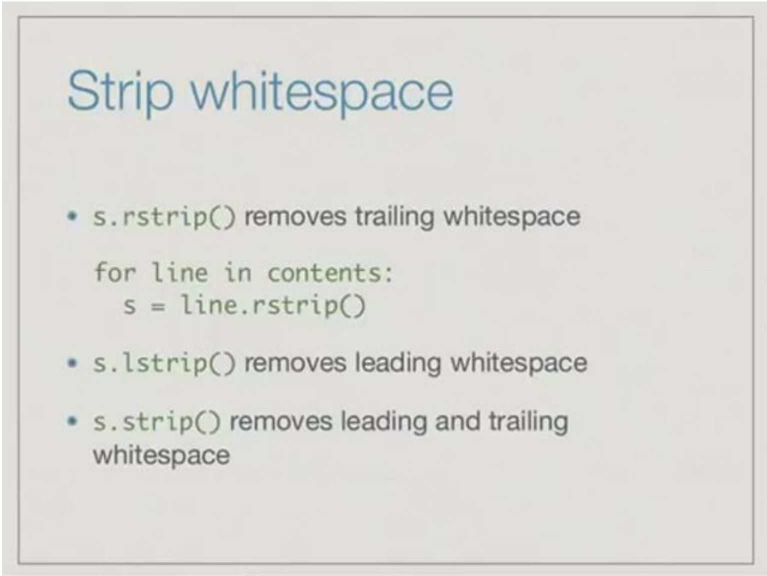
(Refer Slide Time: 00:02)




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.

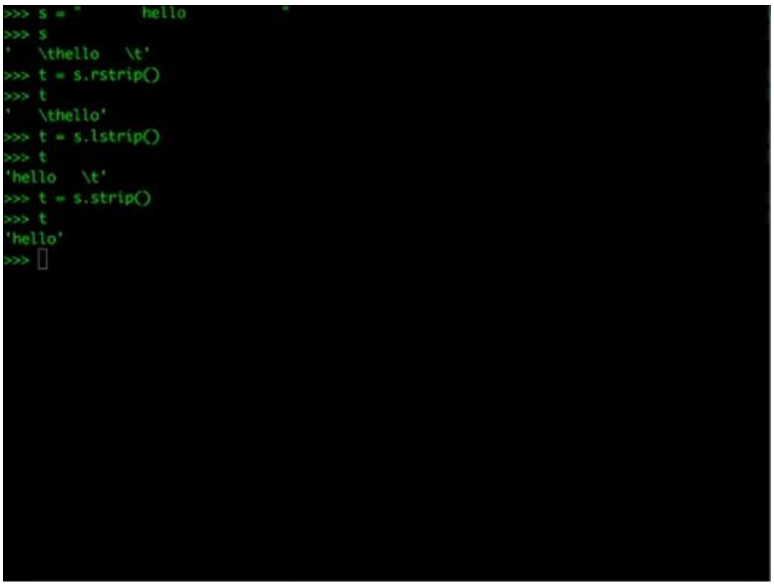
So usually, you are reading and writing files in order to do something with these files, and to do something with this you can use built in string functions which are quite powerful. Among other things, what you can do with these string functions is for example, search for text or search and replace 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)




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.

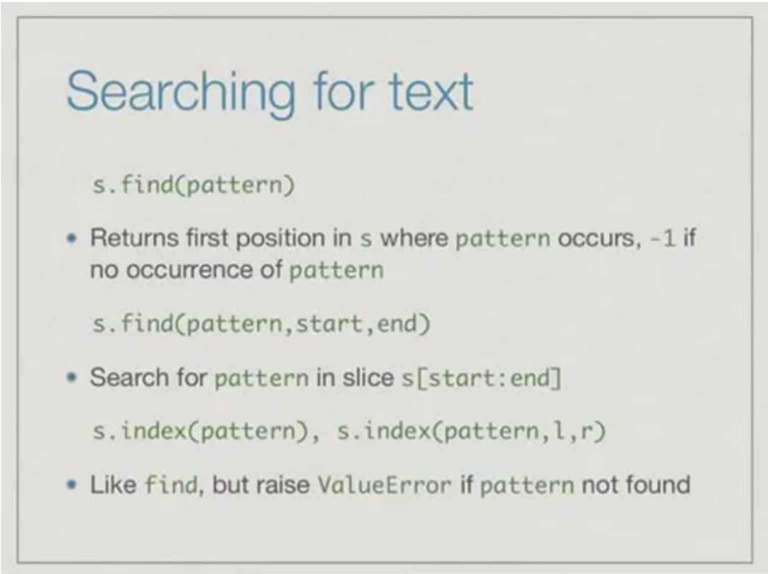
(Refer Slide Time: 01:30)




Let us create a string which as 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 read 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.

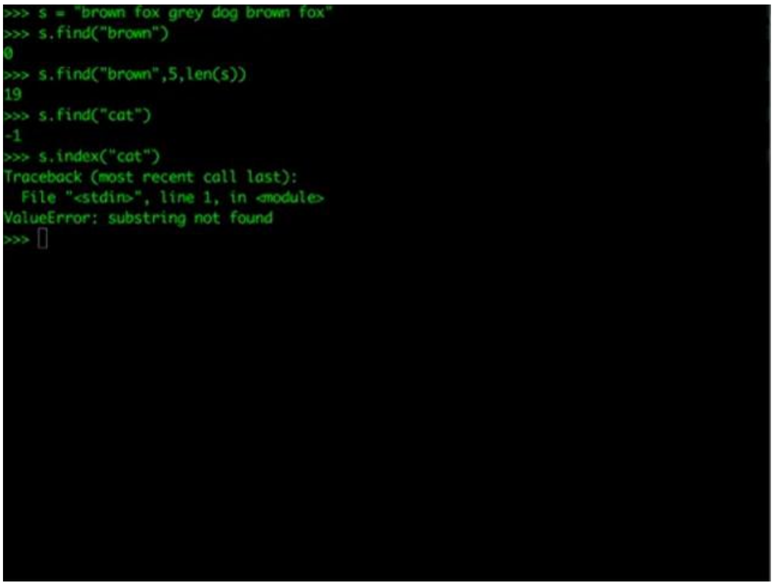
(Refer Slide Time: 02:43)




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.

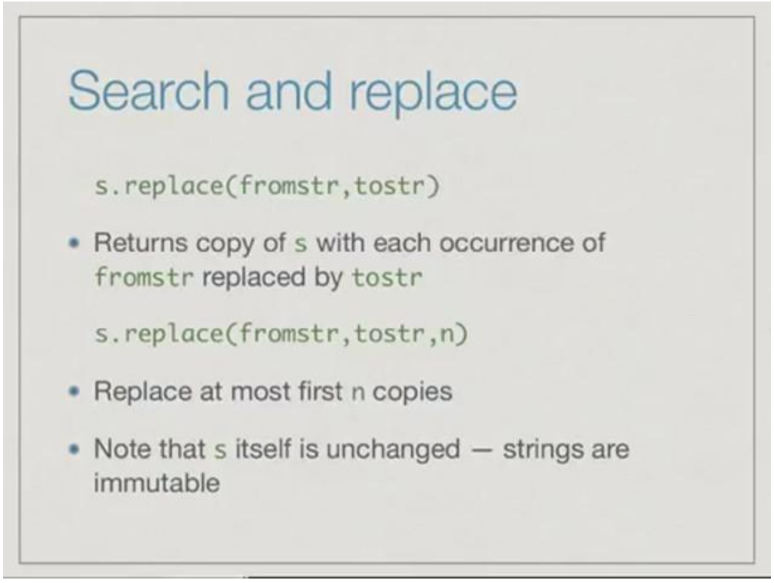
(Refer Slide Time: 03:54)




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.

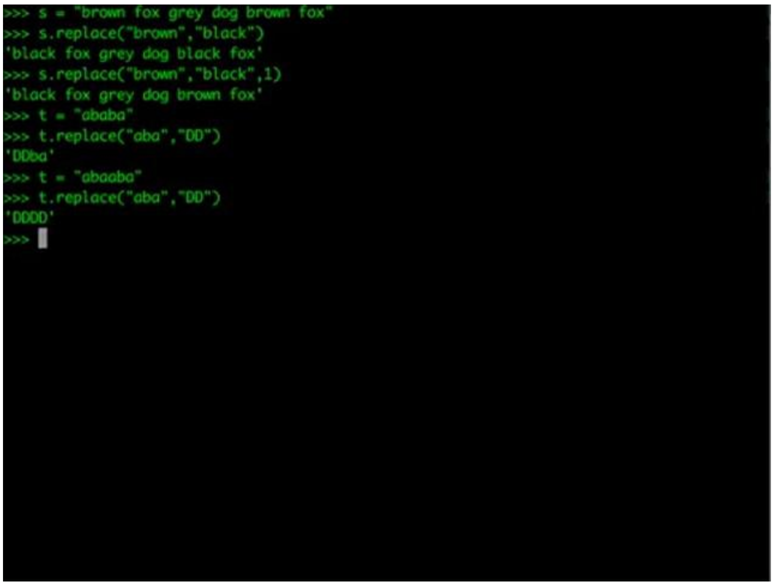
(Refer Slide Time: 04:53)




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)

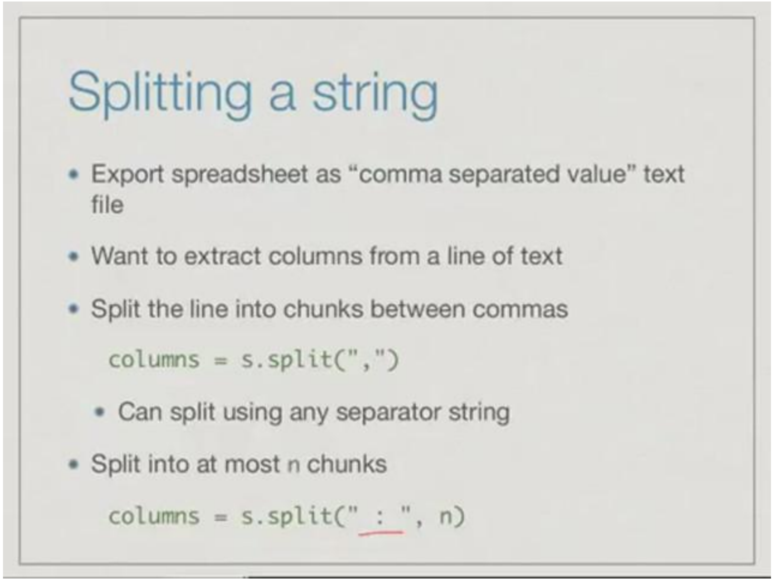



Once again let us see our old example; s is "brown fox grey dog brown fox" and now supposing I want to replace "brown" by "black", then I get 'black fox grey dog black fox.' If I say only want 1 to be replaced then I get 'black fox grey dog' where the second brown is left unchanged. Now you may ask what 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)

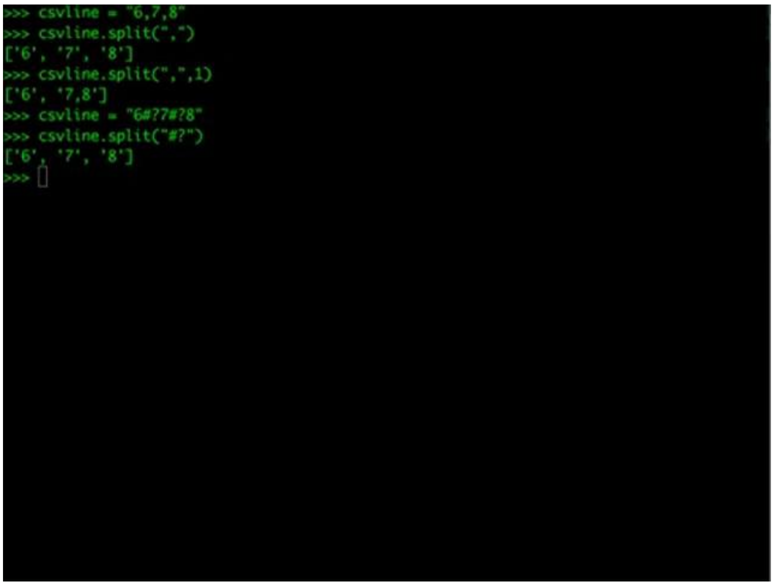



The next thing that we want to look at is splitting a string. Now when we take a spreadsheet and write it out as text, usually what happens is that we will have an output which looks like this. The first column would be written followed by comma then second column, so if we had three columns then the 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)

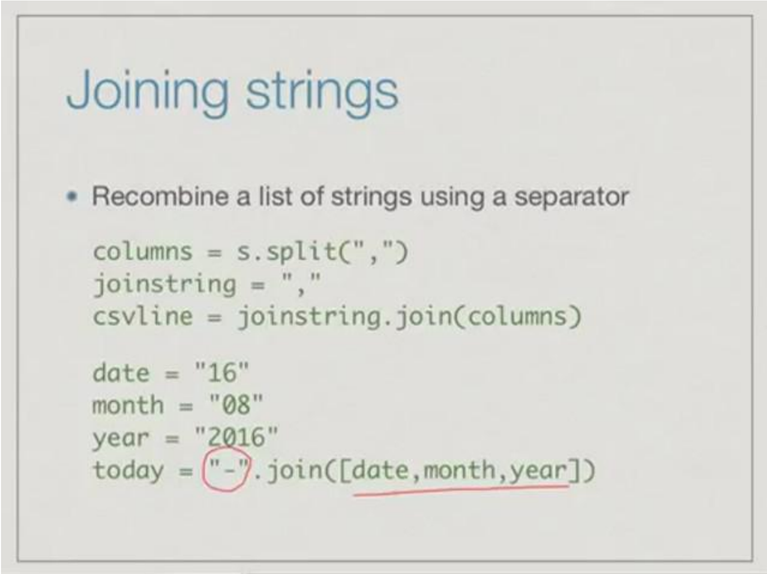



Suppose this is our line of text which I will call CSV line it is a sequence of values separated by commas notice it is a string. Now if I say CSV line dot split using comma as a separator and then I get a list of values the string 6, the string 7, the string 8. Remember this is exactly like what we said about input it does not get 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)



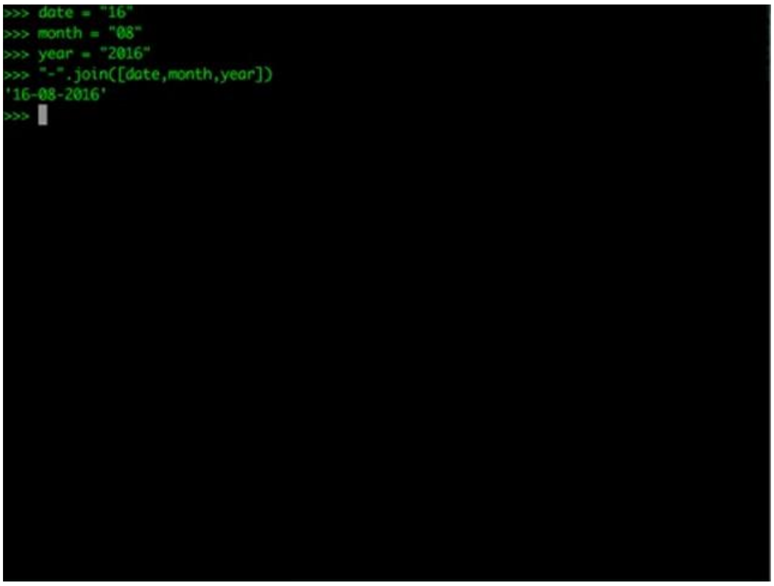

So, the inverse operation of split would be to join strings. Supposing, we have a collection of strings and I want to combine it in to a single string separate each of them by a given separator. So as an example, supposing we take s which is some CSV output and we split it into columns on comma, and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



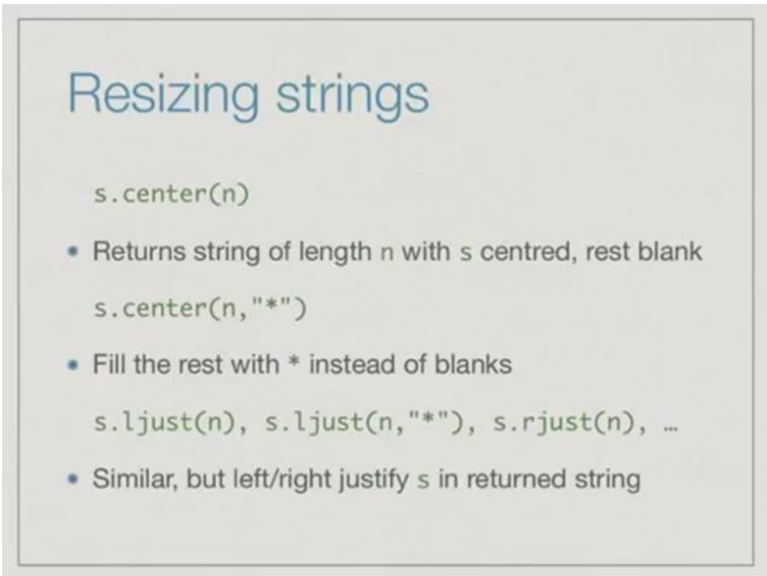

Let us directly do the second example. Supposing, we say date is 16, remember these are all strings month is 08, year is 2016, and now I want to say what is the effect of joining these three things using dash as separator and I get 16 dash 08 dash 2016.

(Refer Slide Time: 13:09)

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.

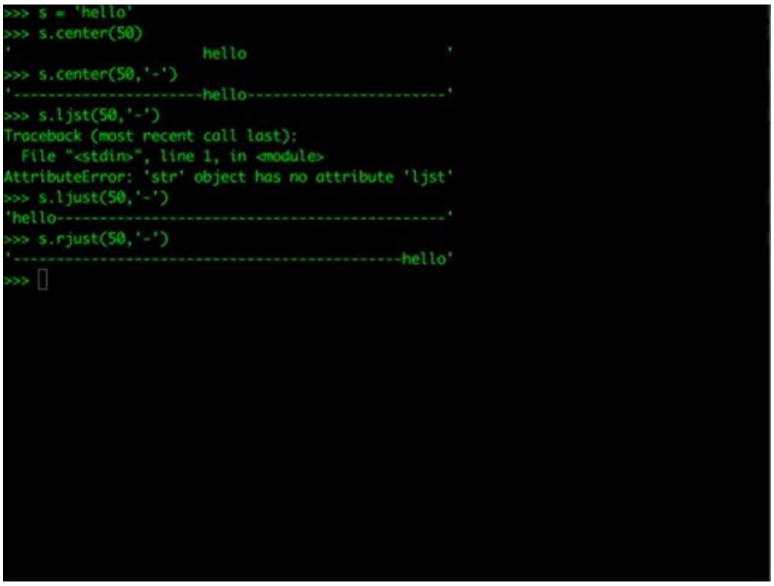
(Refer Slide Time: 13:52)




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.

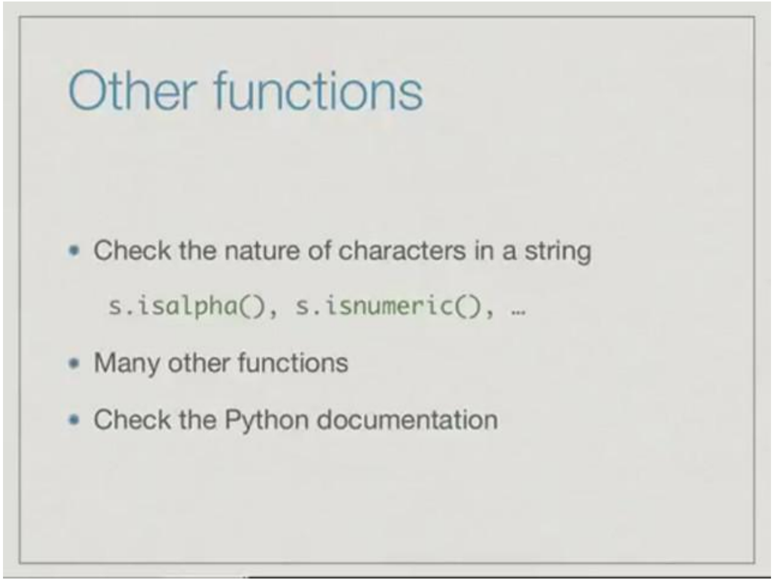
Now by centering what we mean is that on either side there will be blanks instead of blanks you can put anything you want like, stars or minuses. You can give a character which will be used to fill up the empty space on either side rather than a blank. Now you may not want it centered or you may not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

(Refer Slide Time: 14:46)




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)



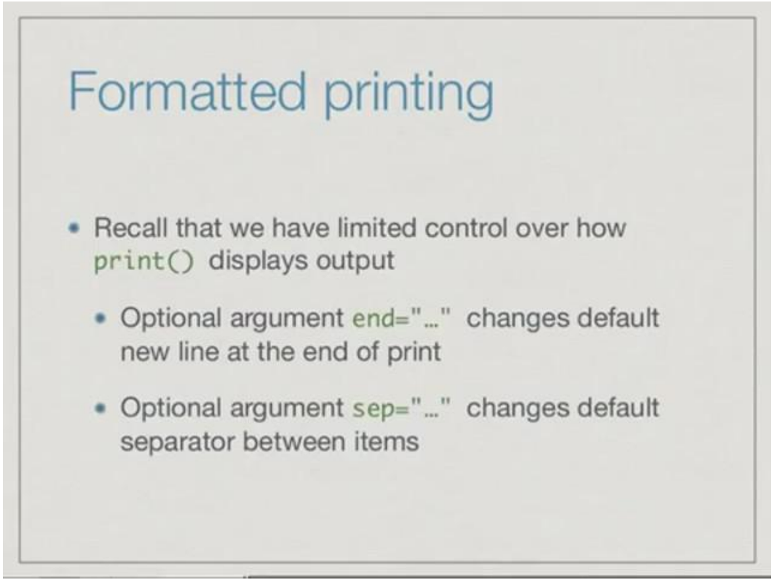

Some of the other types of functions which we find associated to strings are to check properties of strings. Does s consists only of the letters a to z and capital a to capital z. So that is what s dot is alpha says is it an alphabetic string, if it is true it means it is, if it is not it has at least 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 Pythons 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**

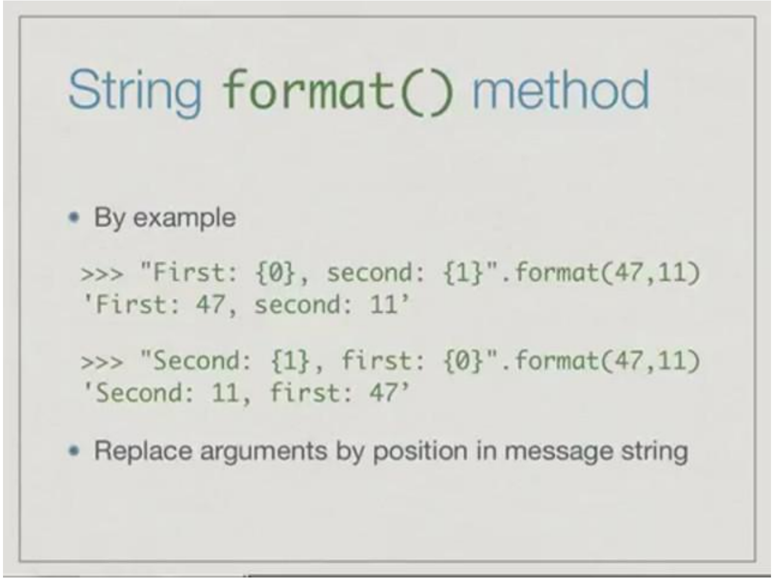
(Refer Slide Time: 00:01)




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)



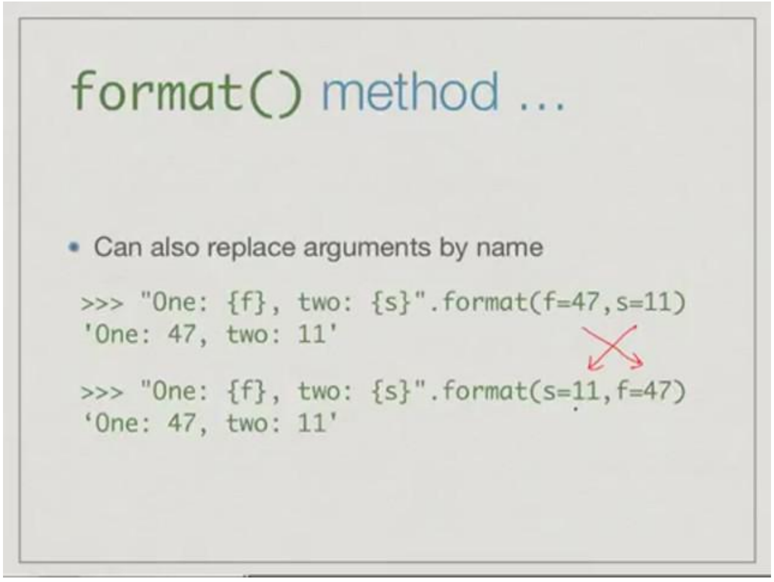

Now, sometimes you want to be a little bit more precise, so for this we can use the format method which is actually part of the string library. So the set of things that we can do with strings, last, in the previous lecture we looked at other things we can do string like, find, replace and all this things,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

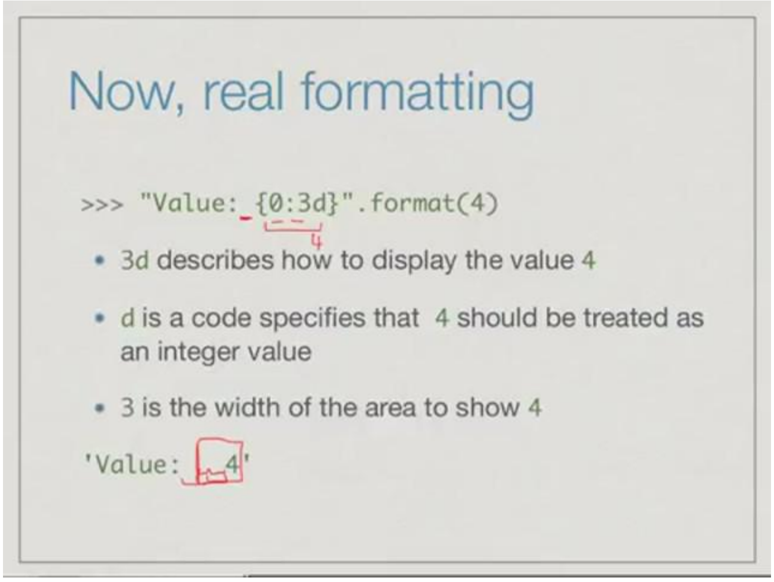
(Refer Slide Time: 02:34)




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)

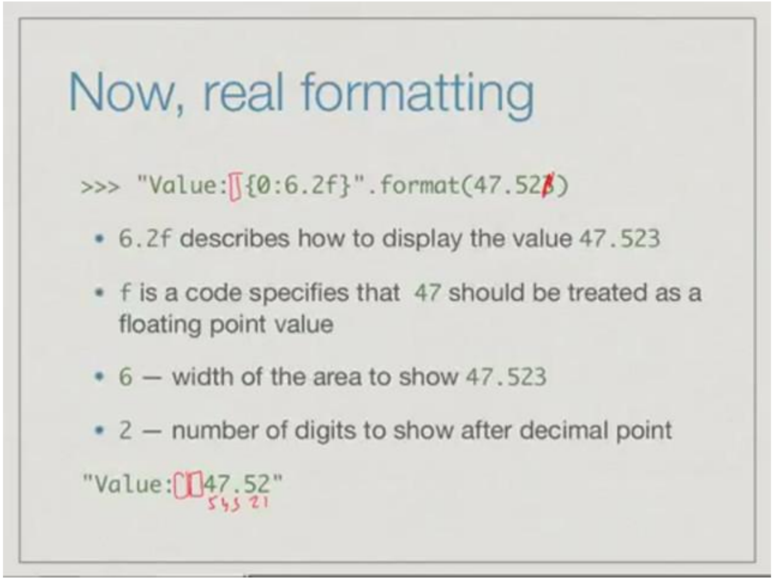



So, up to this point we have not done any formatting. All we have done is we have taken a string and we have told us how to replace values for place holders in the string. There is no real formatting which has happened because whatever we did with that we could have already done using the existing print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

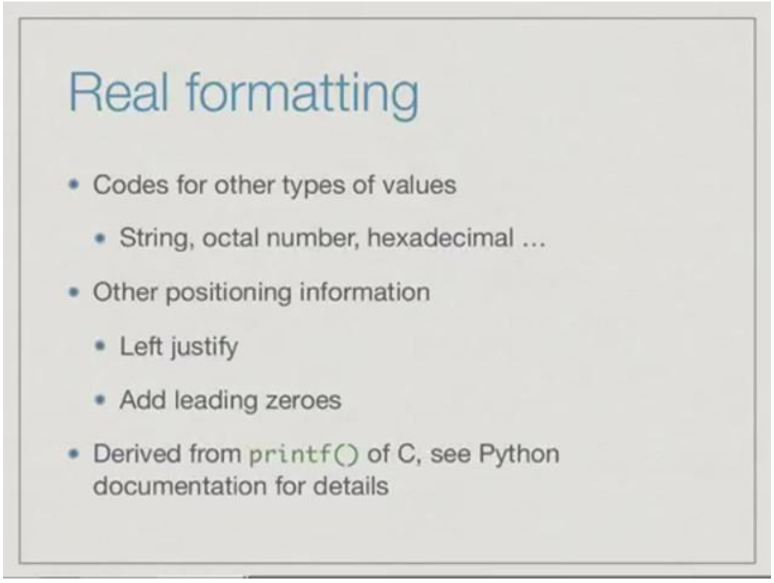
(Refer Slide Time: 05:06)




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)



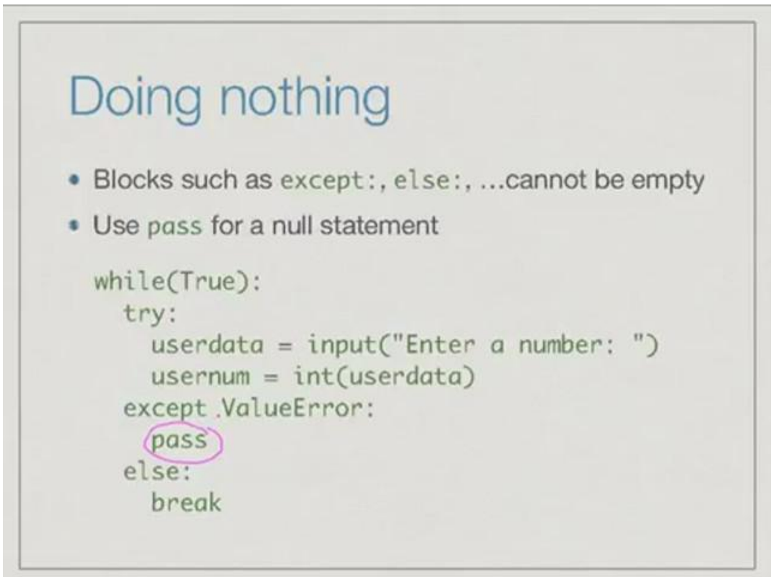

Unfortunately this is not exactly user friendly or something that you can easily remember, but there are codes for other values. We saw f and d, so there are codes like s for string, and o for octal, and x for hexadecimal. All these values can be displayed also using these formatted things. And you can 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)




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.

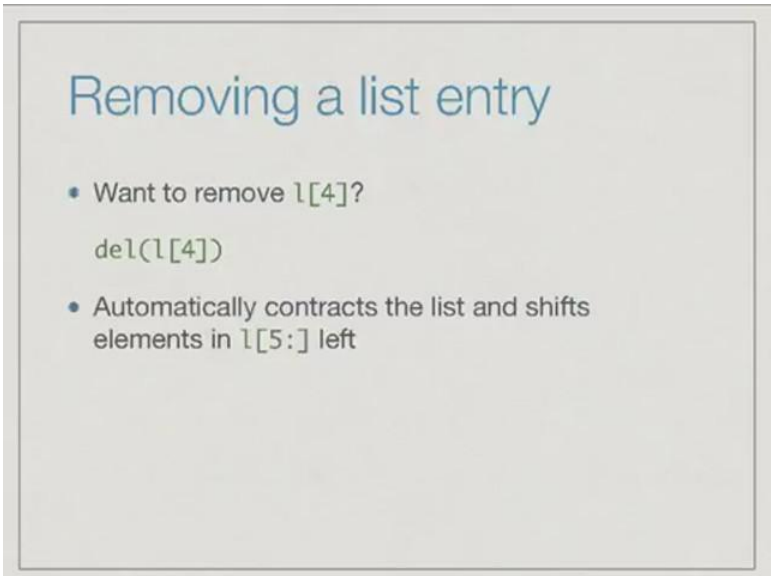
So, we had an example when we are doing the input and print about how to prompt the user in case they had an invalid entry. We were trying to read a number and what we said was that we would input some string that the user provides, so give them a message saying enter the number they provide us with 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.

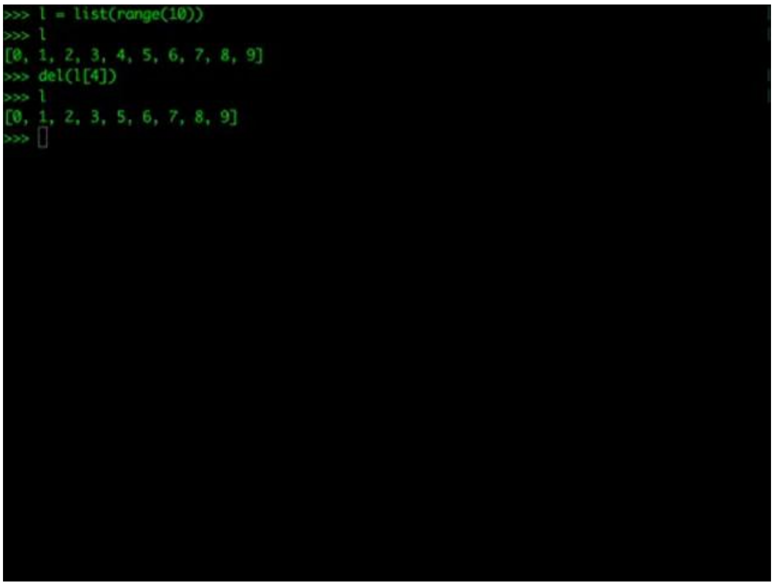
How do I do nothing in Python? So the answer is, that there is a special statement called pass. Pass is a statement which exists only to fill up spaces which need to be filled up. So when you have blocks like except or else, if you put the block name there you cannot leave it empty. In this case you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

(Refer Slide Time: 02:28)



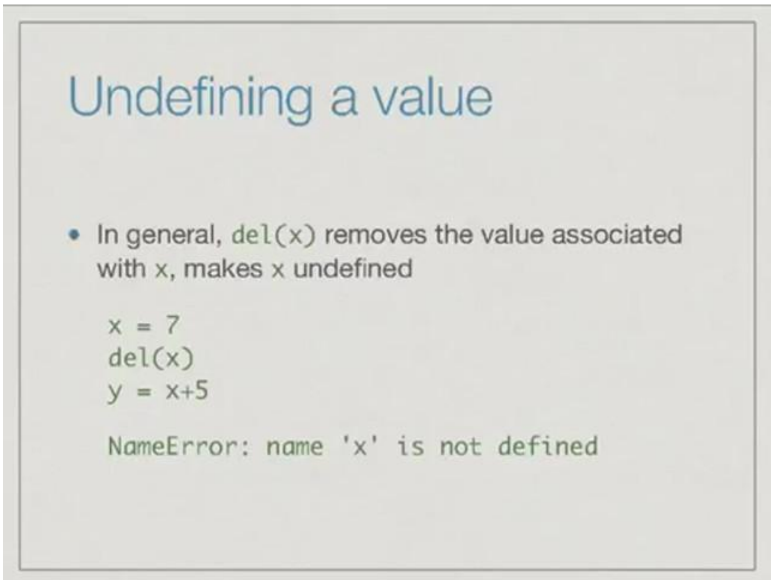

Supposing, I have a list and I want to remove an element from the middle of the list. So one way to do this of course, is to take the slice up to that position this slice from that position then glue them together using plus and so on. But what if I want to directly do this. It turns out that that there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



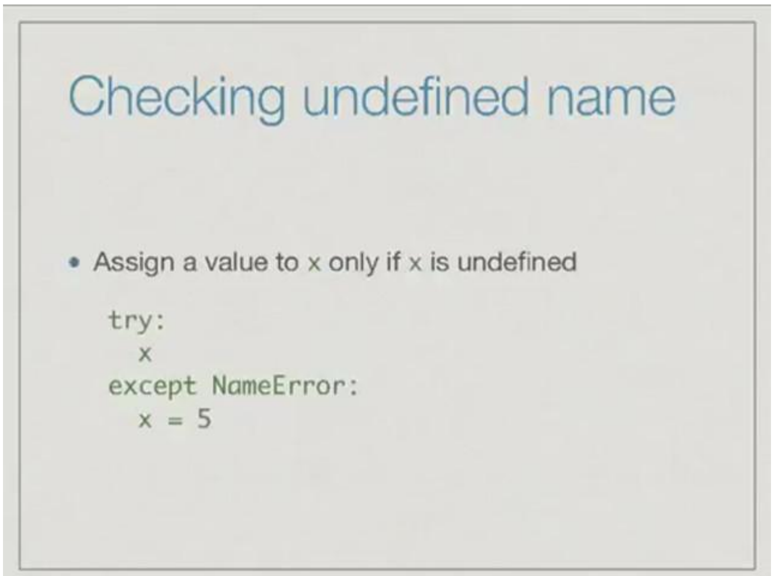

Supposing, we set our list to be the range of values from 0 to 10, 0 to 9 say, and now I say del l 4 then l becomes 0, 1, 2, 3, 5, 6, 7, 8, 9, because l 4 was the value 4 remember the position start from 0. And so it has actually deleted the value at the fourth position. This also works for dictionaries. 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)




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.

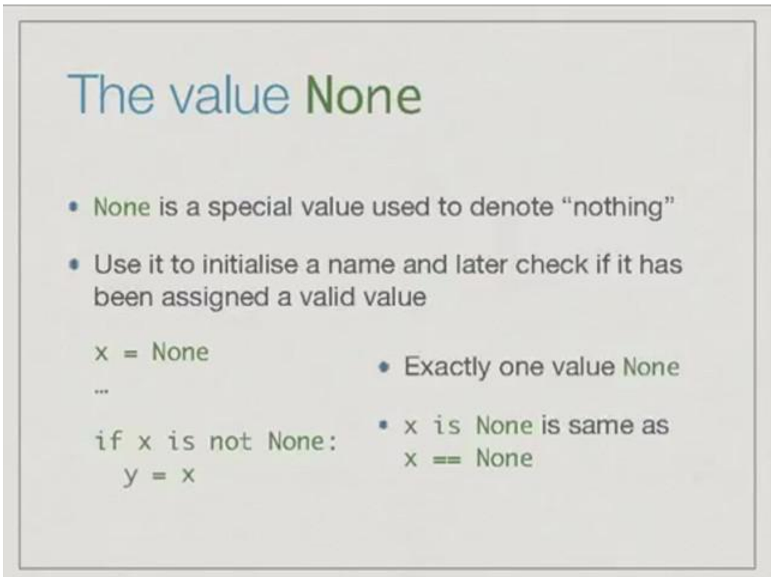
(Refer Slide Time: 04:29)




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 vale 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)

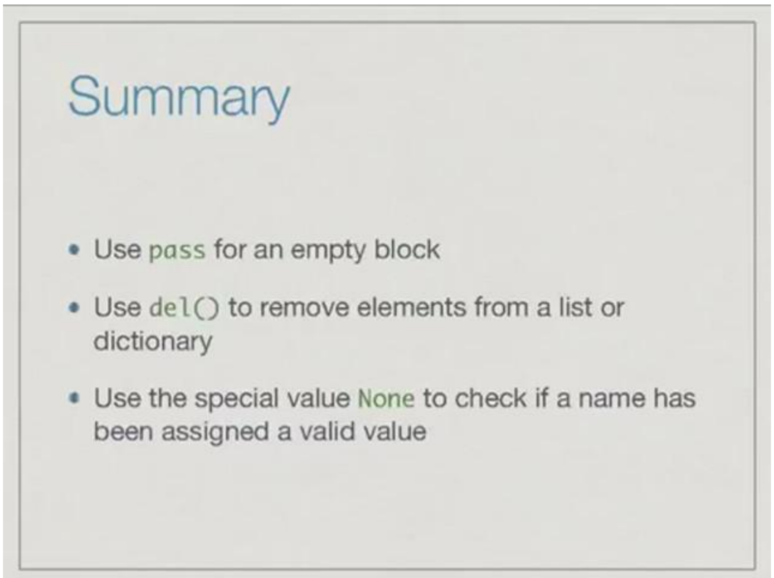



Now, usually what happens is that we want to check whether something has been defined or not so far. It is not a good idea to just leave it undefined and then use exception handling to do it, because you might actually find strange things happening. So, Python provides us with a special value called 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 Nones point to the same thing. Remember was checking when we say l 1 is l 2, we are checking whether l 1 and l 2 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 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 say.

(Refer Slide Time: 06:58)




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.

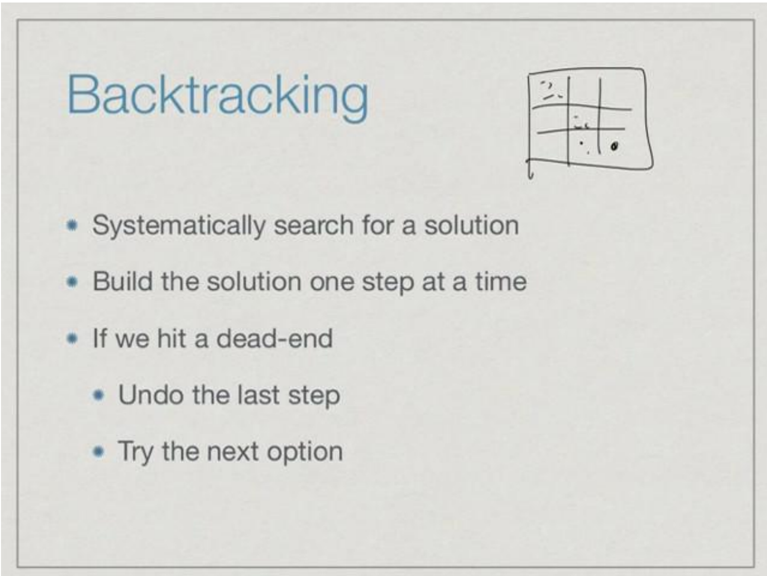
But from a dictionary or a list we might want to remove a key or if might want to remove a position and del is very useful for that. Finally, we have seen that there is a special value none which denotes a null value, it is a unique null value and this can be used to initialize variables to check whether 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.

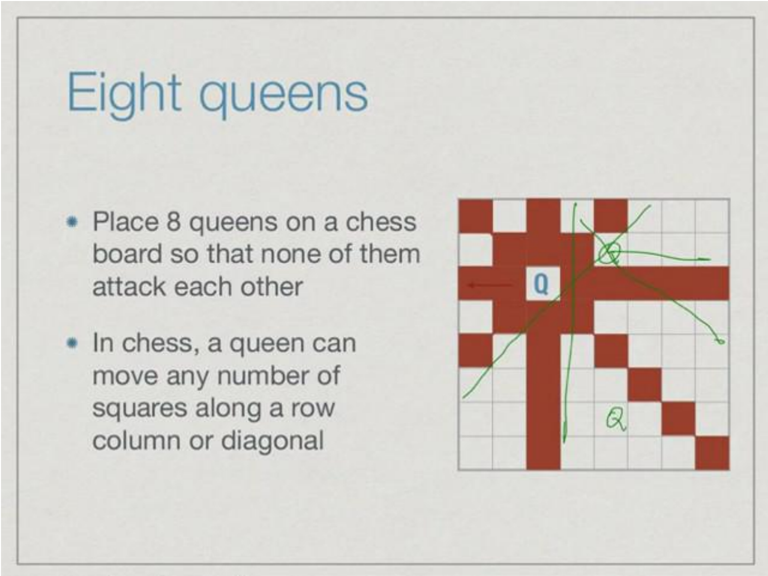
(Refer Slide Time: 00:02)




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)



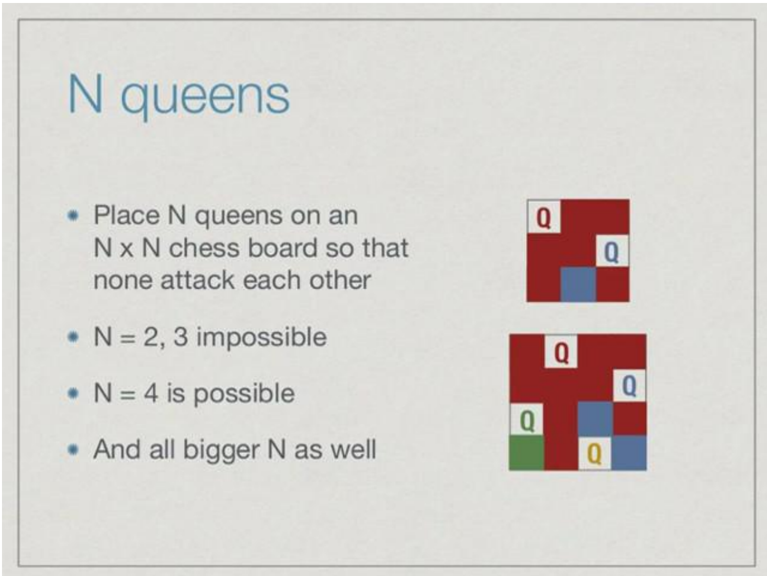

One of the classic problems of this kind is called Eight queens problem. The problem is to place 8 queens on a chess board so that none of them attack each other.

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, it 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)




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.

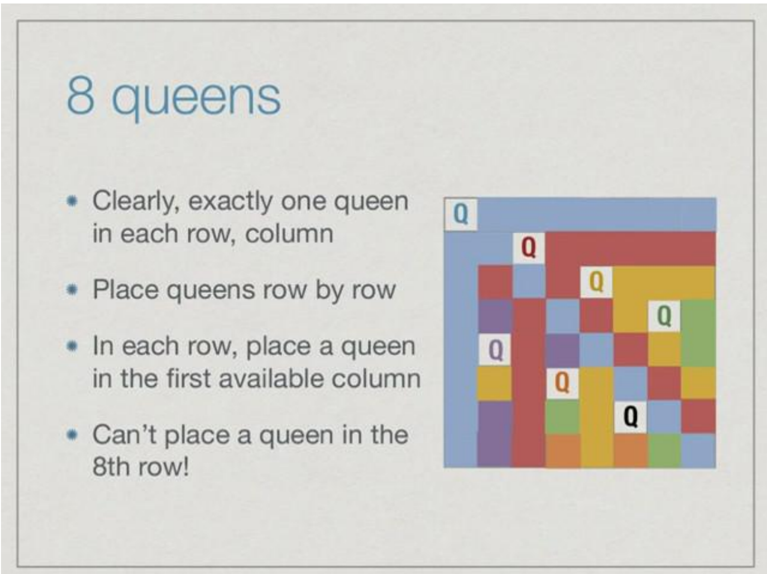
And so there is no possibility of putting a second queen. It turns out that three is also impossible. Supposing we start by putting a queen in the top left corner then we will see that it blocks out the first column, the first row and the main diagonal. This leaves two slots open for the second queen, 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 4 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)



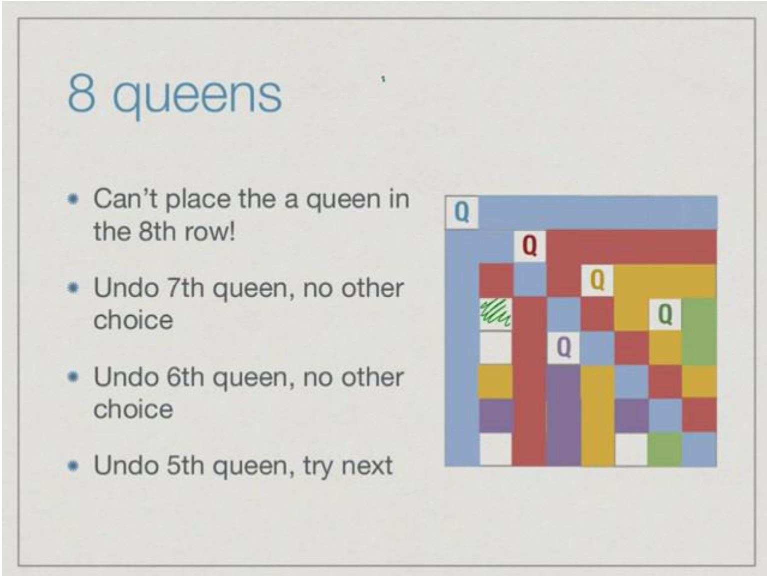

So, as you observed, the first of first thing you know is that there can be exactly one queen in each row and in each column because queens attack the column and row on which they lie. If we have two queens on the same row or the same column they will necessarily attack each other. Since 8 is the classical 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)

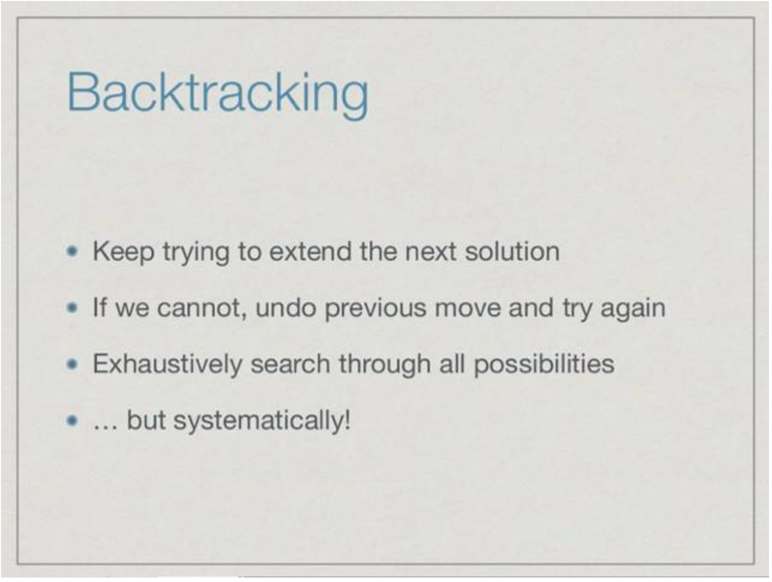



Since we cannot put a place with queen in the 8th row we have to go back and change something we did before now. The last thing we did was to put the 7th queen right. So, we do that and we find that unfortunately for the 7th queen, we had only one choice. So, we have no other choice for the 7th queen. 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.

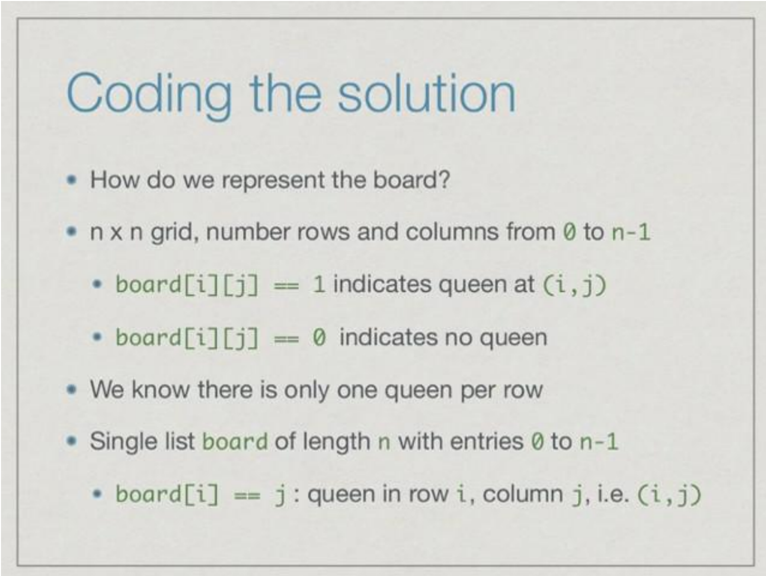
(Refer Slide Time: 07:55)




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)

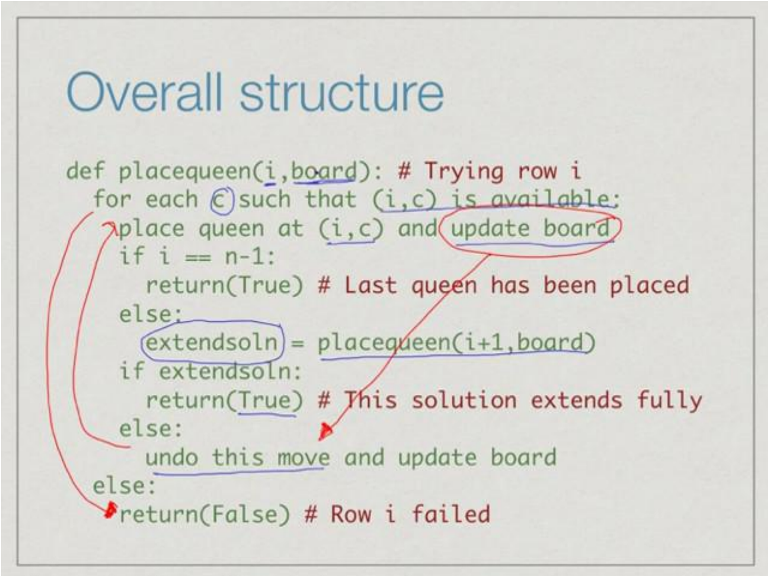



So, how would we actually encode this kind of an approach? Specifically, for the 8 queens problem, so our first question is how to represent the board because a board is what keeps changing as we make moves and undo them. The most obvious way for an N queen solution is to represent the board literally 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 a 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 ith entry corresponds to the ith 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)




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.

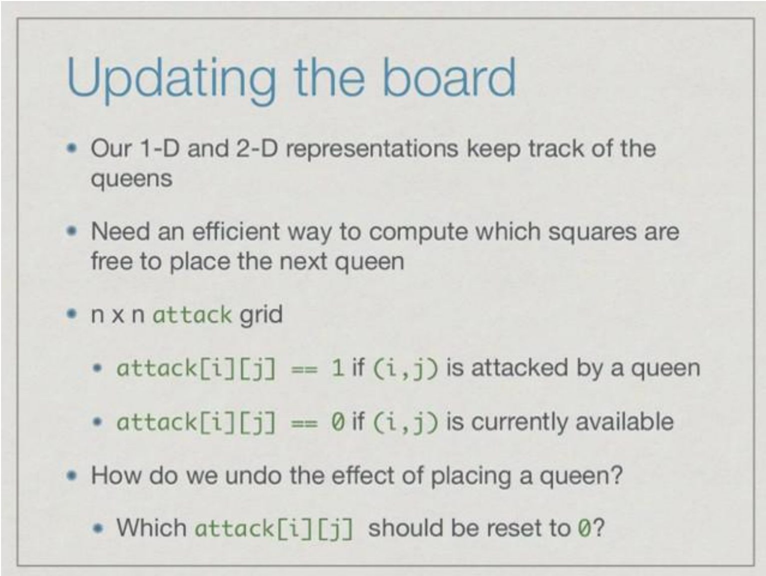
Now, when would it be true; if it succeeded in going all the way to level N minus 1 and N minus 1 returns true. So, when N minus 1 returns true then N minus 2 will return true and so on. Then our level I will also get the value true. Then we can also return true. So, if extend solution returns true we 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)

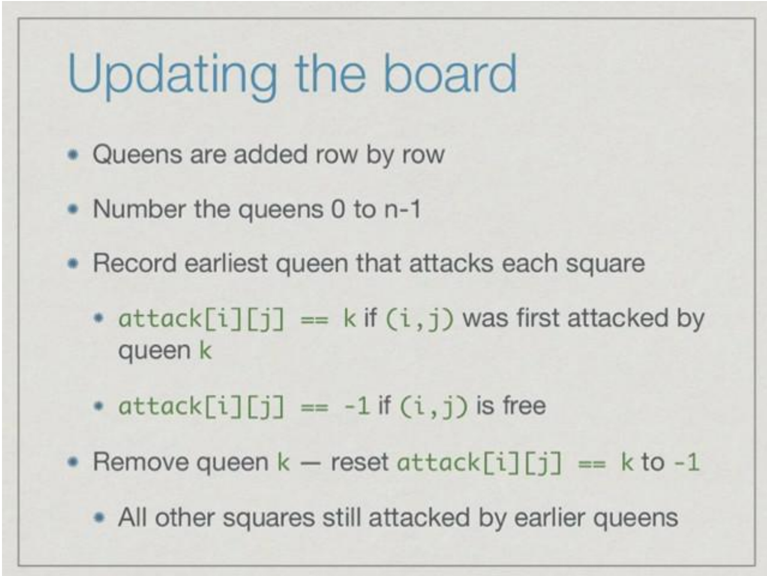



So, we had two representations, a two dimensional representation with 0s and ones and a one dimensional representation which gives us the column position for each row to keep track of the queens in the board, but in order to determine whether a square is free or not, we need to have a better way to compute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

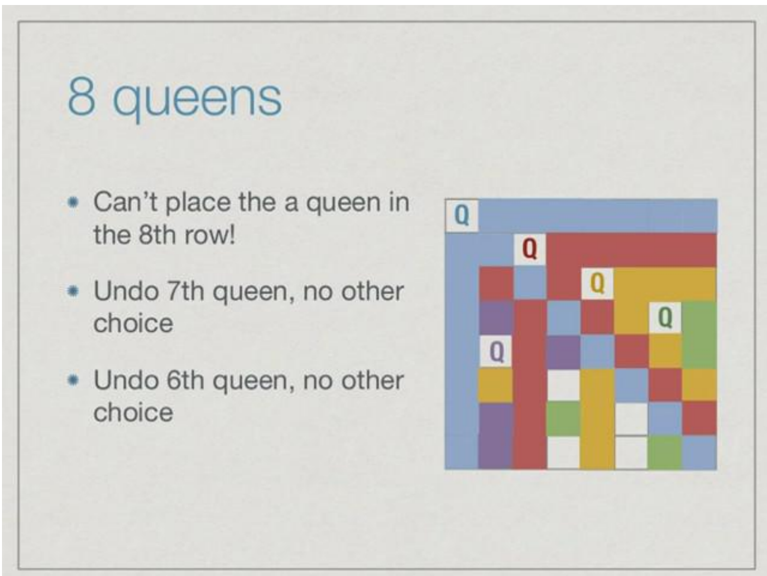
(Refer Slide Time: 15:28)




So, we say attack i j is k if i j was first attacked by queen k and attack i j is minus one if i j is free. So, when we remove queen k we reset 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)

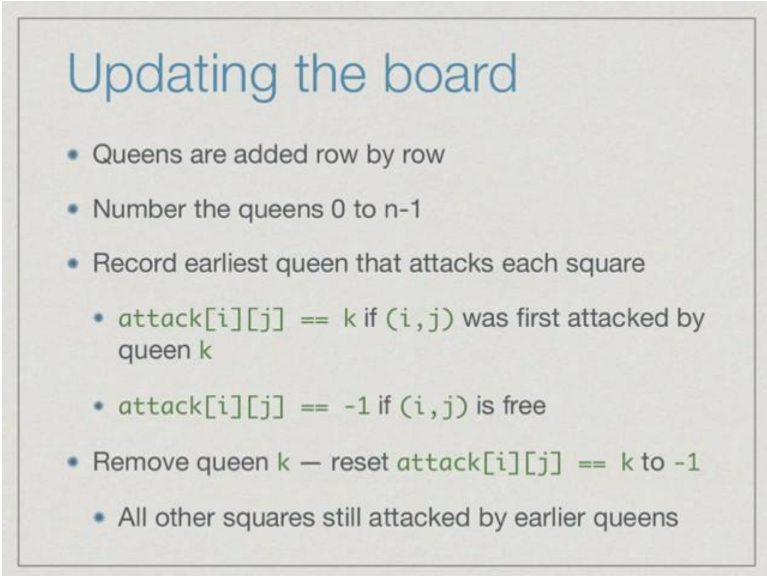


Here is how we had represented our board when we put the blue queen we marked all squares of the blue queen attacked with blue as blue solid squares then, when we put the red queen we only attack when we mark with red those squares new squares which are attacked for example, this particular square, which 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.

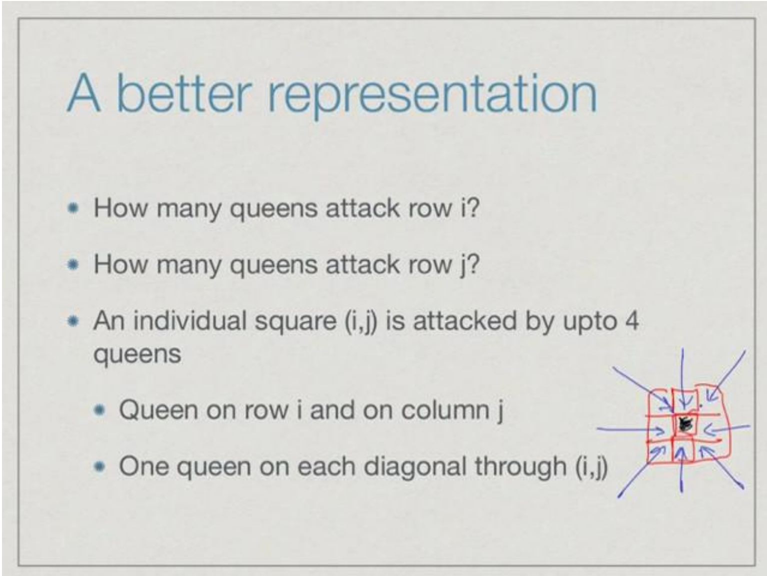
(Refer Slide Time: 17:22)




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.

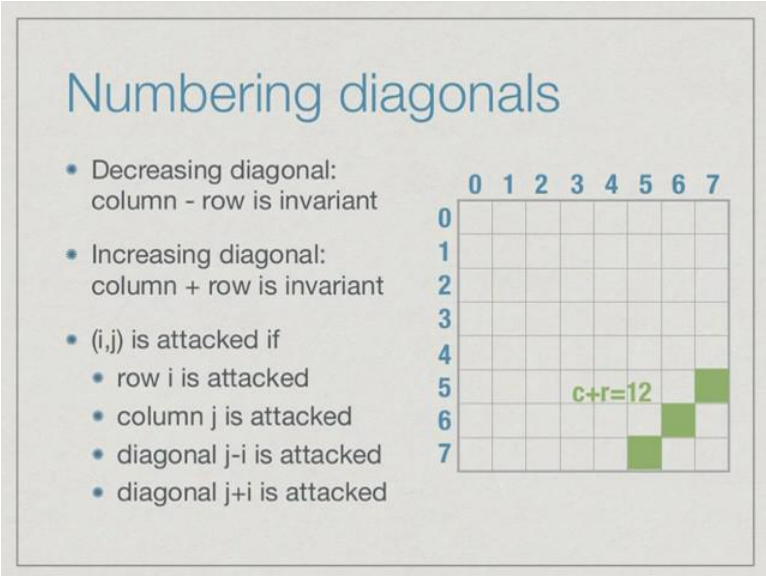
(Refer Slide Time: 18:35)




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)



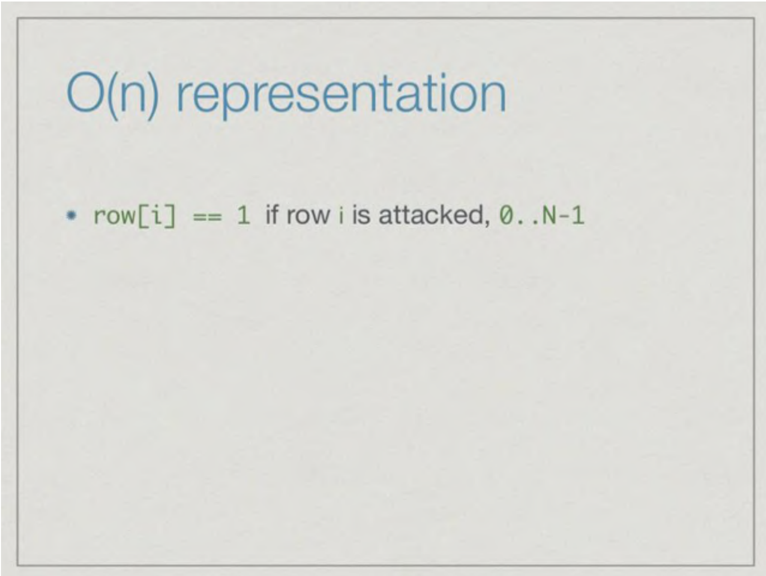

So, rows and columns are naturally numbered from 0 to 7, but how about diagonals. Now if we look at a diagonal from the north west. Let us call these directions north west, south west, north east and south east. If you look at a decrease in diagonal a diagonal that goes from top to bottom like this, then what we find as 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 minus 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 plus 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 plus 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 is along the diagonal whose difference is j plus i whose sum is j plus i.

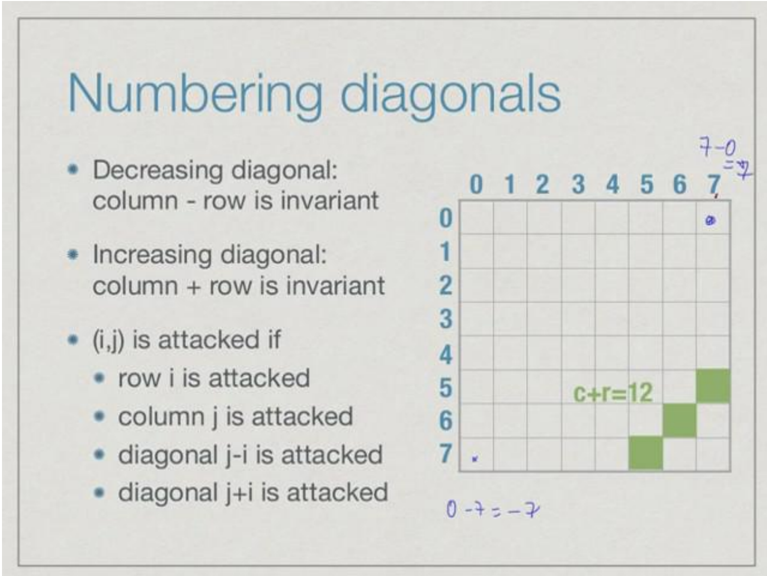
(Refer Slide Time: 22:00)




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.

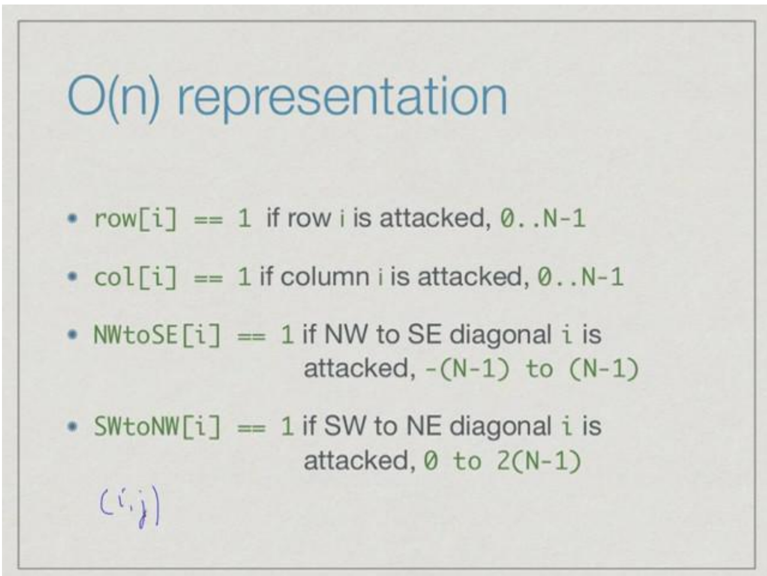
The north west to south east diagonal is the one where the difference is the same and if you look at the differences, if you go back then you see the differences at this diagonal here, the difference is 7 minus 0 is 7 and here the difference is 0 minus 7 is minus 7.

(Refer Slide Time: 22:48)




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.

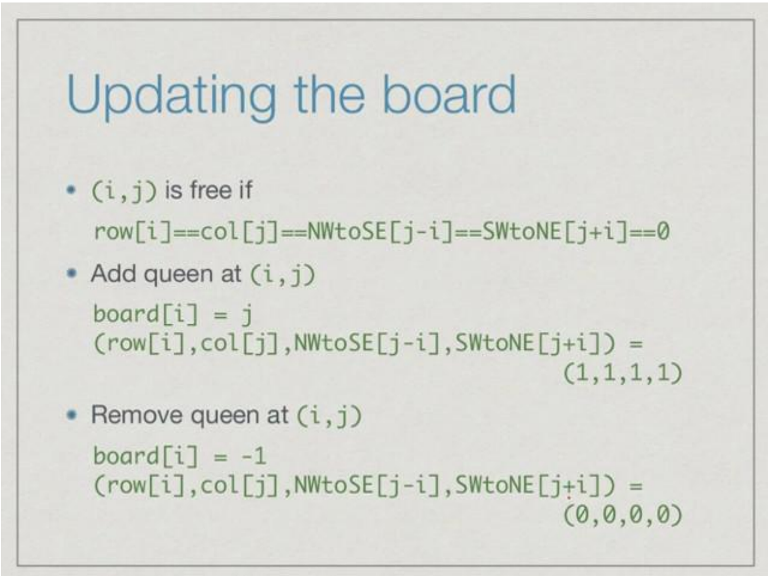
(Refer Slide Time: 23:11)



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)



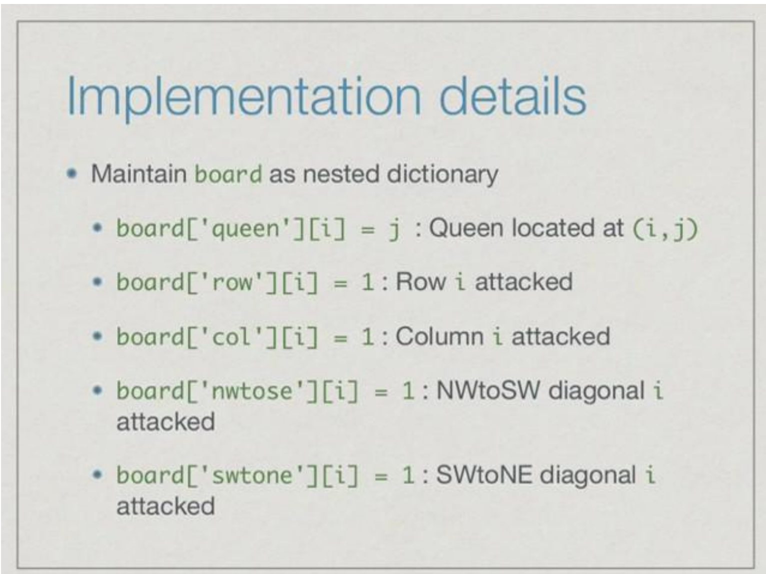

So, i j is free provided row i column j the north west to south east diagonal j minus i and the south west to north east diagonal j plus i are all equal to 0. When we add a queen at i j first we update the board representation to tell us that there is, now the ith row is set to the jth column and for 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)




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 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 ith 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.

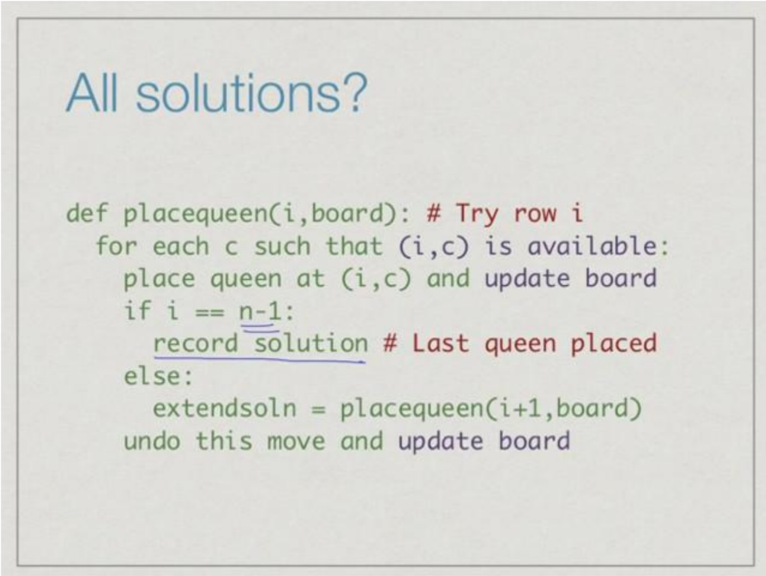
That is why we use a two level nested dictionary. This initializes the board; what how do we print a board well for every row we sort the rows. So, we take board dot queen dot keys will give us 0 1 upto N minus 1 in some random order we sort them and for each such row we print the row and the column 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 minus i is 0 and the diagonal entry j plus 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)




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.

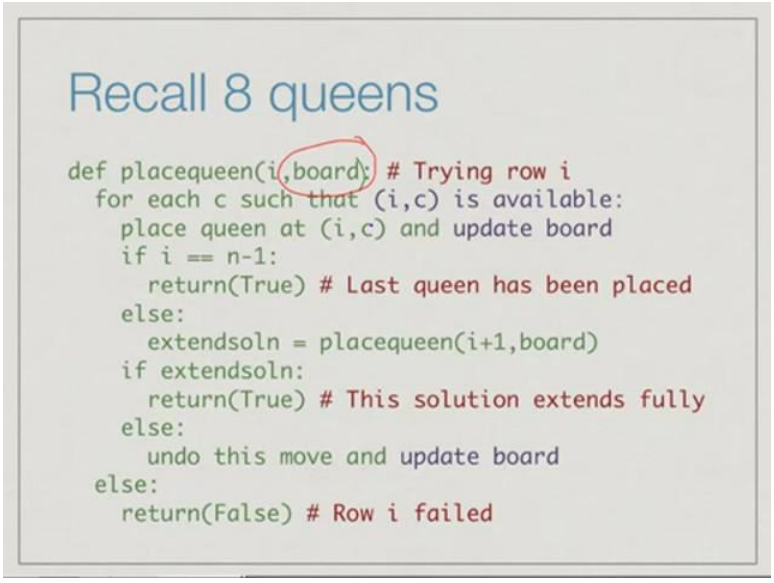
For every j we are going to first add the queen, if it manages to place it extend the solution and finally, we are going to undo it and try the next j; we're just going to blindly try every possible j and we are not going to ever come out complaining that we have not succeeded the rest is pretty much 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

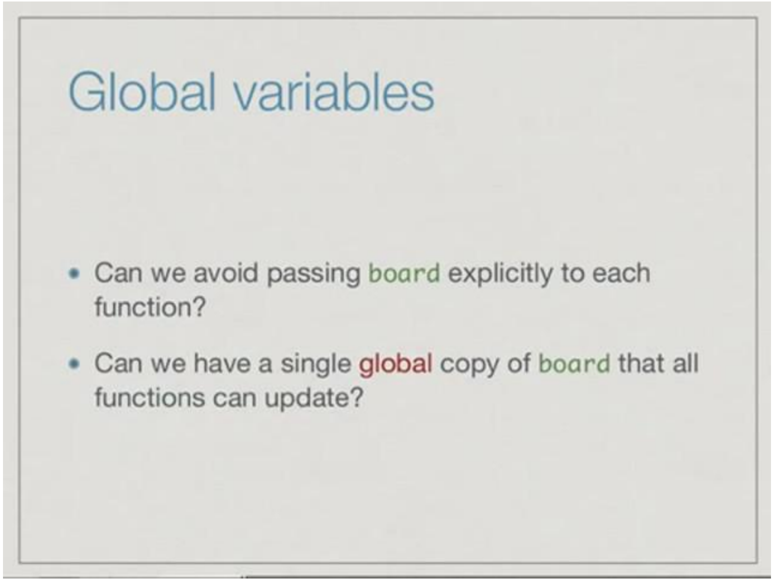
(Refer Slide Time: 00:02)




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.

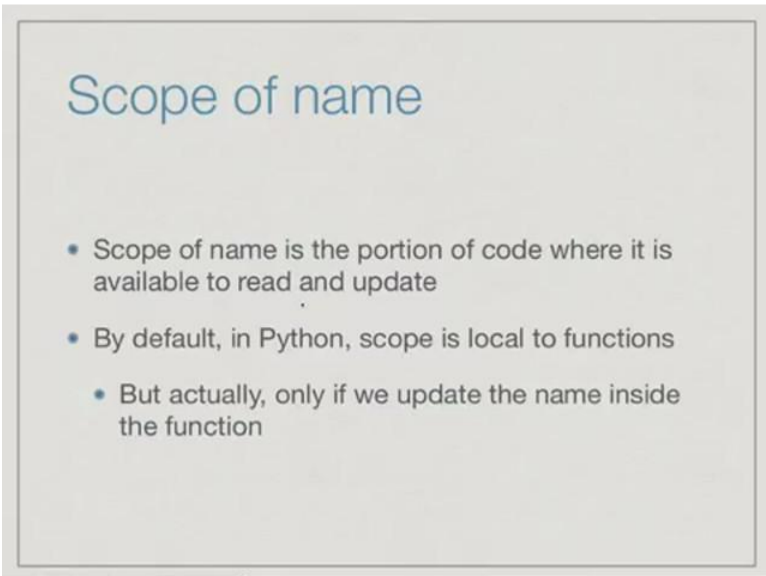
One feature of this solution is that we had to keep passing the board through the functions in order to update them or to resize them and I had to initialize them and so on because the board had to kept updated through each function. Now the question is can we avoid passing the board around all over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

(Refer Slide Time: 00:31)



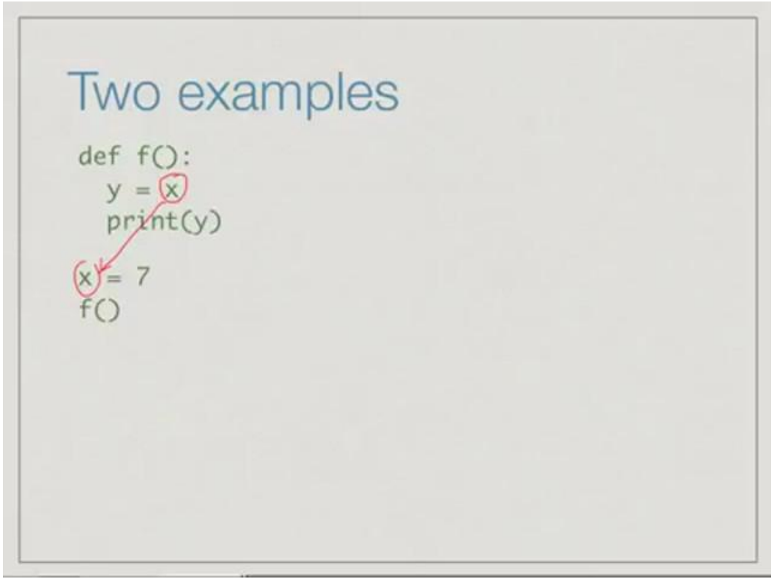

So, can we avoid passing this board explicitly or can we have a single global copy of the board that all the functions can update which will save us passing this board back and forth.

(Refer Slide Time: 00:42)



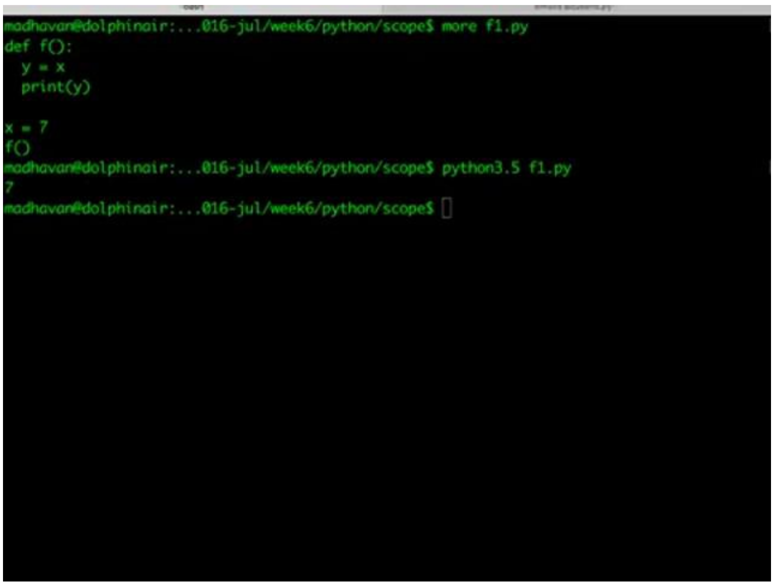

So, this brings us to a concept of Scope. The scope of a name in Python is the portion of the code where it is available to read an update. Now by default in python scope is local to a function, we saw that if we use a name inside a function and that it is different from using the same name outside the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

(Refer Slide Time: 01:08)



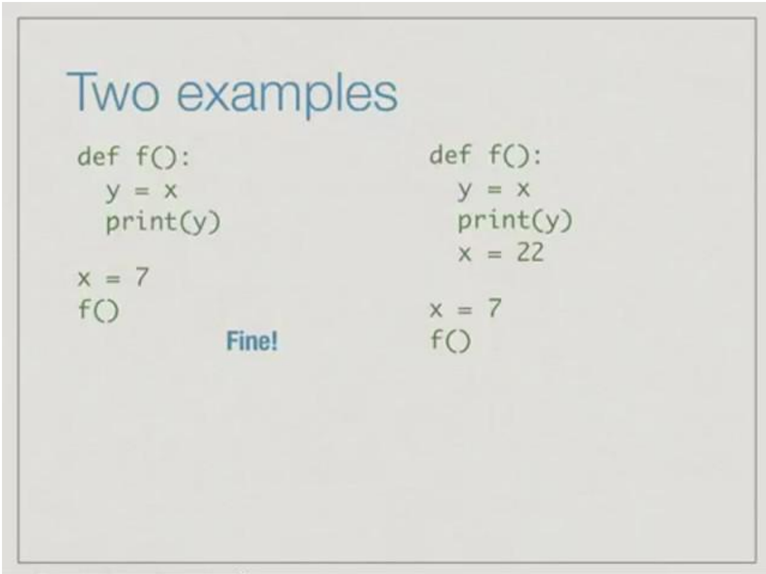

Let us look at this particular code. Here we have a function f which reads the values x and prints it by storing it in the name y, Now the question is: what is this x? Well there is an x here. So, will this x inside the function correctly reflect the x outside the function or not.

(Refer Slide Time: 01:33)



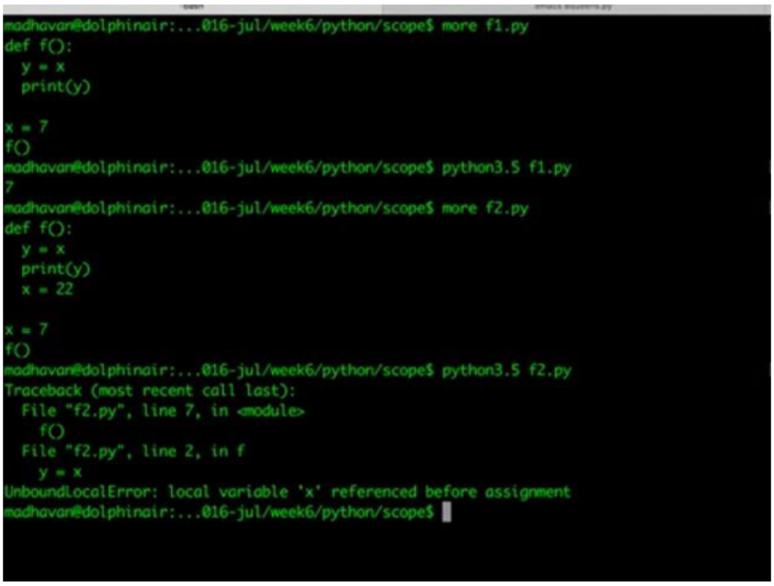
So here we see that function, we have written a file f1 dot py which contains exactly that code. So, we have function f which reads an x from outside and tries to print it. If you run this, then indeed it prints the value 7 as we expect, so y gets the value 7 because the x has the value 7 outside and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

(Refer Slide Time: 02:03)



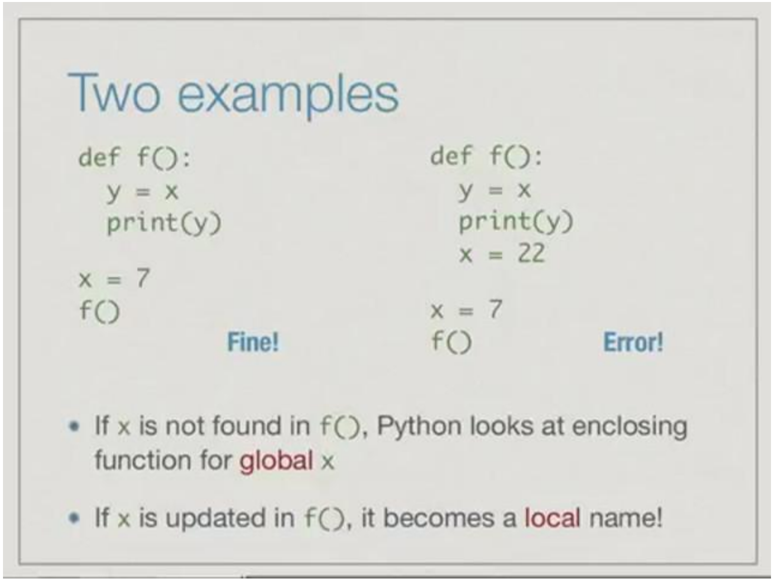

So this works. Now what if you do this, and this is exactly the same function except that after printing the values of y it sets x equal to 22 inside f. Now what happens?

(Refer Slide Time: 02:19)



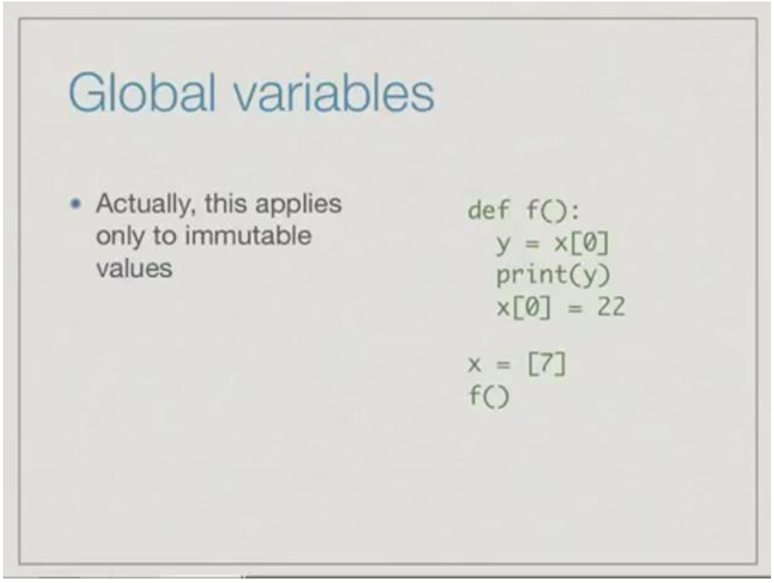
So here is f2 dot py the code in the middle of the screen, so only difference with respect to f 1 dot py is extra assignment x equal to 22 inside f. Now if you try to run f 2 dot py, then it gives us an error saying that the original assignment y equal to x gives us an unbound local name there is no 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)



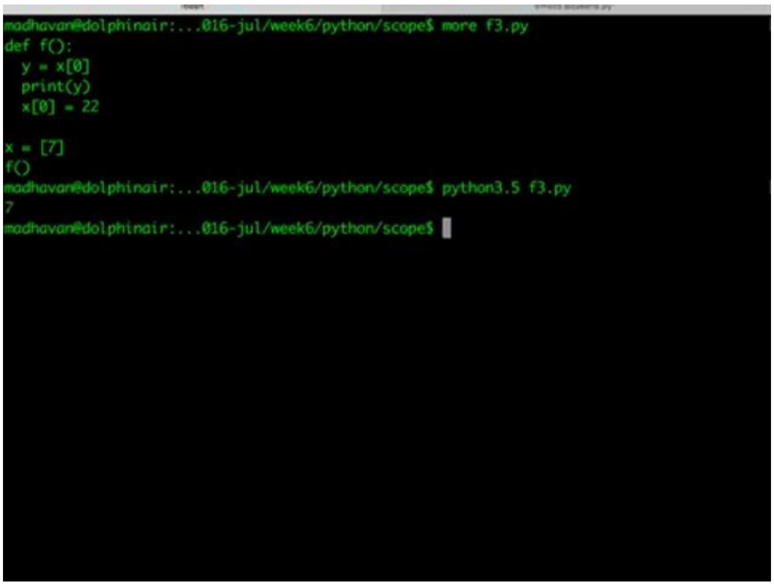

So if x is not found in f, Python is willing to look at the enclosing function for a global x. However, if x is updated in f then it becomes a local name and then it gives an error.

(Refer Slide Time: 03:15)



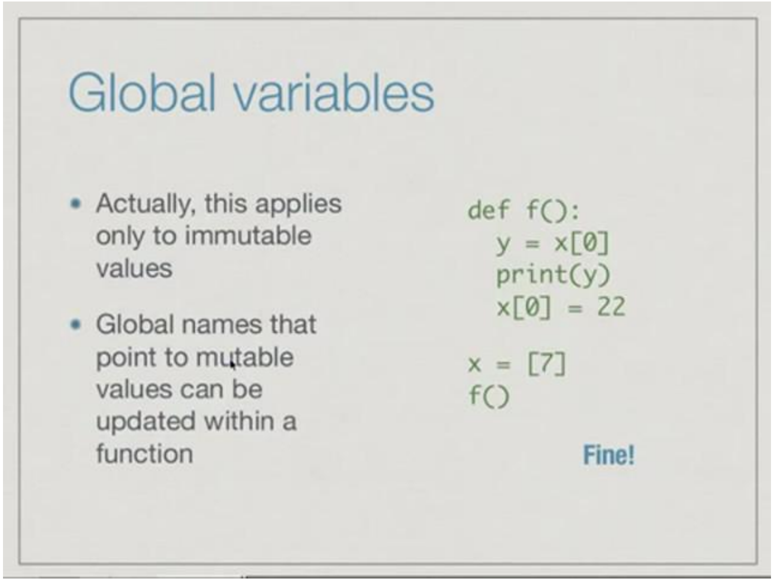

So strictly speaking this applies only to immutable values. If we change this function as follows we made x not an integer, but a list for example and we asked y to pick up the 0th element in the list and then later in f we change the 0th element of x to 22.

(Refer Slide Time: 03:38)




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.

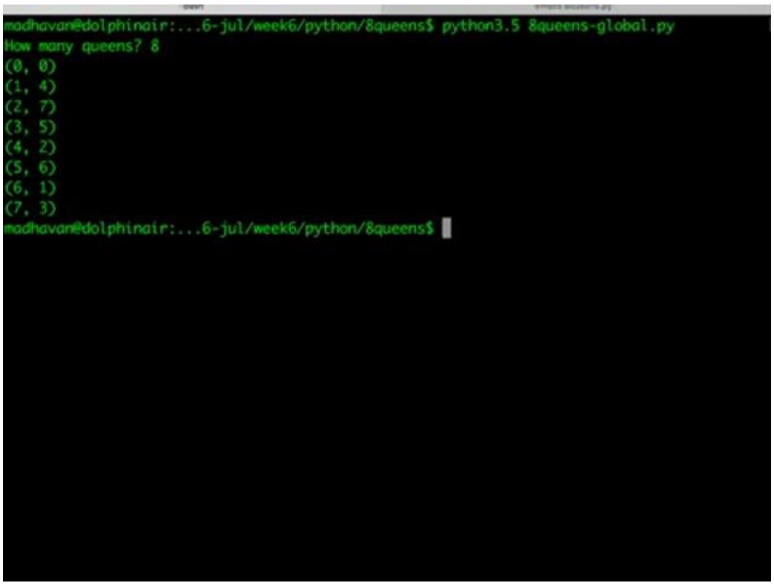
(Refer Slide Time: 03:55)




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.

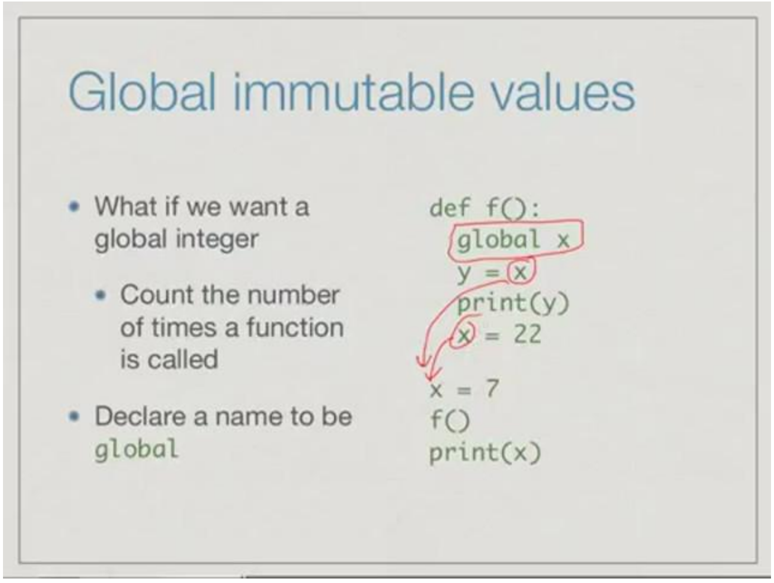
(Refer Slide Time: 04:33)




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.

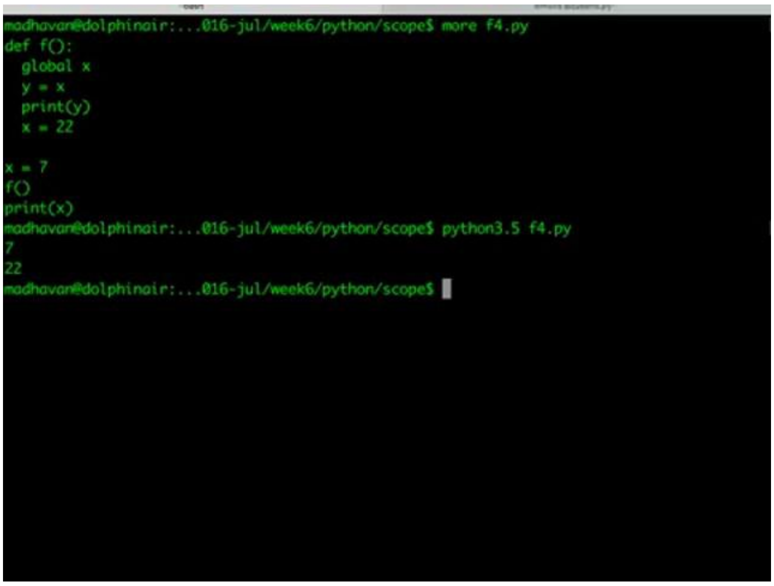
(Refer Slide Time: 05:36)




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?

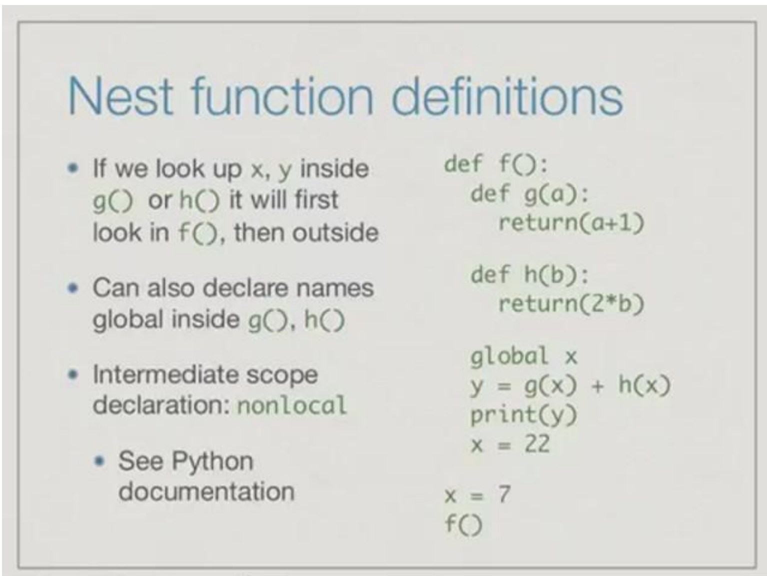
Python has a declaration called Global, which says a name is to be referred to globally and not taken as a local value. If we change our earlier definition of f, so that we add this particular tag global x then it says that this x and this x both refer to the same x outside. This is the way of telling 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)




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 except 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)



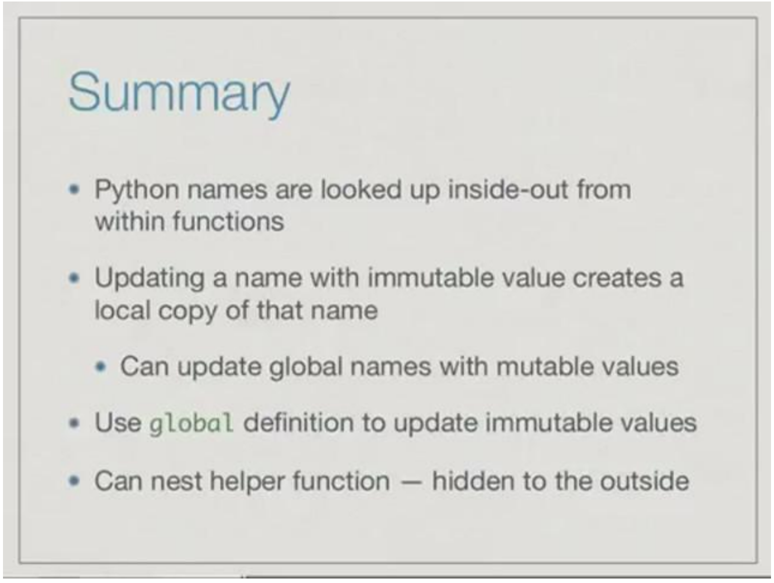
While we are on the topic of local scope, Python allows us to define functions within functions. So, here for instance the function f has defined functions g and h, g of a returns a plus 1, h of b returns two times b. Now we can update y for instance by calling g of x plus h of x rather than just setting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




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.

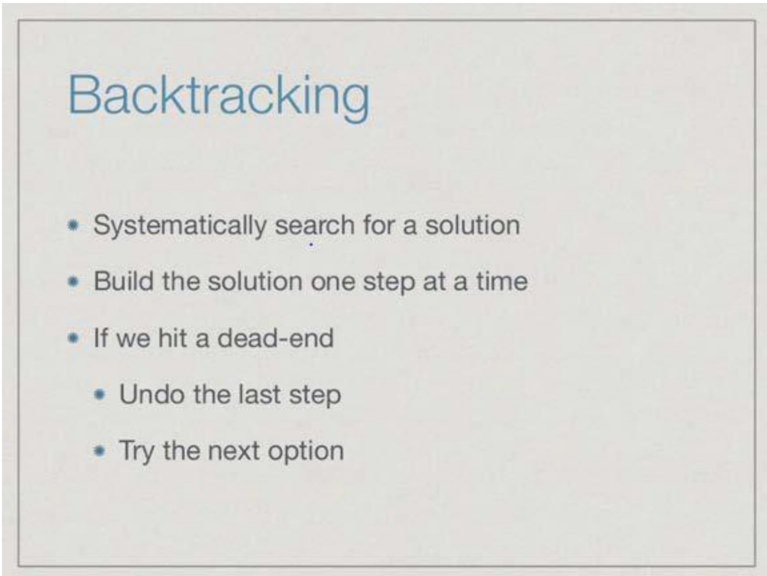
On the other hand, if you have mutable values like lists and dictionaries there is no problem. Within a function we can implicitly refer to the global one and update it. And this we saw in our 8 queens solution, we can make the board into a global value and just keep updating it within each function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

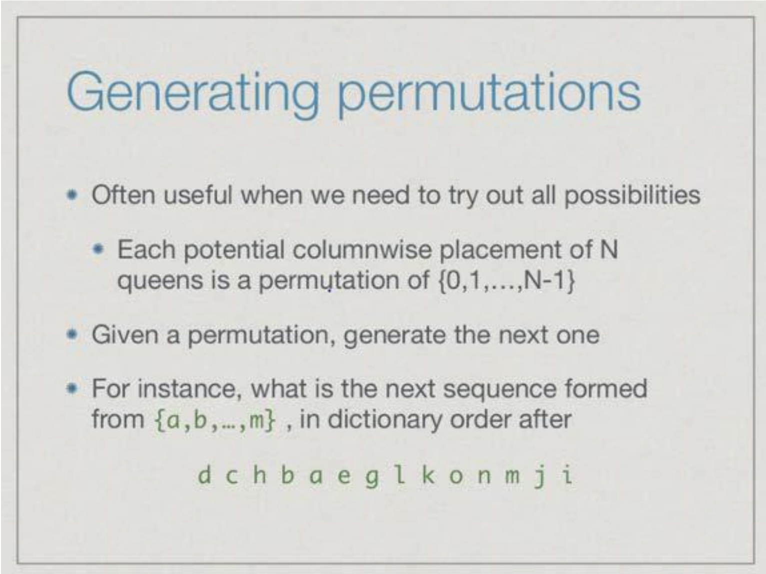
**Week - 06**
**Lecture - 03**
**Generating Permutations**

(Refer Slide Time: 00:03)




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)

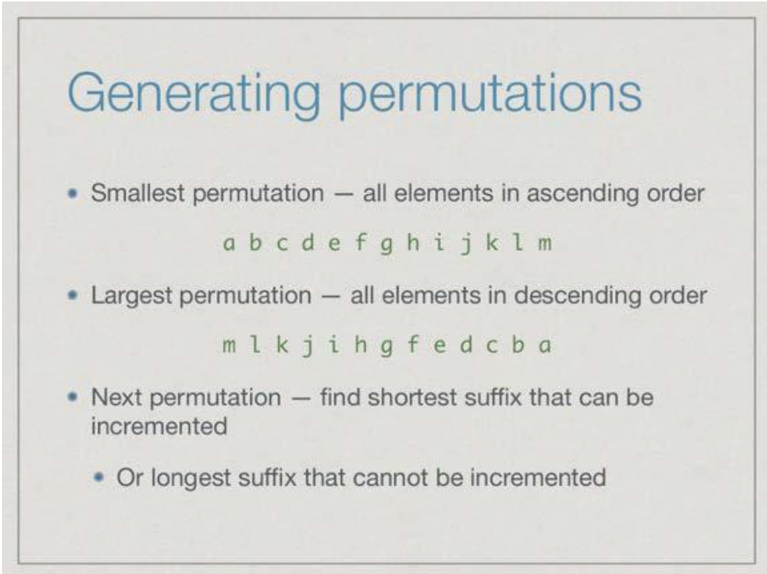



Now, in the process of doing the backtracking we need to generate all the possibilities. For instance, remember when we try to printout all the queens we ran through every possible position for every queen on every row, and if it was free then we tried it out and if the solution extended to a final case 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.

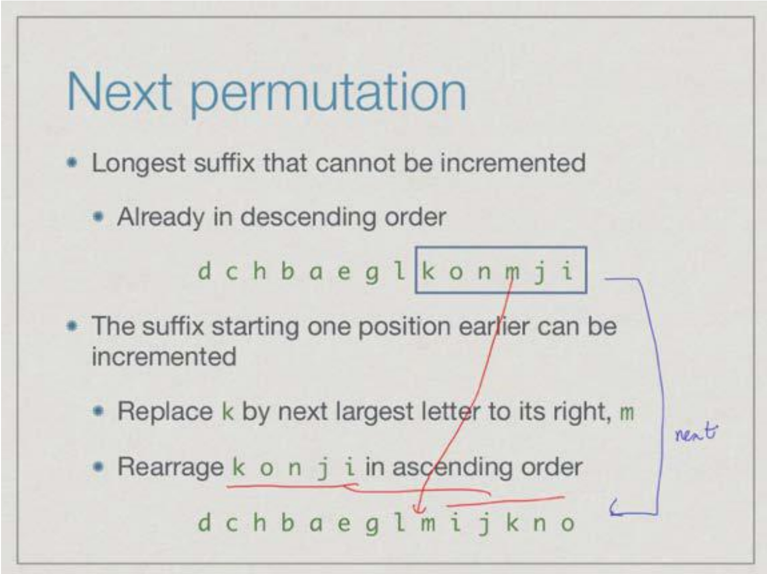
(Refer Slide Time: 01:54)




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 a 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)



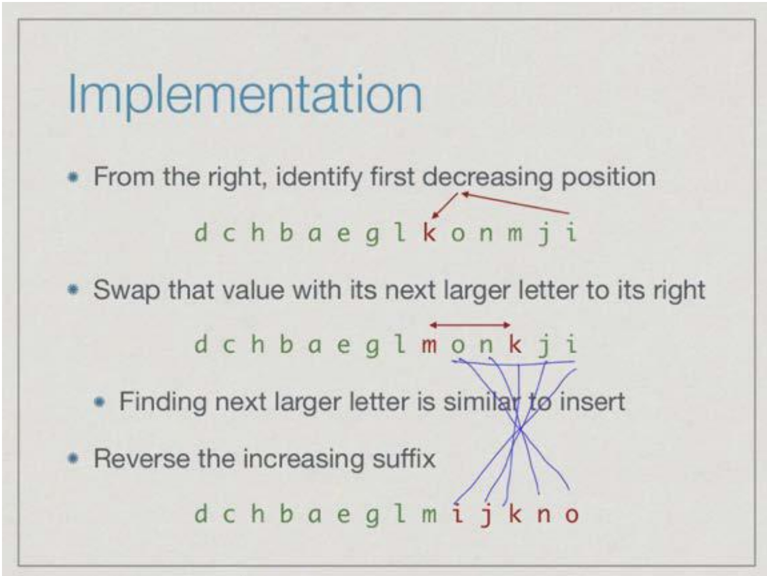

We want to find the longest suffix that cannot be incremented. So, a suffix that cannot be incremented is one which is as large as it could possibly be which means that it is already in descending order. If you look at example that we had before for which we want to define the next permutation, we find 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.

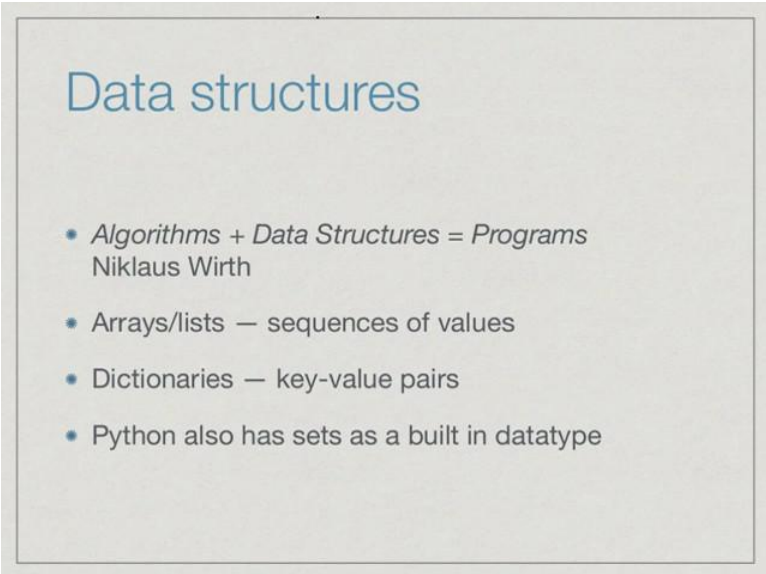
This is a concrete way in all to which find the next permutation, walk backwards from the end and see when the order stops increasing. So, wherever we first decrease this that is the suffix that you want to increment, of course if we go all the way and go back to the first letter and we are not found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

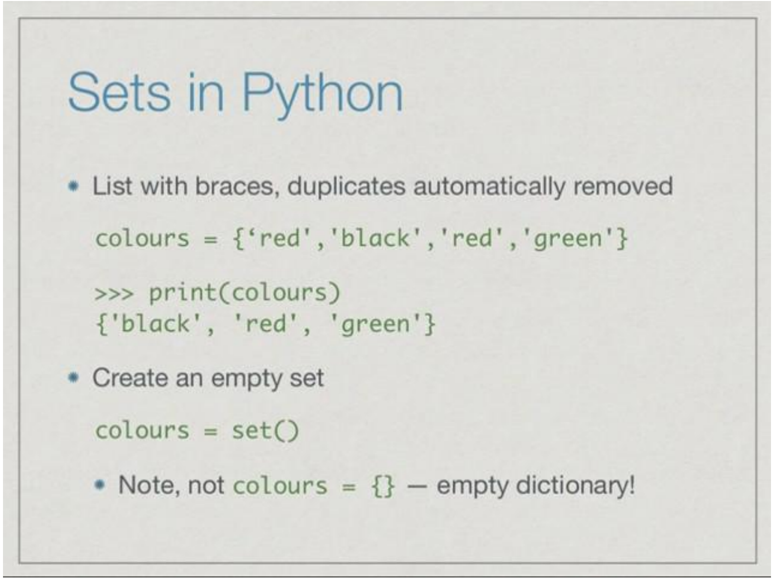
(Refer Slide Time: 00:01)




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.

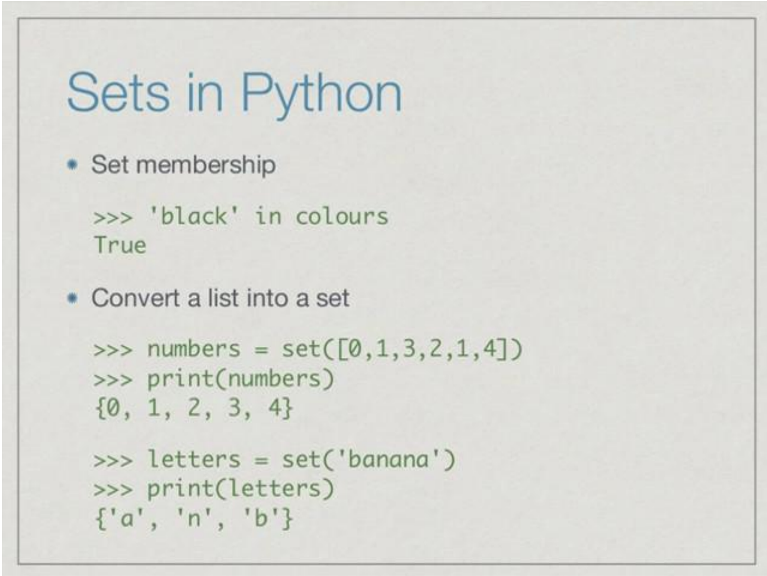
So far we have seen algorithms in some detail. So, now let us take a closer look at some specialized data structures. The data structures that we have seen that are built into python began with arrays and lists which are just sequences of values. We also saw dictionaries which are key value pairs and 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)




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)

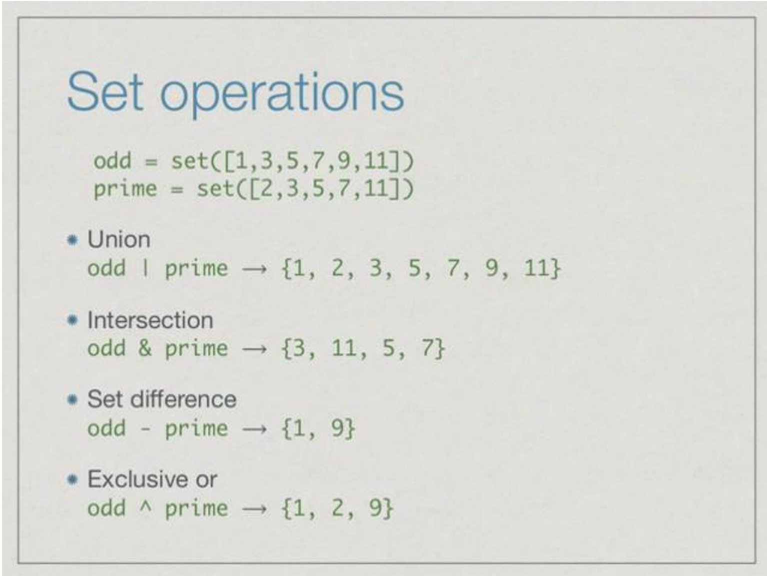



So, we say colours equal to set with no arguments. Like lists and other data structures, we can test membership using in. So, if in the previous lists set colours which had red, black and green, we ask whether black is in colours by using the word in, then, the return value is true. In general we can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



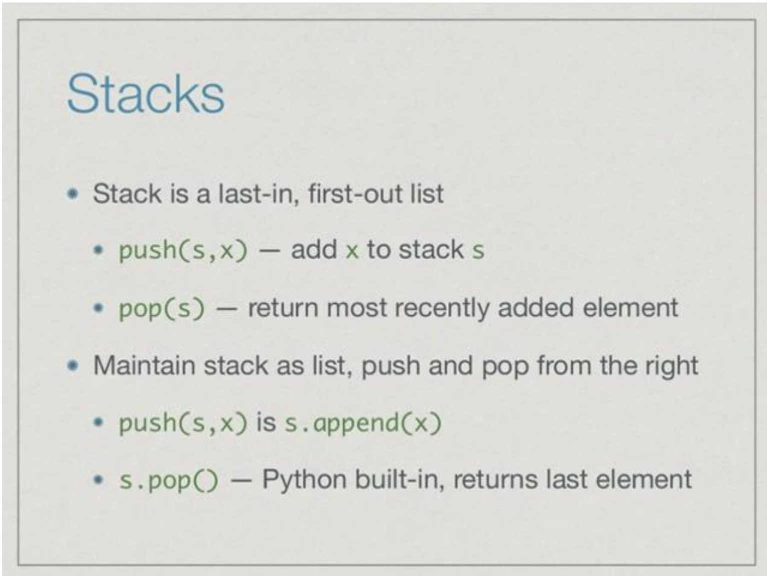

So, as you would expect sets support basic operations like their counterpart in mathematics, so suppose we set up the odd numbers to be the set of all odd numbers between 1 and 11and the prime numbers to be the set of all prime numbers from 1 and 11 between 2 and 11 using these set function as we saw 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)

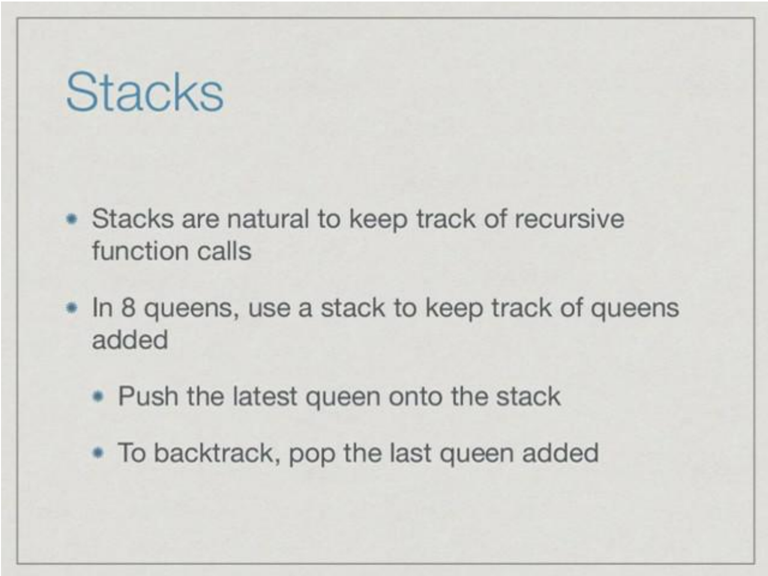



Let us look at different ways in which we can manipulate sequences. A list as we saw is a sequence in which we can freely insert and delete values all over the place. Now, if we impose some discipline on this, we get specialized data structures one of which is a stack. A stack is a last in first out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

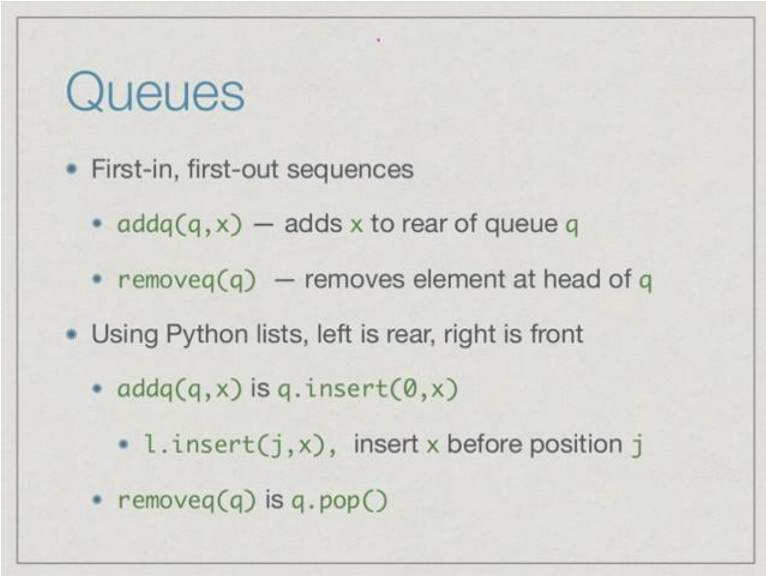
So, you can use the built-in append function that is available for lists to say s dot append x when we want to push and it turns out that python's lists actually have a built in function called pop which removes the last element and returns it to us. So, we just have to say s dot pop, where s is a list and we get exactly the behavior that we expect of our stack.

(Refer Slide Time: 05:57)




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)



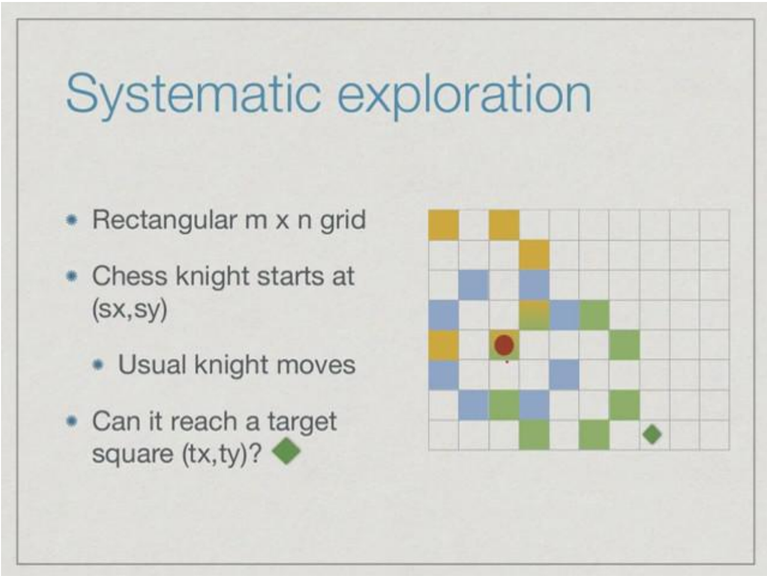

Another disciplined way of using a list is a queue. Unlike a stack which is last in first out, a queue is a first in first out sequence. In other words, we add at one end and we remove at another end. This is exactly like a queue that you see in real life, where you join the queue at the back and when 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)




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.

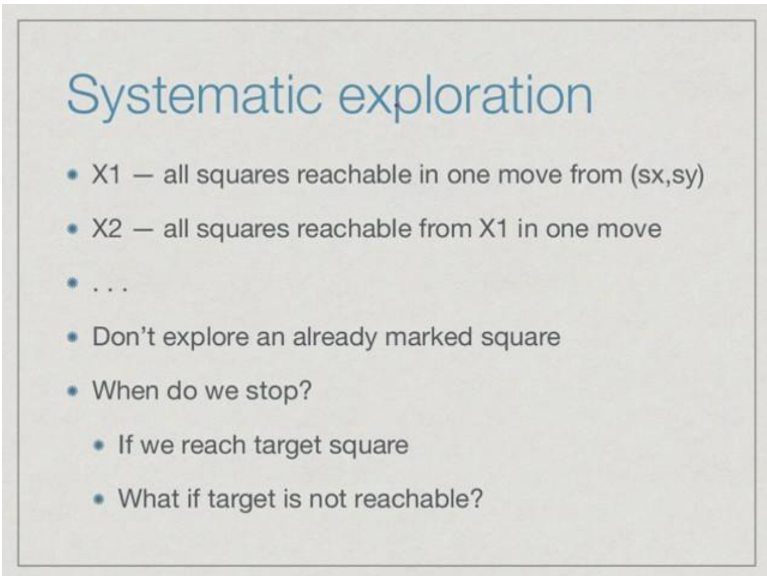
Can I hop using a sequence of knight moves from the red square to the green diamond?

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.

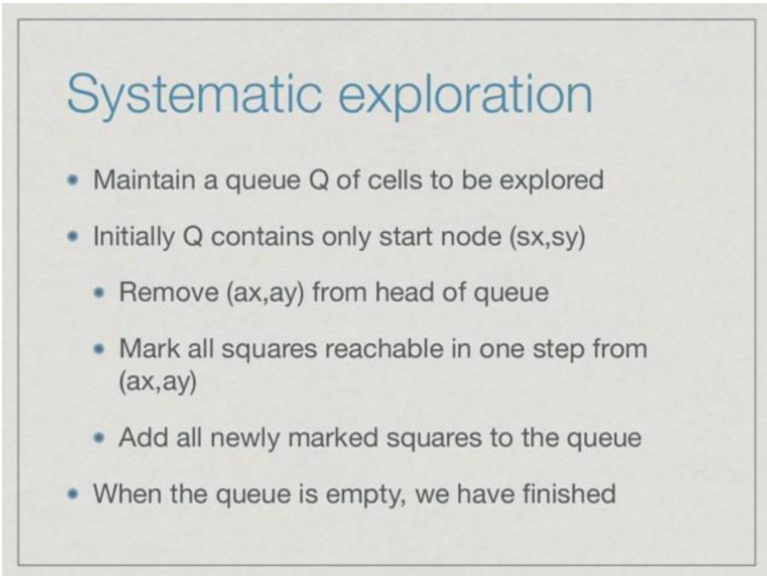
(Refer Slide Time: 10:34)



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 x2 from x 1in 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)



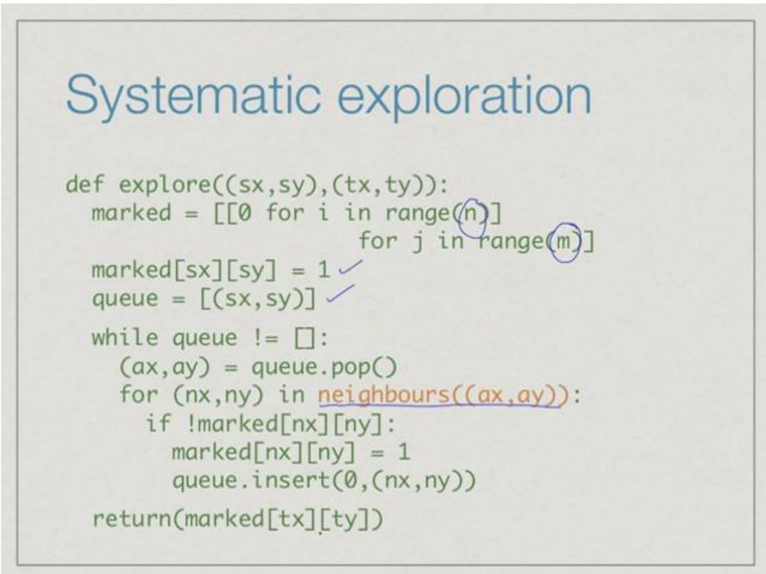

Of course since we know that we are looking for the target square, if ever we marked the target square, we can stop. On the other hand, it is possible that the target square is not reachable. In this case, we may keep going on exploring without ever realizing that we are fruitlessly going ahead and we 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)



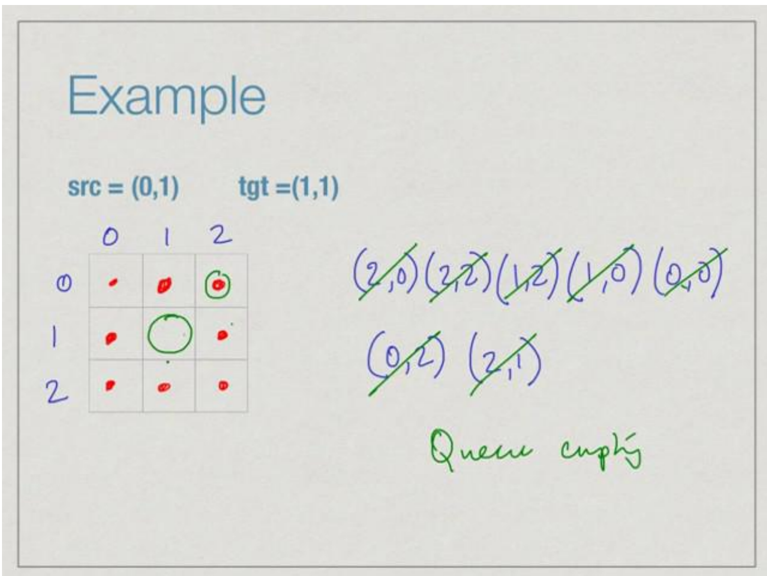

Here is some python pseudo code for this. We are going to explore from s x, s y to t x, t y. We assume that we have given to us the values m and n indicating the number of rows and columns in our grid. So, what we do is initially we set the marked array to be 0.

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, a 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)



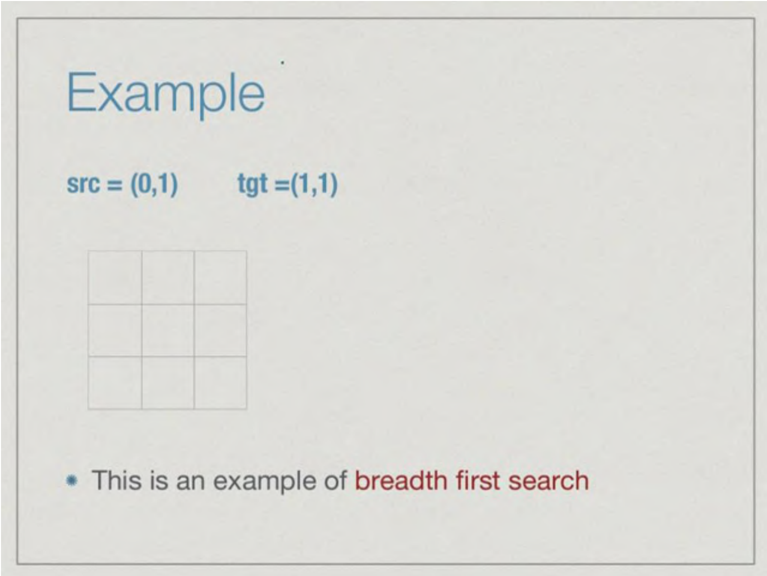
Let us look at an example of how this works. So, here we have a three by three grid. Remember that the cells are 1 0 1 2 and 0 1 2 by our naming convention we want to start from the top center square. This is our source and here in the center is our target. So, let us erase all these marks and set up 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.

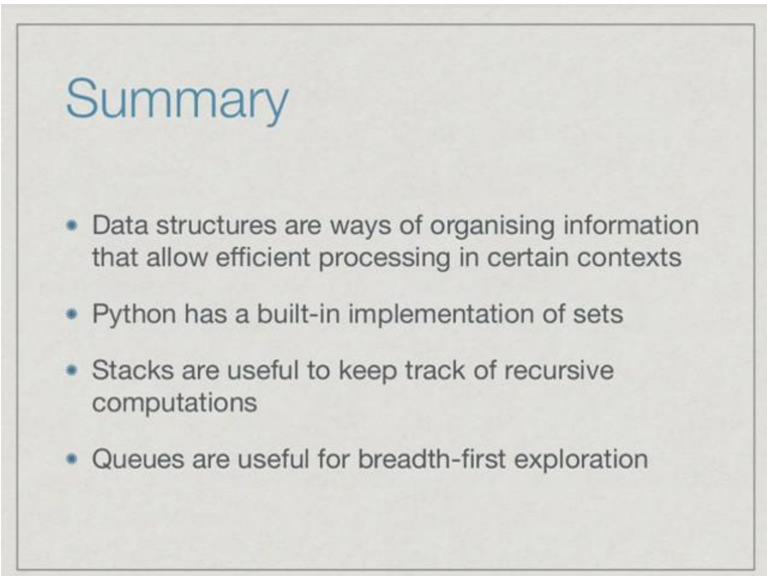
Now, when I remove 0 2 which is this one, I find that both these neighbors I explored. So, I add nothing. I continue with 2 1. Again, I find both its neighbors explored and do nothing. Now at this point, the queue is empty and since the queue is empty, I stop and I find that my square of interest namely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

(Refer Slide Time: 17:35)




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)

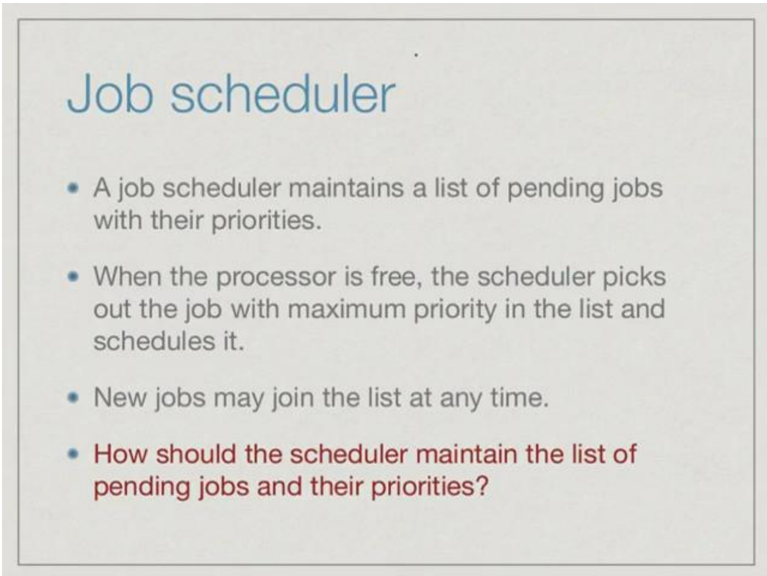



To summarize data structures are ways of organizing information that allow efficient processing in certain contexts. So, we saw that python has a built-in implementation of lists of sets rather we also saw that we can take sequences and use them in two structured ways. So, stack is a last-in first-out 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**

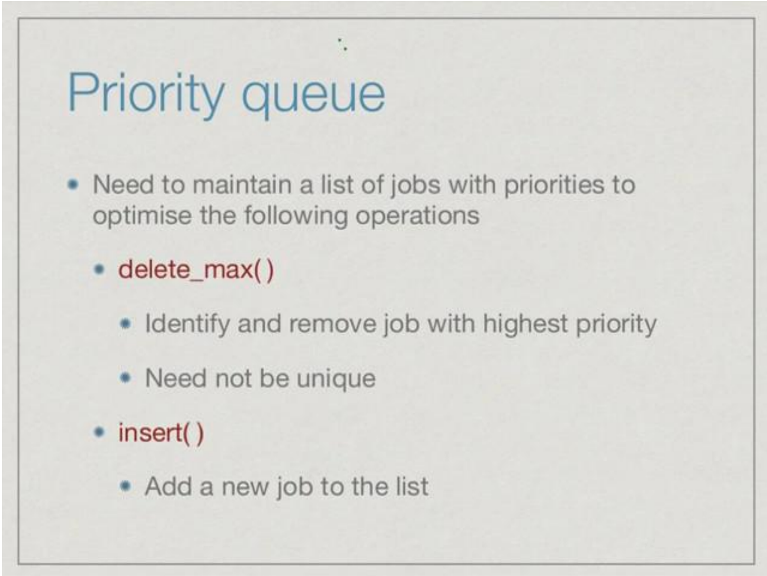
(Refer Slide Time: 00:02)




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.

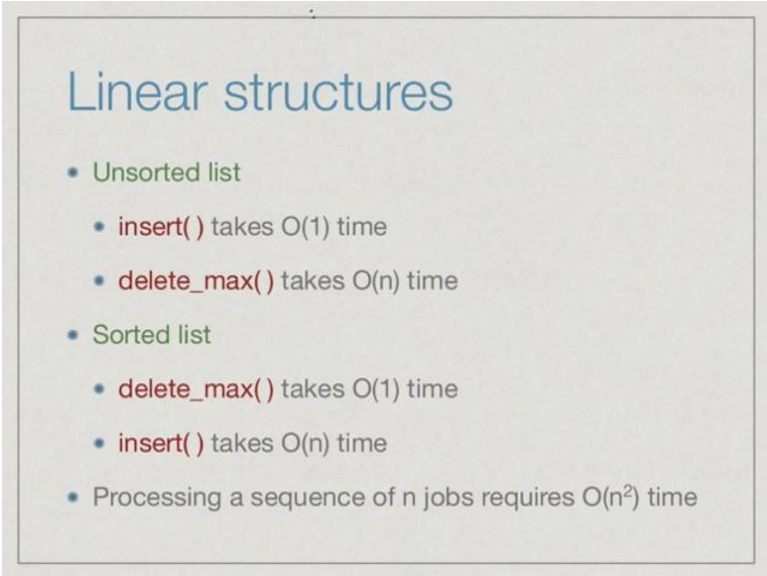
(Refer Slide Time: 00:47)




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 it is own priority.

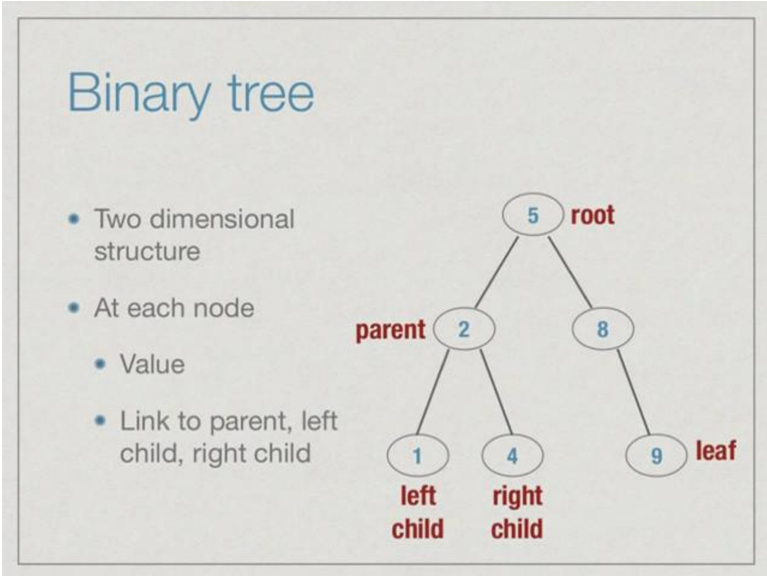
(Refer Slide Time: 01:39)




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.

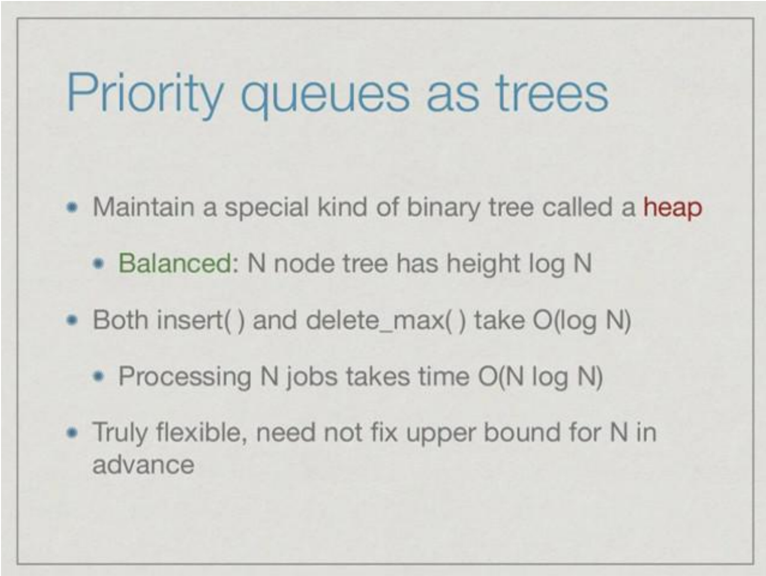
(Refer Slide Time: 02:55)




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.

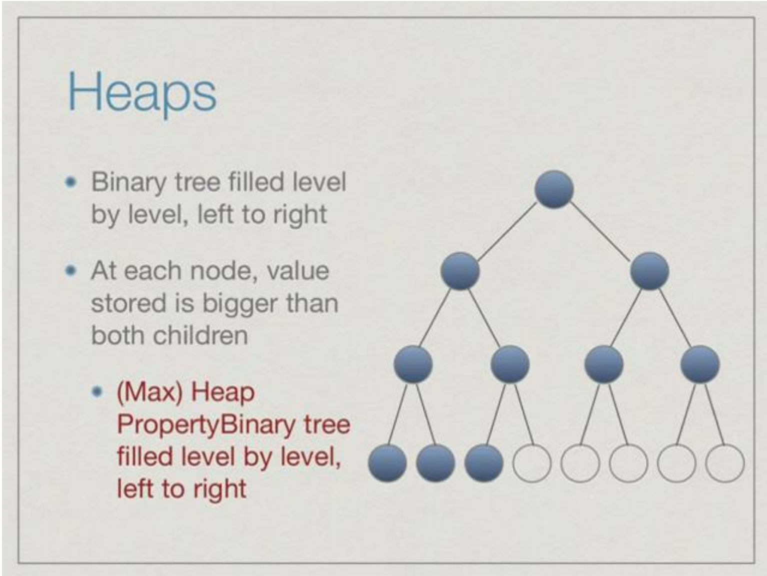
(Refer Slide Time: 03:40)




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)

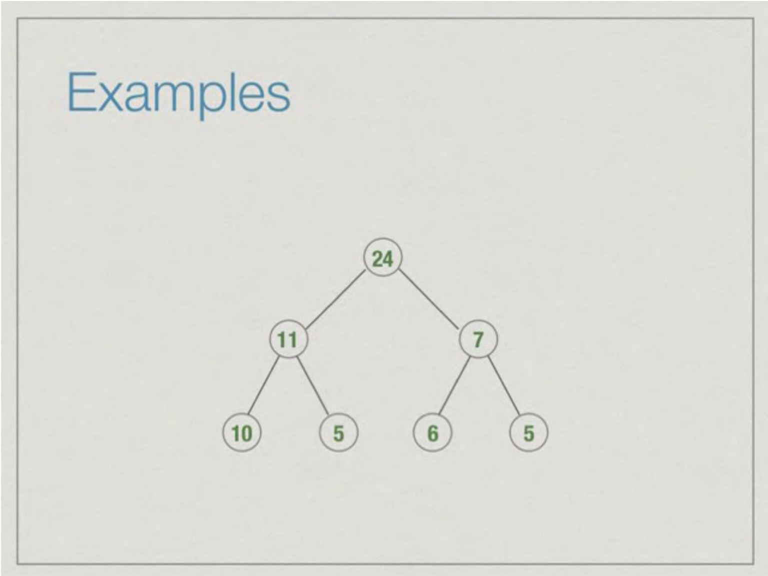



What is a heap? Heap is a binary tree with two properties, the first property is structural: which are the values which are stored in the tree, remember that leaves, nodes in a binary tree may have 0 children, 1 children or 2 children. So, we could have in general a very uneven structure. Heaps have 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.

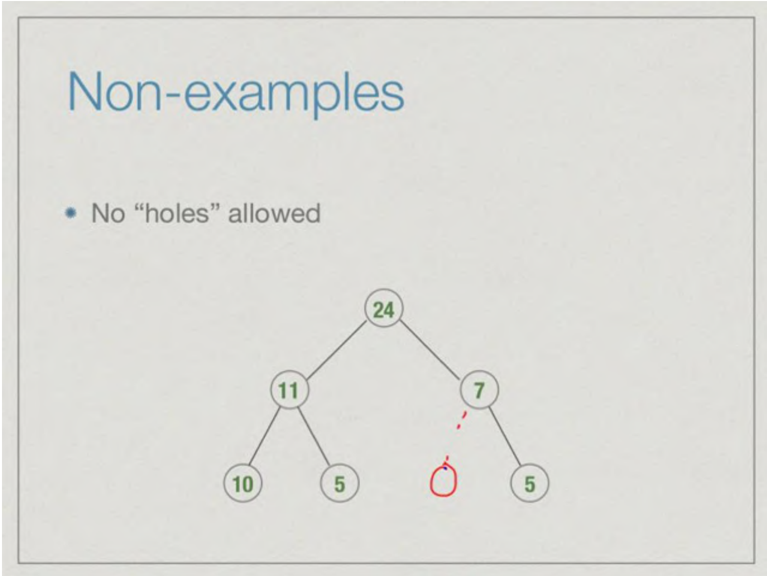
(Refer Slide Time: 06:06)




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.

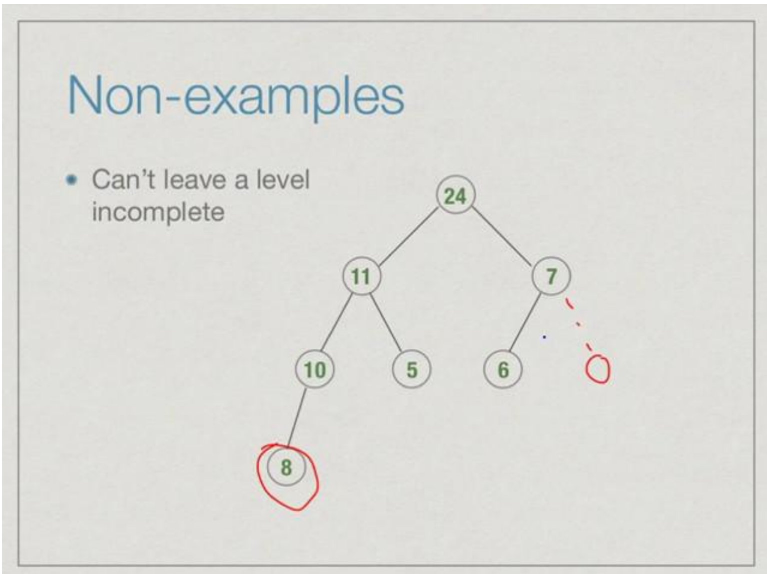
Here is another example where the bottom level is actually full, here we have 7 nodes and once again at every point we see that 11 is bigger than 10 and 5, 7 is bigger than 6 and 5. So, we have the value property - the max heap property along with the structural property.

(Refer Slide Time: 06:42)



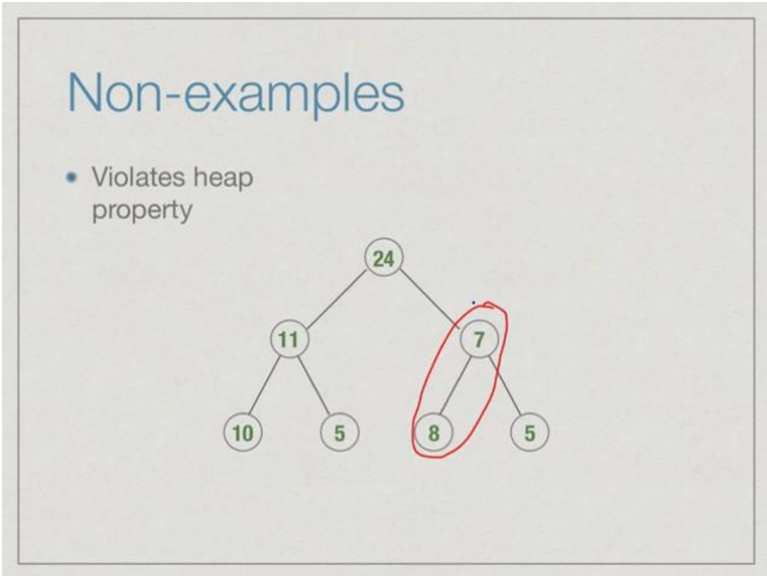

Here is an example of something which is structurally not a heap because it is a heap we should have it filled from top to bottom, left to right. So, we should have a node here to the left of 7 before we add a right child therefore, this is not a heap.

(Refer Slide Time: 06:59)



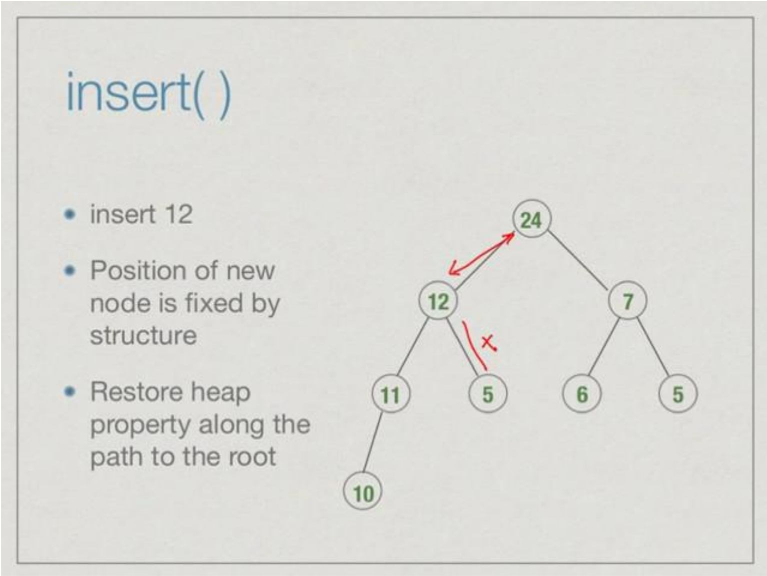

Similarly, here the node 8 could not have been added here before we filled in the right child of 7. So, once again this has not been filled correctly left to right, top to bottom and therefore, this is not a heap.

(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)

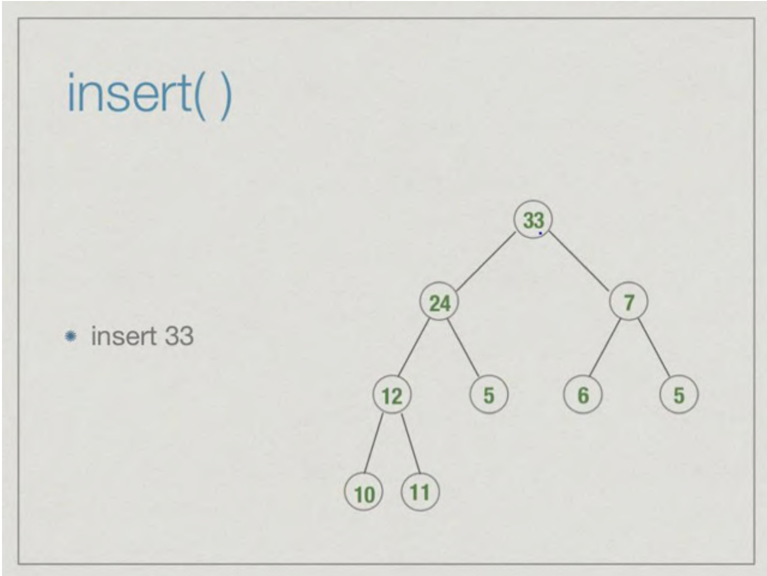



Our first job is to insert a value into a heap while maintaining the heap property. So, the first thing to note is that we have no choice about where to put the new node, remember that heap nodes are constructed top to bottom, left to right. If we want to insert 12 it must come below the 10 to the left 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 it is 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 it's current parent. So, we look and we find that it is not. So, again we exchange these.

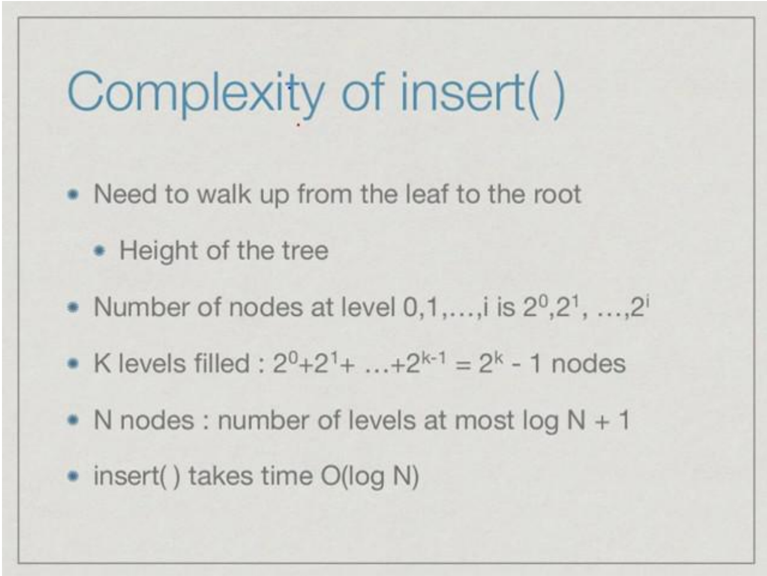
Now, notice that because 11 was already bigger than 5, 12 will remain bigger than 5. There is no need to check anything down from where we got, we only have to look up. Now, we have to check whether there is still a problem above. In this case, there is no problem 12 is smaller than 24. So, we stop.

(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.

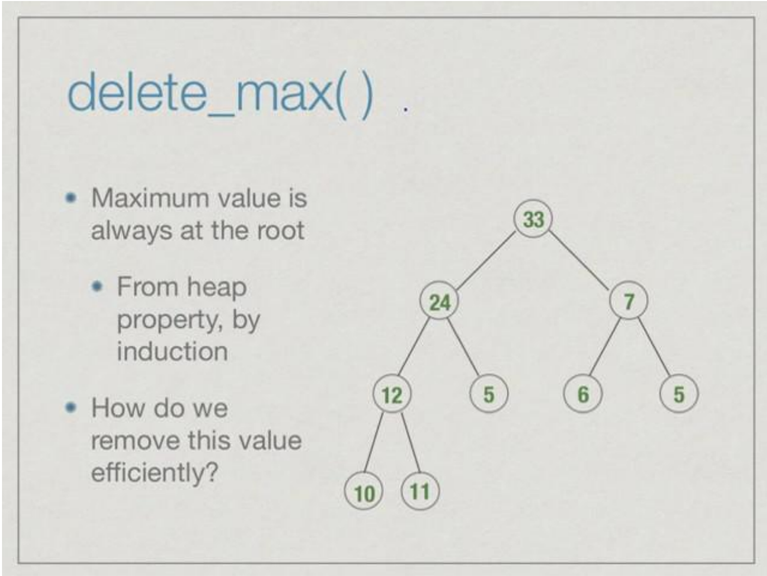
(Refer Slide Time: 09:35)




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.

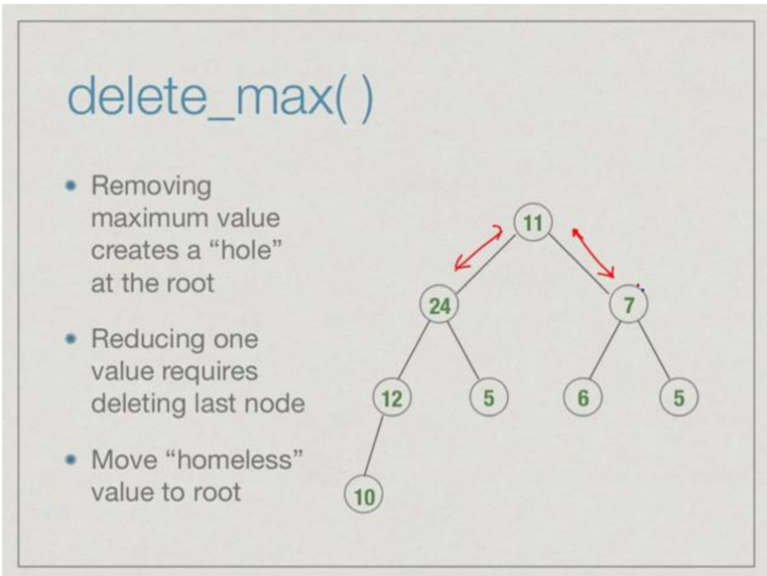
Now, we argued before or we mentioned before that a balanced tree will have height log n. So, we can actually measure it correctly by saying that the number of nodes at level i is 2 to the i. Initially, we have 1 node 2 to the 0, then at the first level we have 2 nodes 2 to the 1 and second level we 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)




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 it's 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)

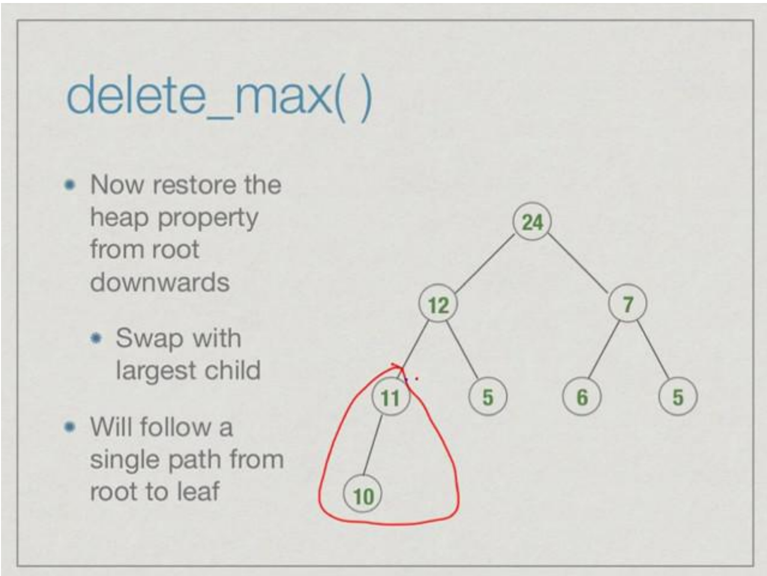


If we remove this node, first of all we cannot remove the node because it is a root. If you remove this value then we have to put some value there. On the other hand, the number of values in the node in the heap has now shrunk. So, this node at the bottom right must be deleted because the structural 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 it's 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)

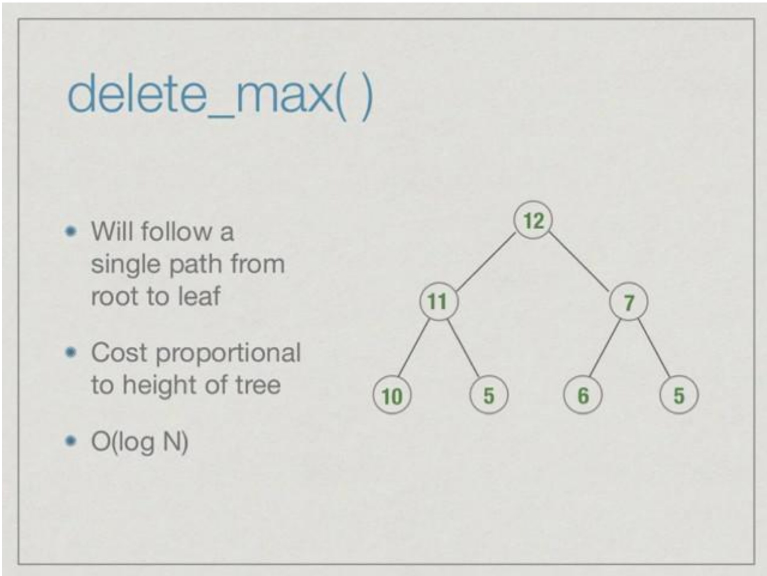



To restore the property what we do is, we look at both directions right and we exchange it with the largest child. Suppose, this had been 17 here then we could swapped 11 with

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 respective 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.

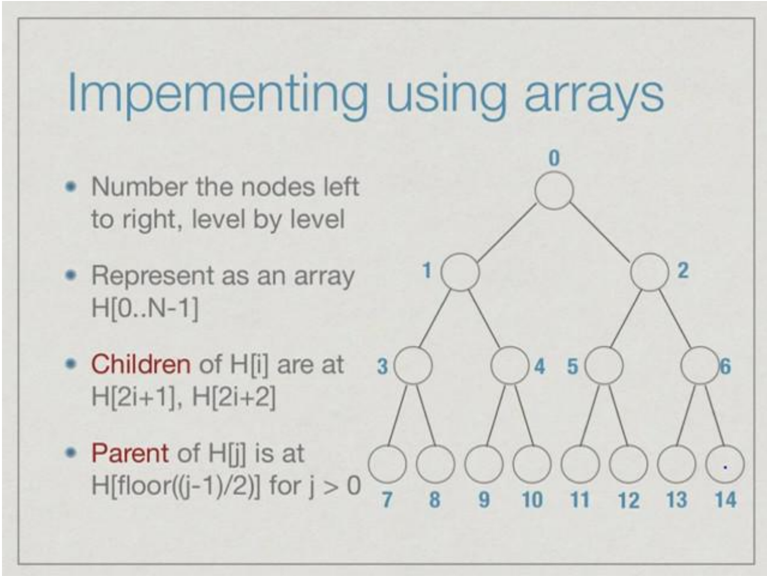
(Refer Slide Time: 13:03)




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)

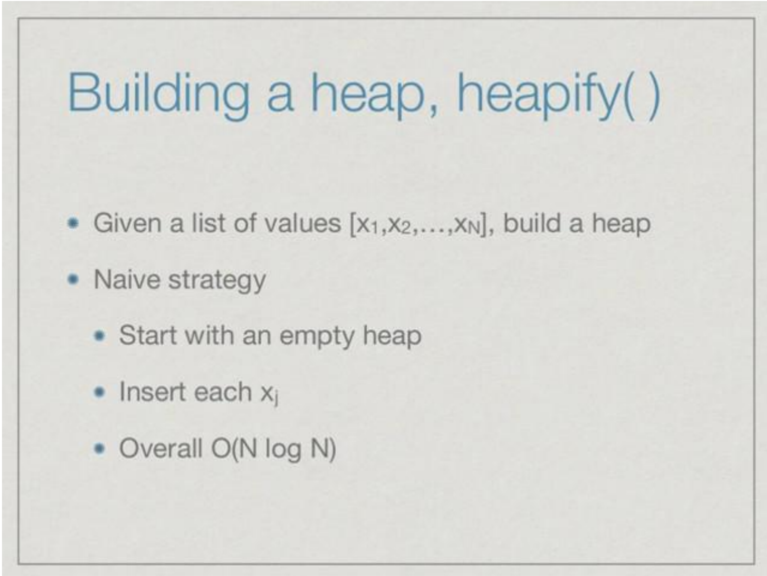



One very attractive feature of heaps is that we can implement this tree directly in a list or in an array. So, we have an n node heap, we can represent it as a list or an array with position 0 to n minus 1. The position 0 represents a root then in order 1 and 2 represent the children, then 3, 4, 5, 6, 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 2 i plus 1 and 2 i plus 2 right. 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 it's children are and by reversing this calculation we can also find the index of the parent, the parent of j is that j minus 1 by 2. Now, j minus 1 by 2 may not be an integer. So, we take the floor.

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

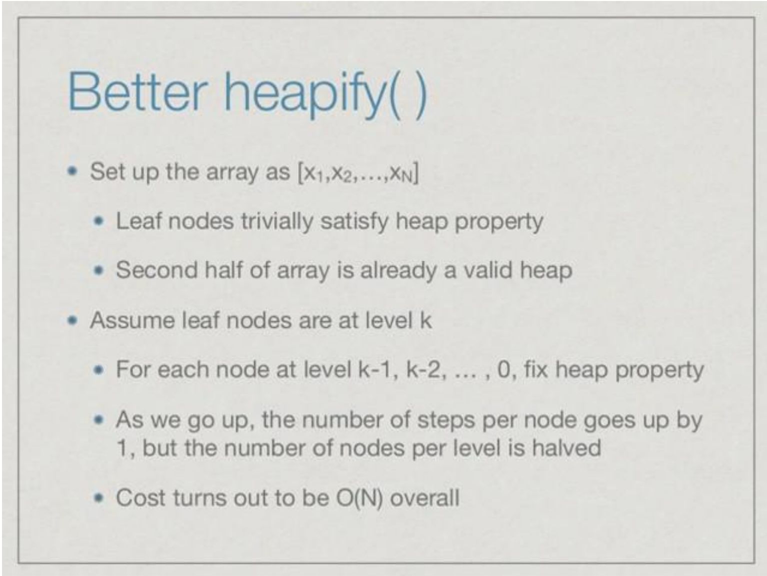
(Refer Slide Time: 15:32)




This allows us to manipulate parent and children nodes by just doing index arithmetic we go from i to 2 i plus 1, 2 i plus 2 to go to the children and we go from j to j minus 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 x2, 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.

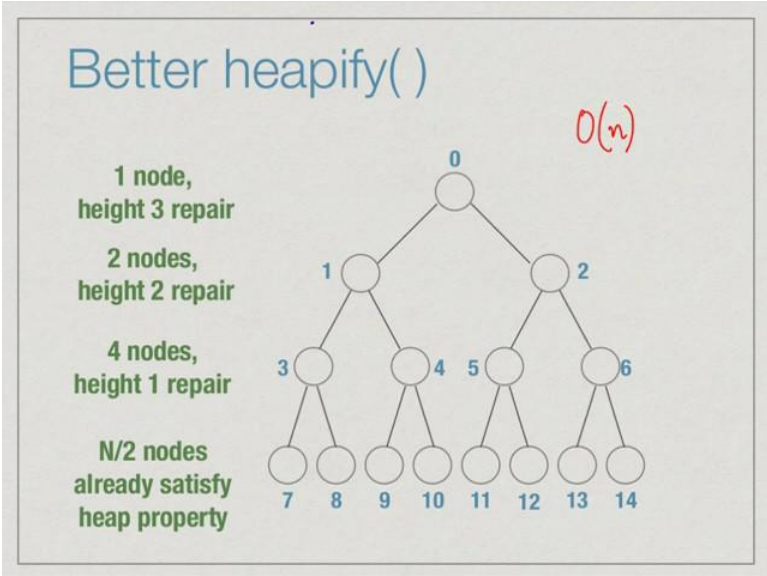
(Refer Slide Time: 16:23)




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.

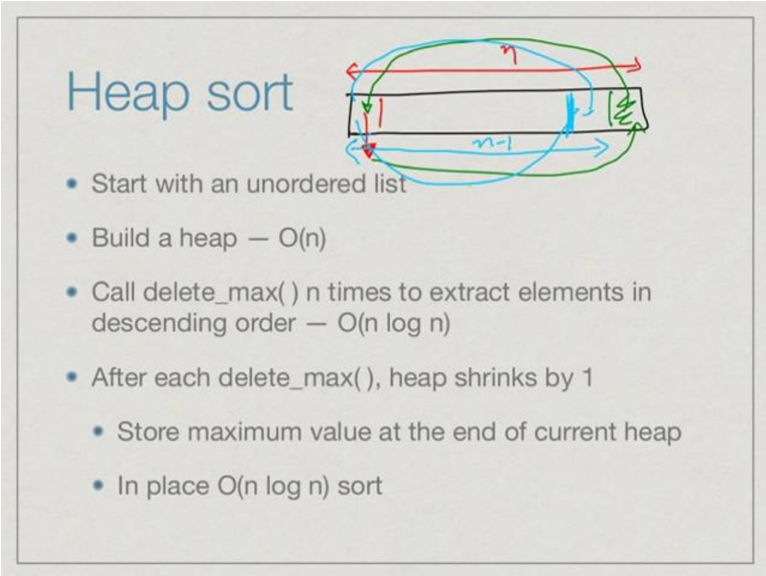
(Refer Slide Time: 17:19)




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 it's children, we may have to swap this with it's 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)



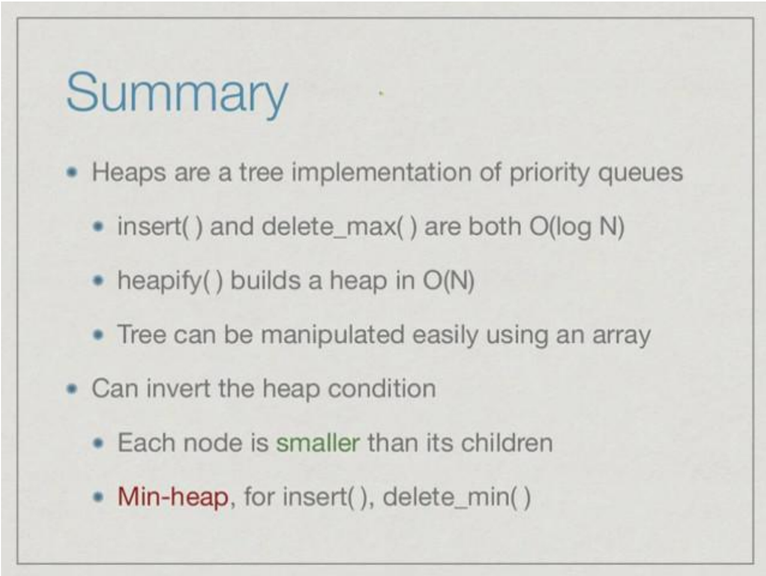

A final use of heap is to actually sort, we are taking out one element at a time starting with maximum one. It is natural that if we start with a list, build a heap and then do n times delete max we will get the list of values in descending order. We build a heap in order n time, call delete max n times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



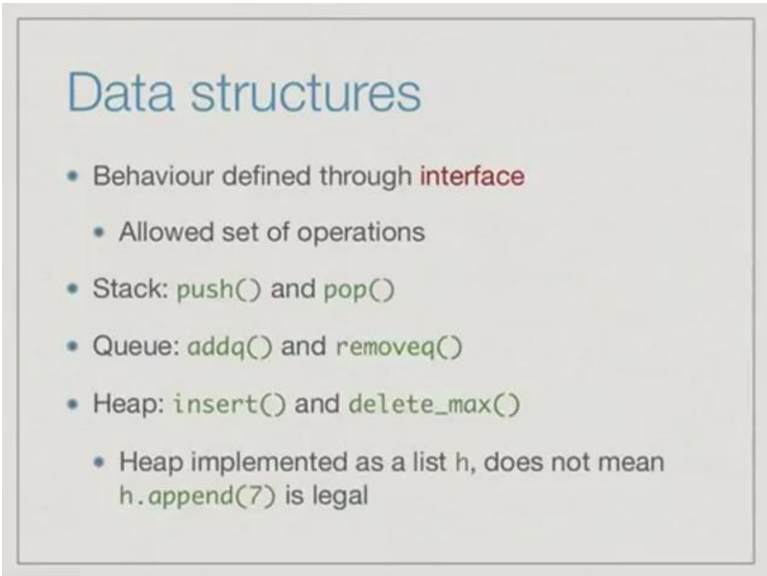

To summarize heaps are a tree based implementation of priority queues in which both insert and delete max can be done in log n time. We can do a bottom up heapify to build a heap in order n time and these are trees, but they can be manipulated very easily using an array. Now, in this case we were looking 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 order n log 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)



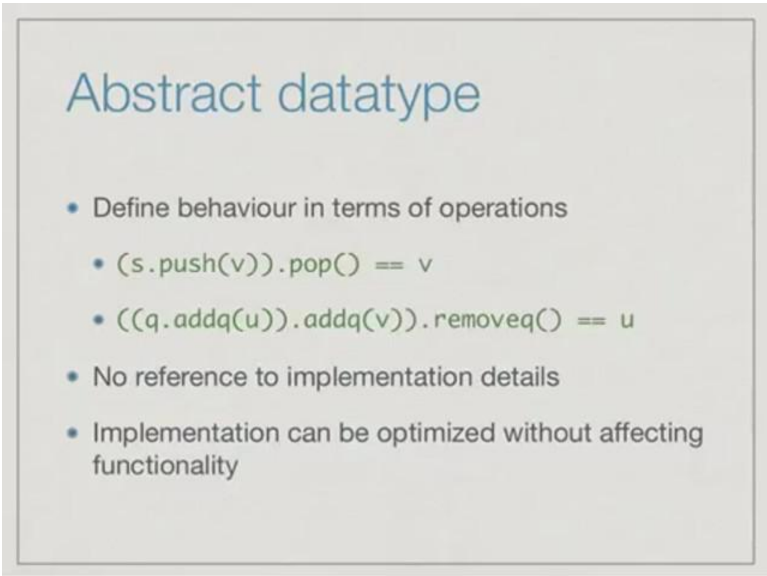

We have seen how to implement data structures such as, stacks, queues and heaps using the built in list type of Python. It turns out that one can go beyond the built in types in python and create our own data types. So, we will look at this in more detail in this weeks' lectures.

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 it is 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)




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.

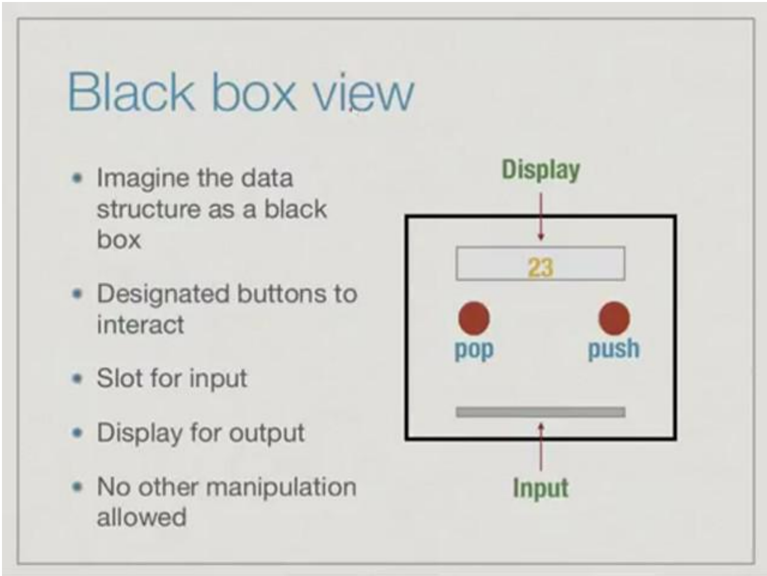
The important thing is that we would like to define the behavior of a data type without reference to the implementation. Now this can be very tedious because you have to make sure that it capture all the properties between functions, but this can be done and this is technically how an abstract data type is defined. Now 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)

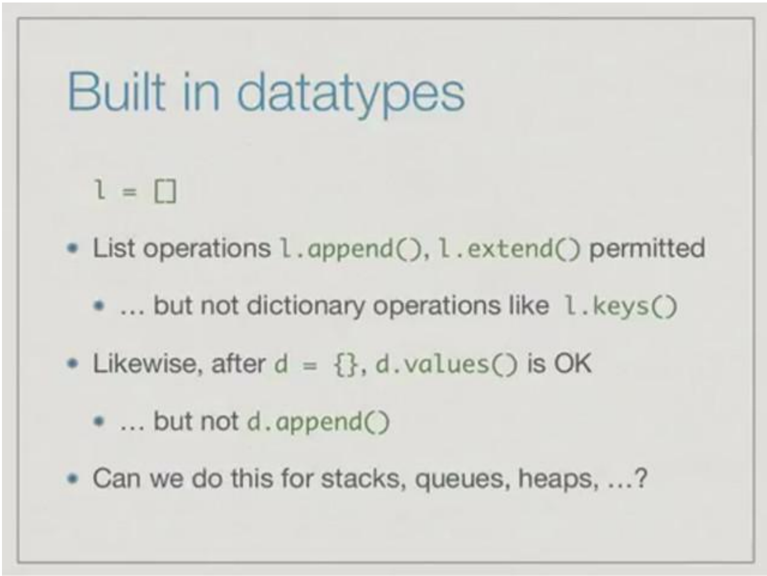



So, good way to think of an abstract datatype is as a black box which allows limited interaction. Imagine something like an ATM machine. So, we have the data structure as a black box and we have certain buttons which are the public interface, these are the functions that we are allowed to use. In this 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 if 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)



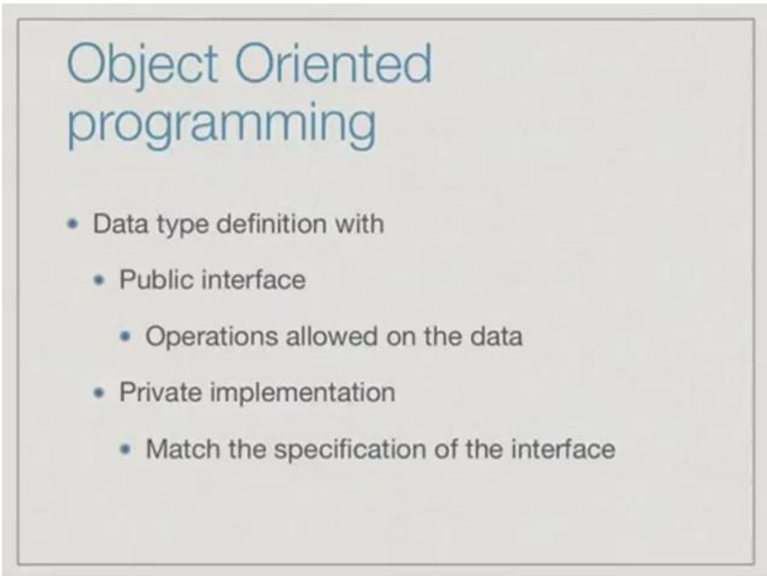

In a sense this is already implemented when we use the built in data types of python, if we announce that the name l is of type list by setting l to the empty list then immediately python will allow us to use operations like append and extend on this list, but because it is of list type and not dictionary 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 it is 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)

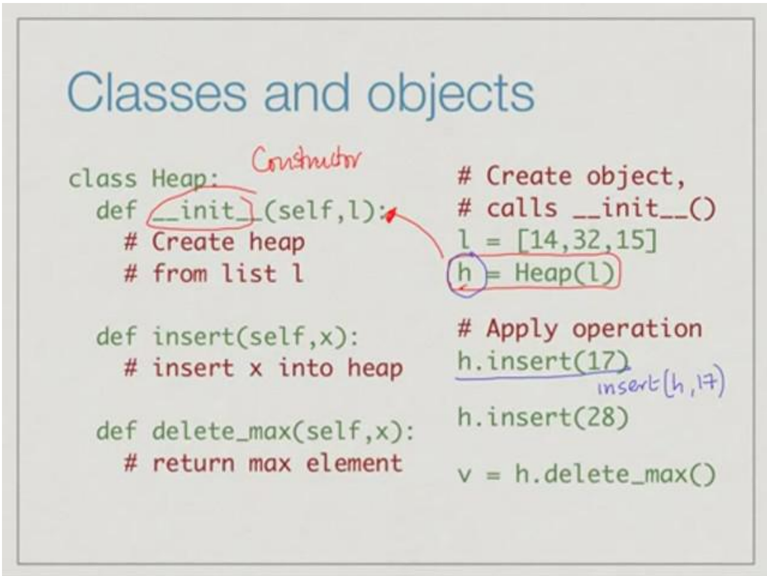



This is one of the main things which are associated with a style of programming called Object Oriented program. In object oriented program, we can provide data type definitions for new data types in which we do precisely what we have been talking about we describe the public interface, that is the operations 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)




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 execute 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 it is 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 it is 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.

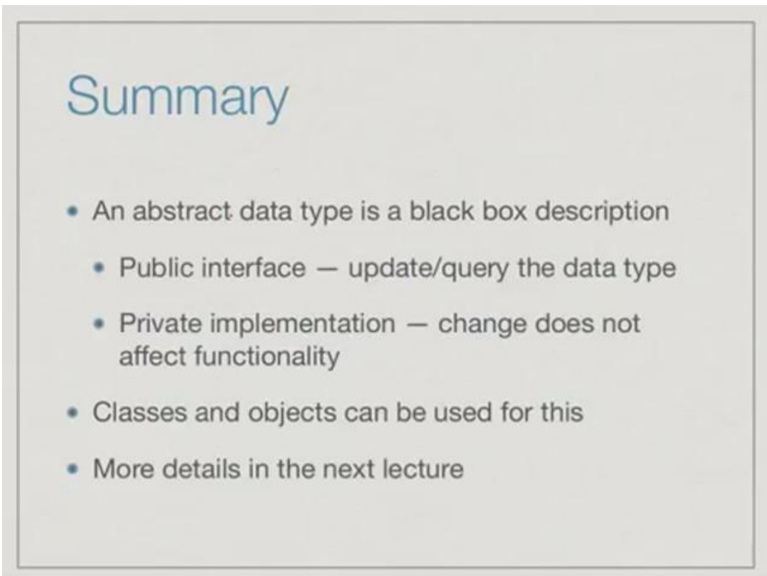
When we create on object of type heap we call it like a function. So, we say h is equal to heap l, so this implicitly says give me an instance of the class heap with the initially value l passed to it now this calls this function init which is why it is called init. So, init is what is called a constructor. 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 attach to this function call or this class were 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)



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.

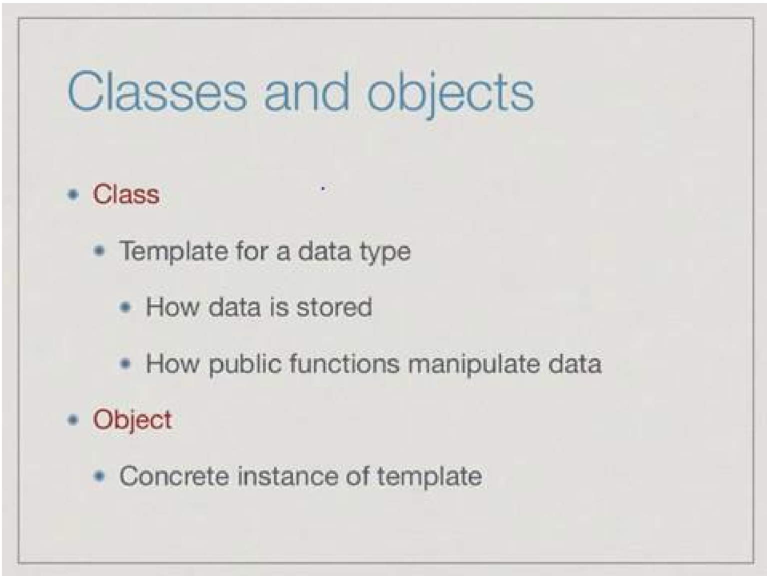
They may behave differently in terms of efficiency, you might see that one version is faster than another or one version slower than another, but this is not the same as saying that the functions change. So, we do not want the values to change, if we have a priority queue and we insert a set of values 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**

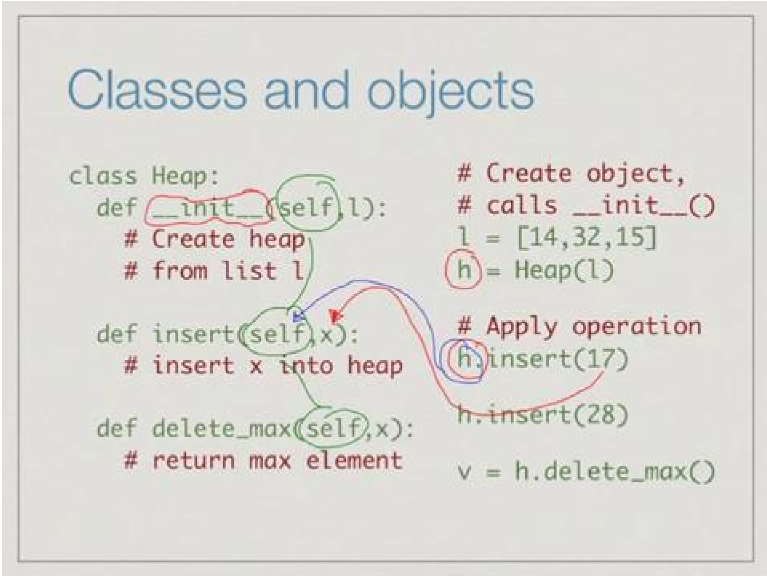
**Week - 07**
**Lecture - 02**
**Classes and Objects in Python**

(Refer Slide Time: 00:02)




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)



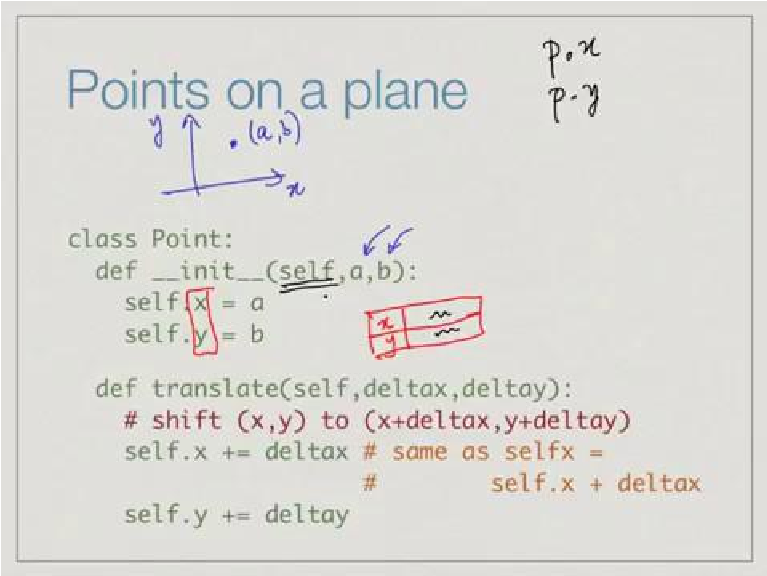

We saw a skeleton implementation of a heap. This had a special function called init, which was used to create the heap, and then we had functions insert and delete. Now one thing which we did not explain is this argument self that run through this. This is a convention which we have in python that every 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 heap 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)



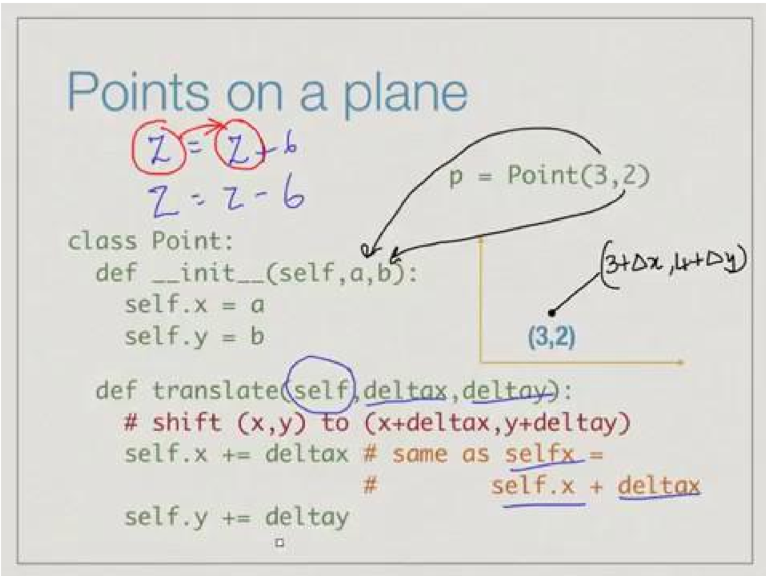

Our first example is just a representation of a point x y. So, we are just thinking about a normal coordinate system, where we have the x-axis, the y-axis. Therefore, a given point is given some coordinate like a comma b. This is a point with x-coordinate a and y-coordinate b, this is a familiar concept 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)

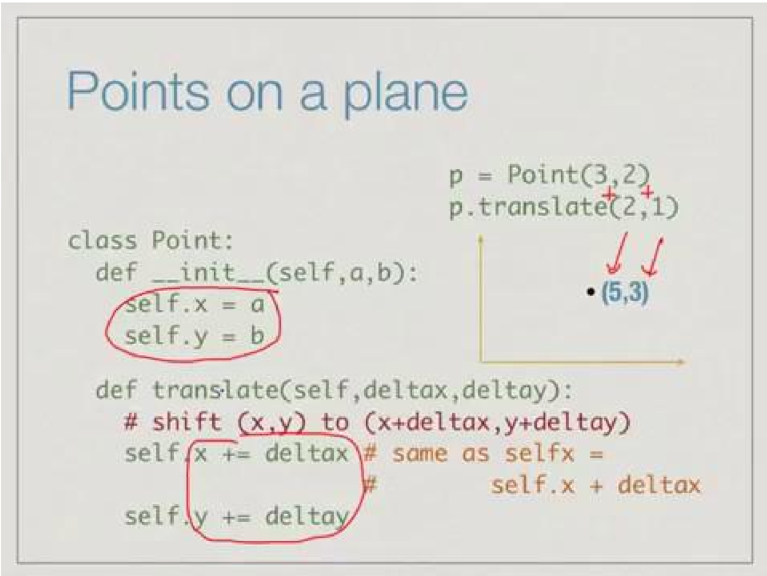



Let us look at how this works. For instance, if we say p is equal to point 3, 2; then 3 will be passed as a, and 2 will be passed as b. And this will set up a point that we have drawn here, which internally is represented as self dot x is 3 and self dot y is 2. Now here is a slightly different function, 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.

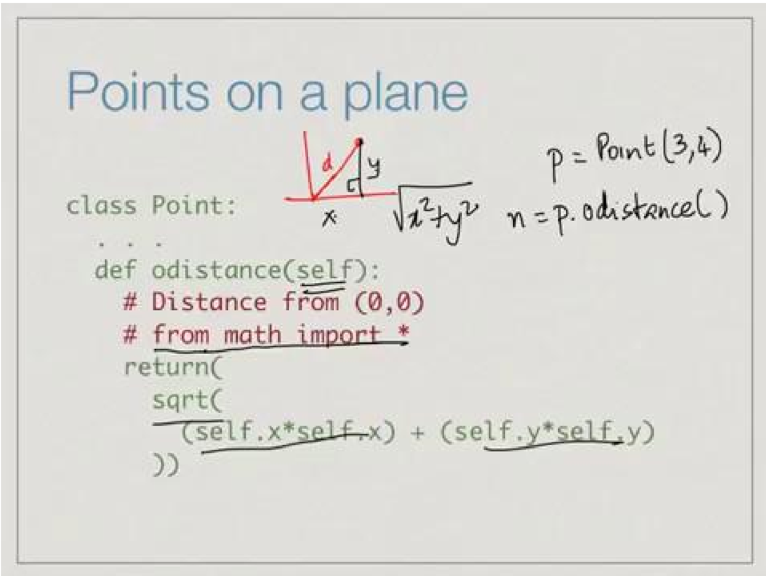
(Refer Slide Time: 06:37)




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)

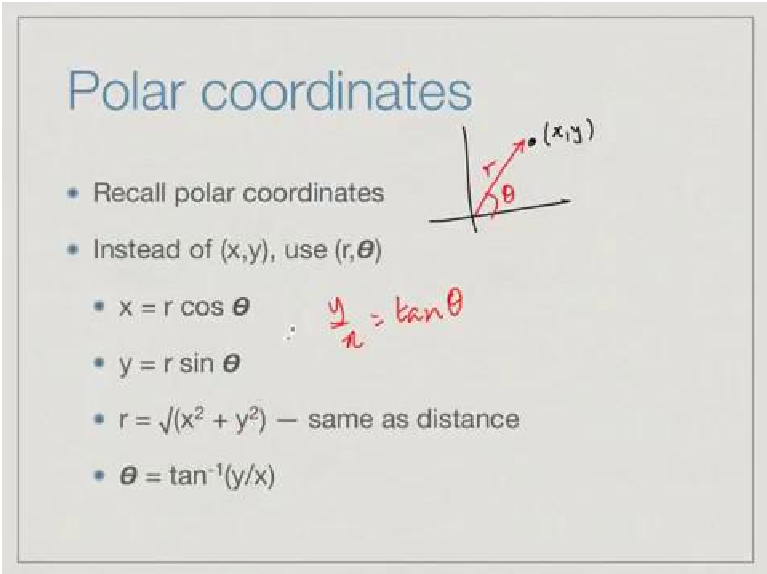



Let us look at different functions. So supposing we want to compute the distance of a point from the origin. So, we want to know what is this distance. This distance by Pythagoras' theorem is nothing but the square root of x square plus y square. So, remember this is like a hypotenuse of a right angled 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.

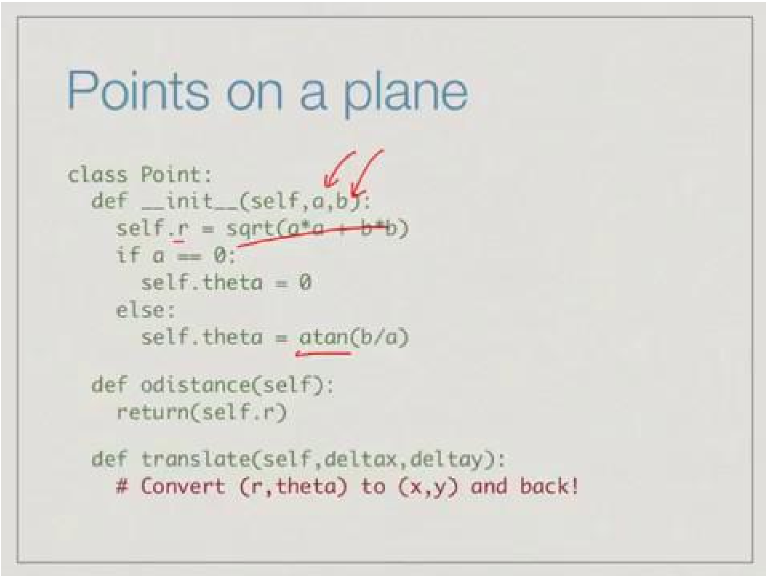
(Refer Slide Time: 09:01)




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)




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.

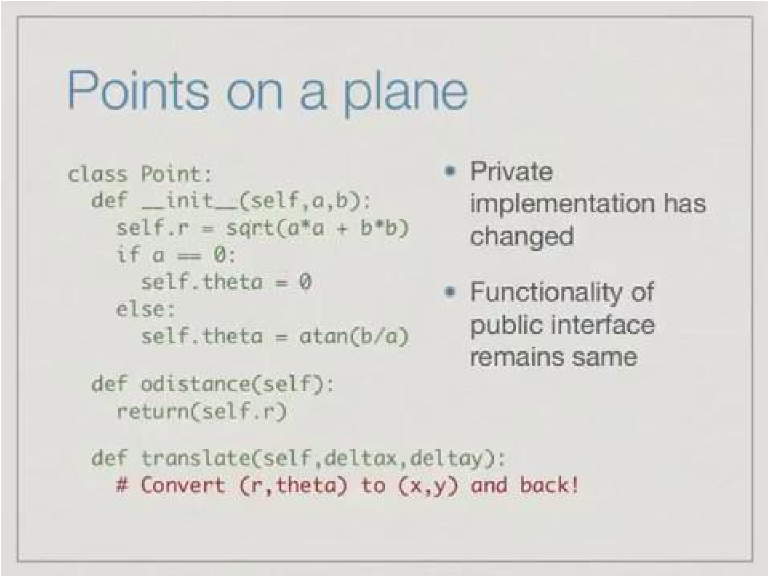
Now internally we are now keeping self dot r and self dot theta. We are not keeping self dot x and self dot y. This is useful because if you want the o distance - the origin distance we just have to return the r value we do not do any computation. So, in other words if we are going to use o distance 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.

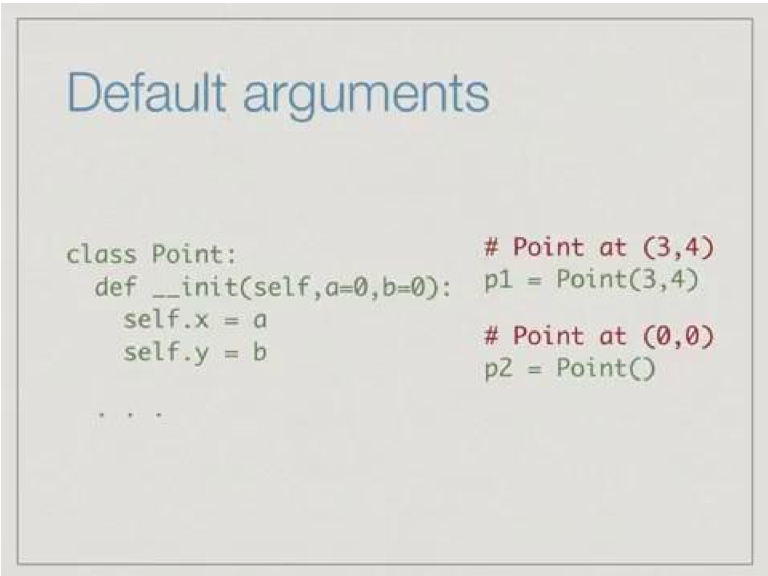
If you think translate happens more often it's probably better to use x and y; if you think origin from the distance is more important, so probably better to use r and theta. So, often there is no one good answer. It is not like saying that a heap implementation is always better than a sorted list implementation 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)




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.

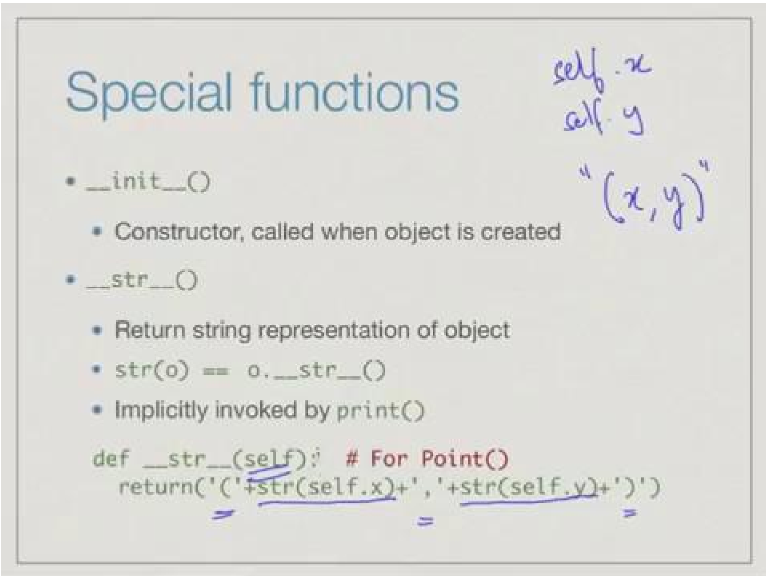
(Refer Slide Time: 13:48)



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)



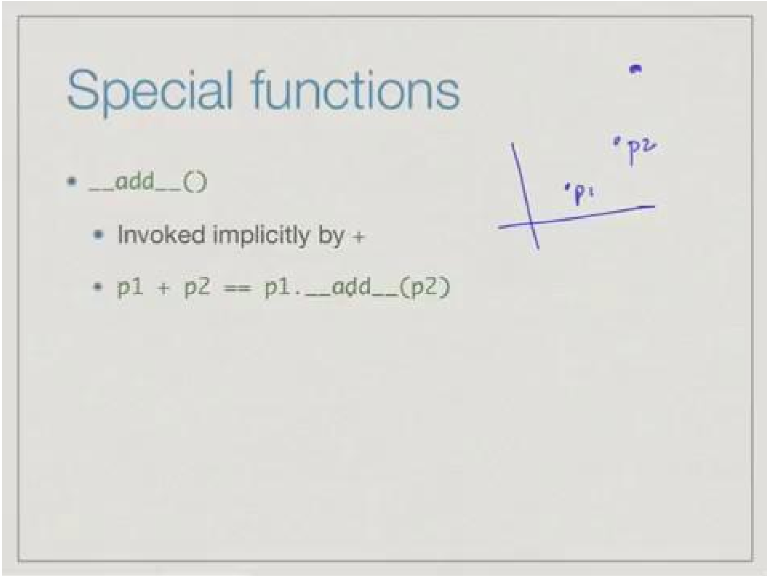

The function init clearly looks like a special function because of these underscore underscore on either side which we normally do not see when we or normally do not thing of using to write a python function. As we said before python interprets init as a constructor, so when we call a object like p equal 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.

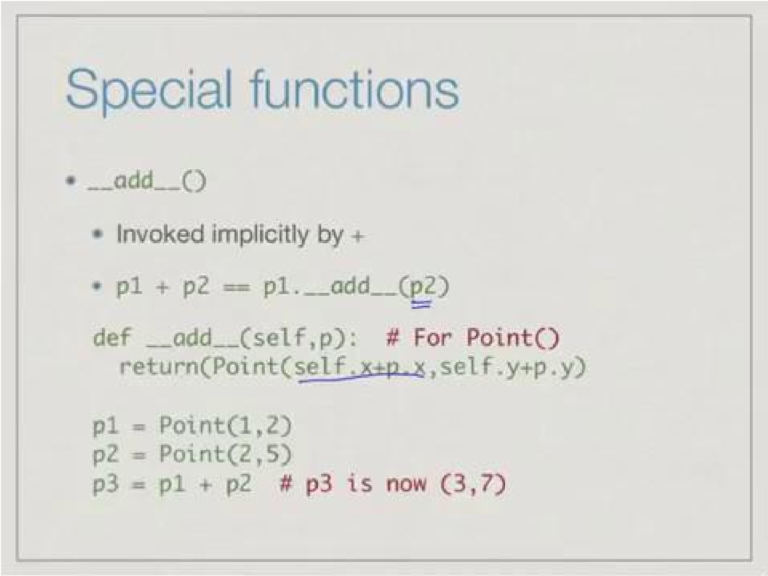
(Refer Slide Time: 16:55)




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 p 1 plus p 2, if these are two points would invoke p 1 underscore underscore add p 2. So, what we would expect that if I had p 1 and if I had p 2 then I would get something which gives me a point where I combine these two.

So, I get the x coordinate as x 1 p 1 plus p 2 and y coordinate p 1 plus p 2. It's up to us. I mean it does not mean it has to be this way, but if I say p 1 p 2 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.

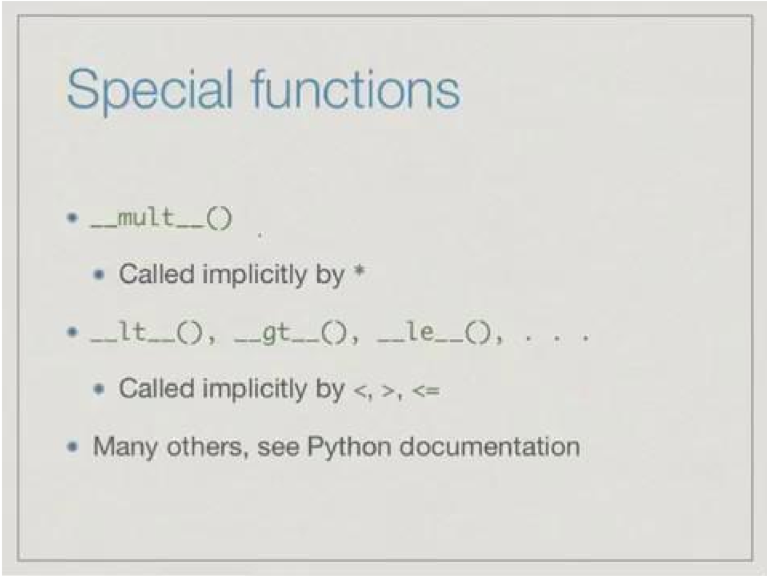
(Refer Slide Time: 17:42)




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)




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.

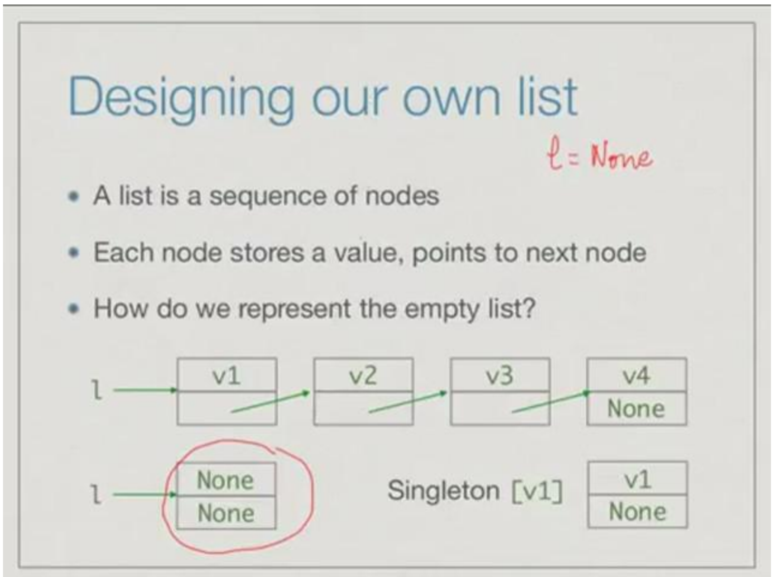
We just write p 1 less then p 2 for readability enough in our program, and internally it will call a function less than lt, similarly greater than will call the function gt, and there is something for less than equal to greater then equal to and so on. These are all very convenient functions that python 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)



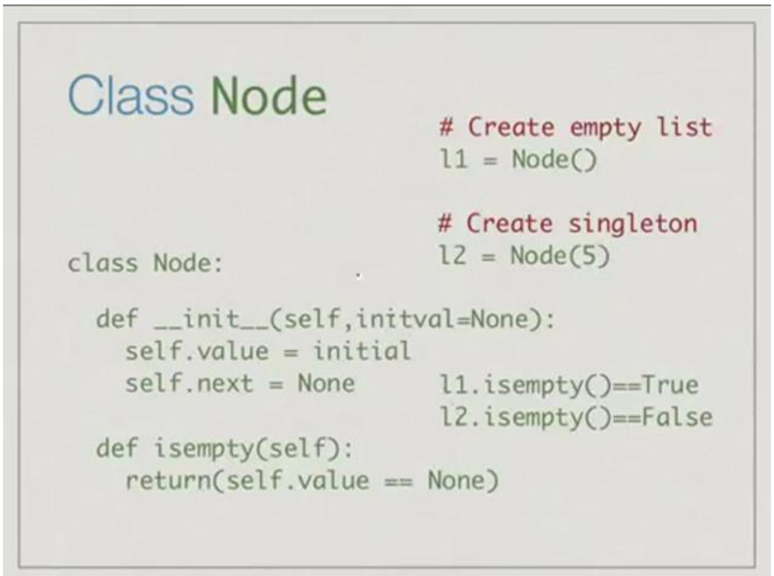

Now that we have seen the basics about how to define classes and objects, let us define an interesting data structure.

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 v1, v2, v3, v4 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)

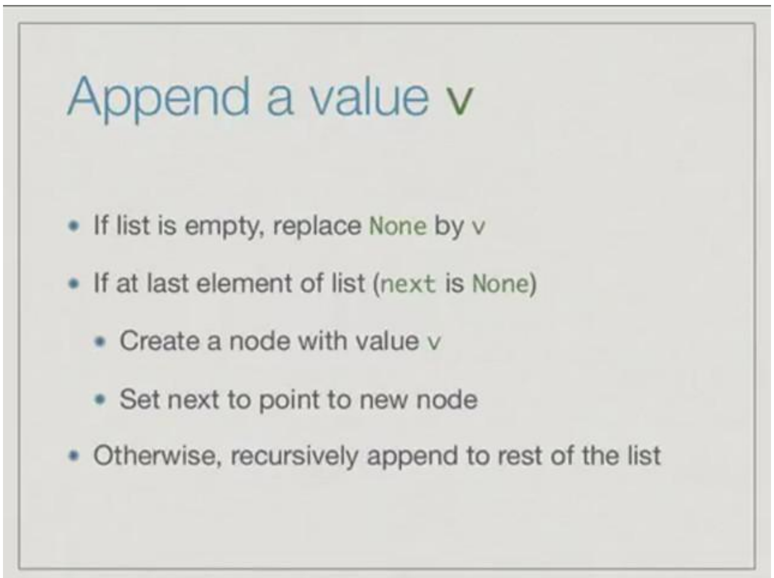



Here is the basic class that we are going to use, it is a class node. So, inside each node we have 2 attributes value and next as we said and remember that self is always used with every function to denote the object under consideration. We will use this default scheme that if we do not say anything 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.

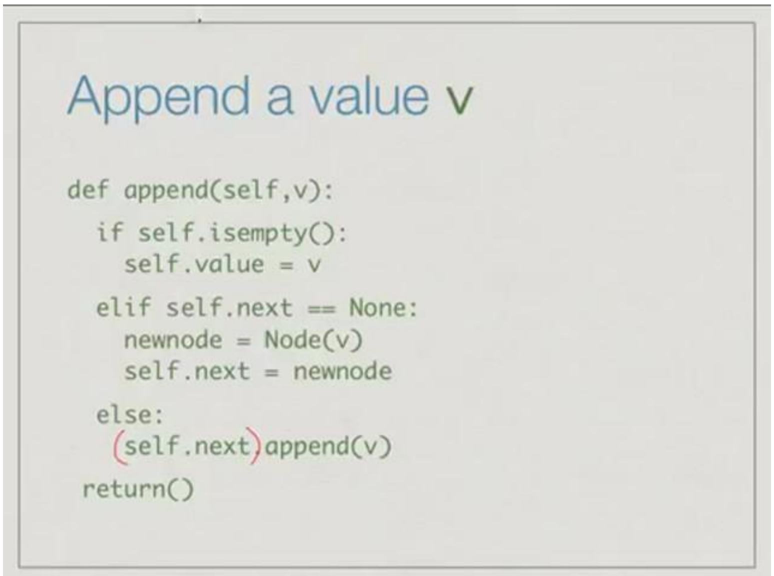
(Refer Slide Time: 03:50)




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)

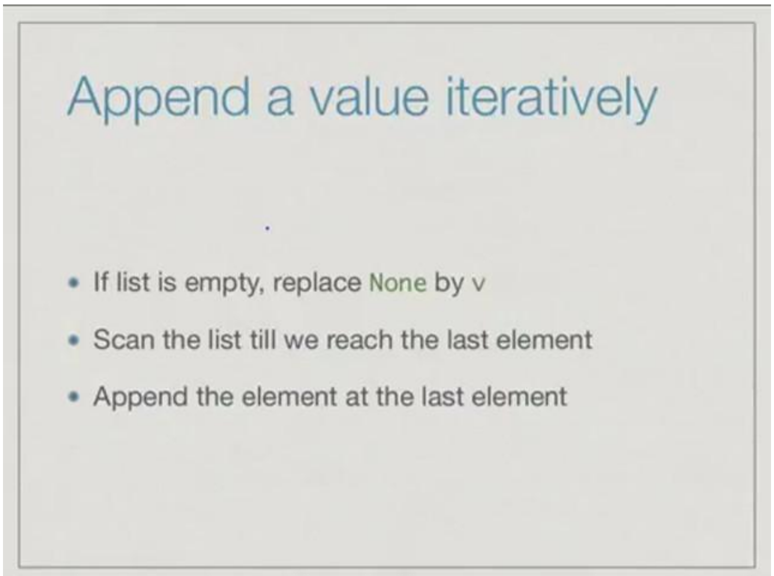



This gives us a very simple recursive definition of append. So, we take append and we want to append v to this list. If it is empty, then we just set the value to v. So, this just converts the single node with value none to the single node with value of v, otherwise if we are at the last node that is 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.

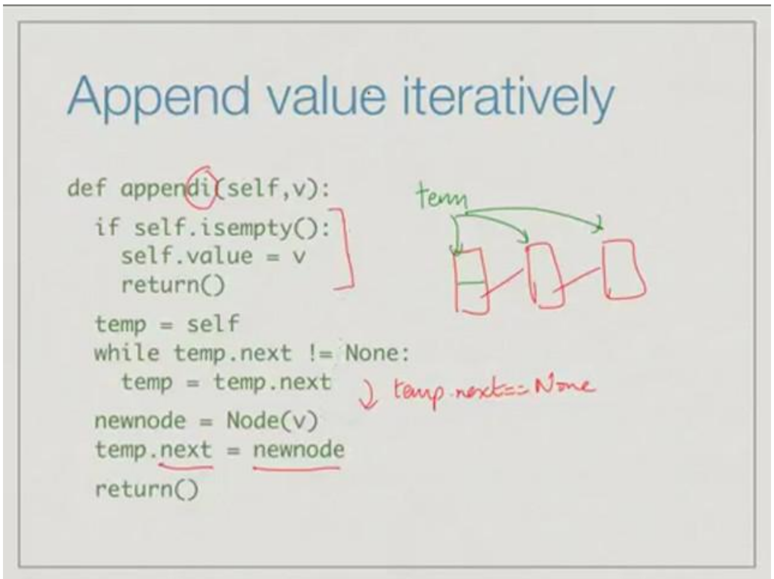
We have been abundantly careful in making sure that this is parsable. So, we have put this bracket saying that we take the object self dot next and apply append to that actually python will do this correctly. We need not actually put the bracket, we can just write self dot next dot append v and python will correctly bracket this as self dot next dot append. So, this dot is taken from the right.

(Refer Slide Time: 06:23)




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.

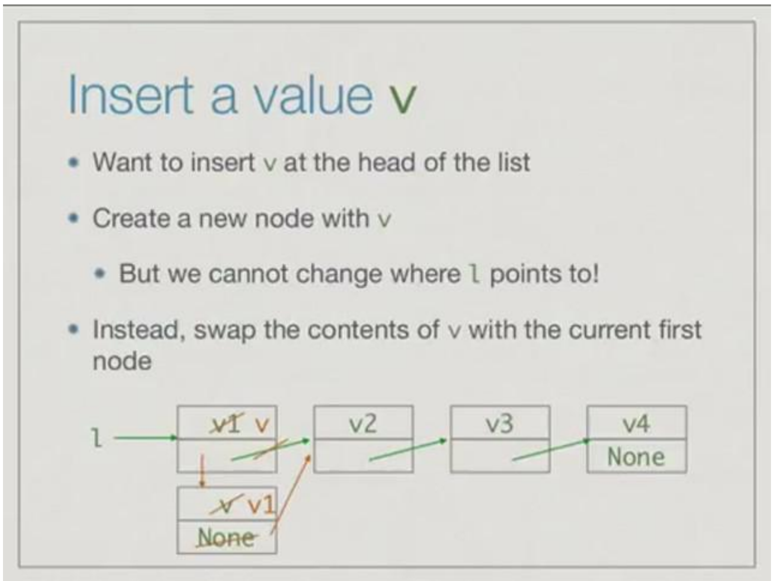
(Refer Slide Time: 06:53)




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)



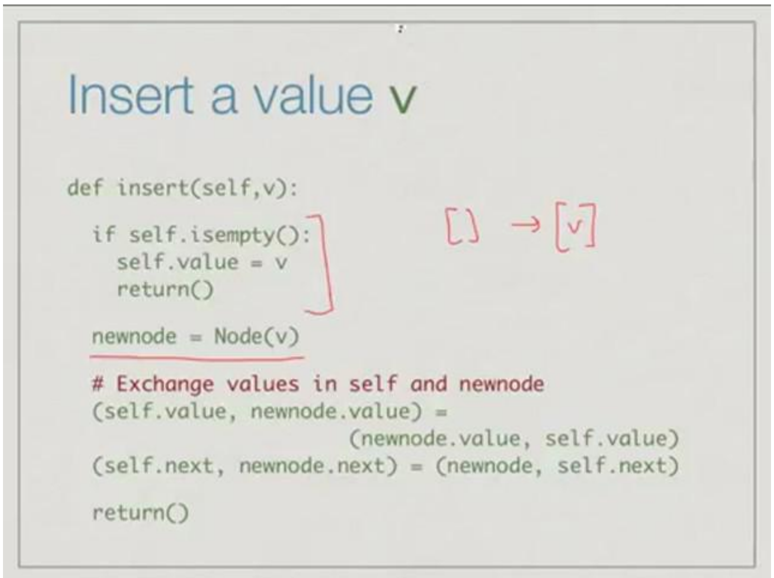

What if we do not want to append, but we want to insert. Now it looks normally that insert should be easier that append, but actually insert is a bit tricky. So, by insert we mean that we want to put a value at the beginning. We want to put a node here which has v and make this pointer.

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.

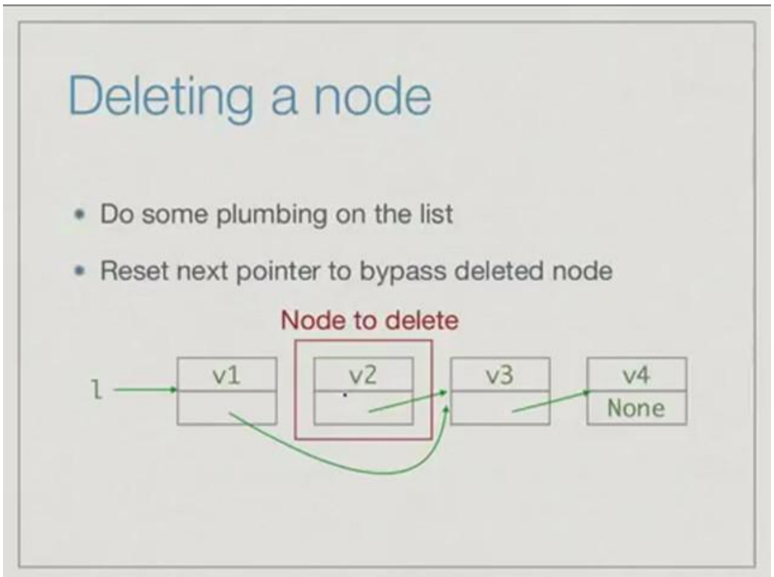
(Refer Slide Time: 10:22)




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)

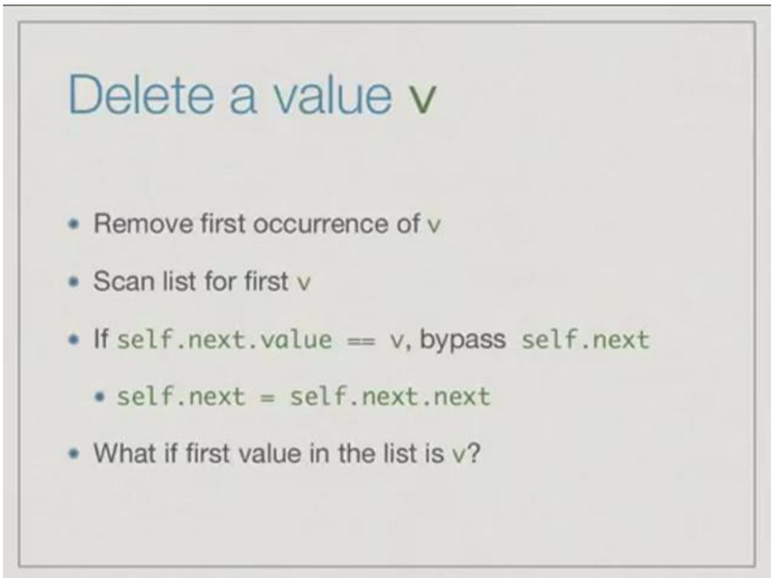



What if we want to delete a node? How do we specify to delete a node? Well we specify it by a value, but let us just suppose you want to delete say the second node in this list.

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 essense, 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.

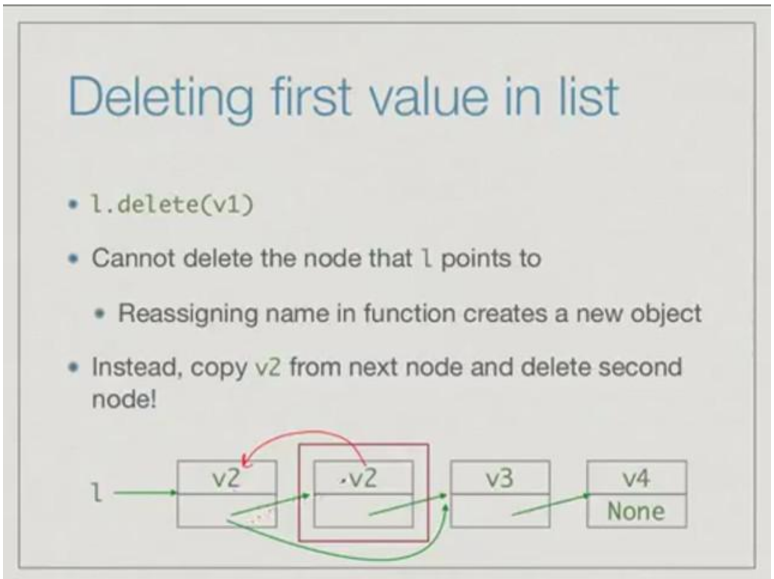
We provide a value v and we want to remove the first occurrence of v. We scan the list for the first v. Now notice that in this plumbing procedure we need to be at v 1 in order to make it point to v3. If we wanted to delete the next node then we are in good shape because we can take the next dot next 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)



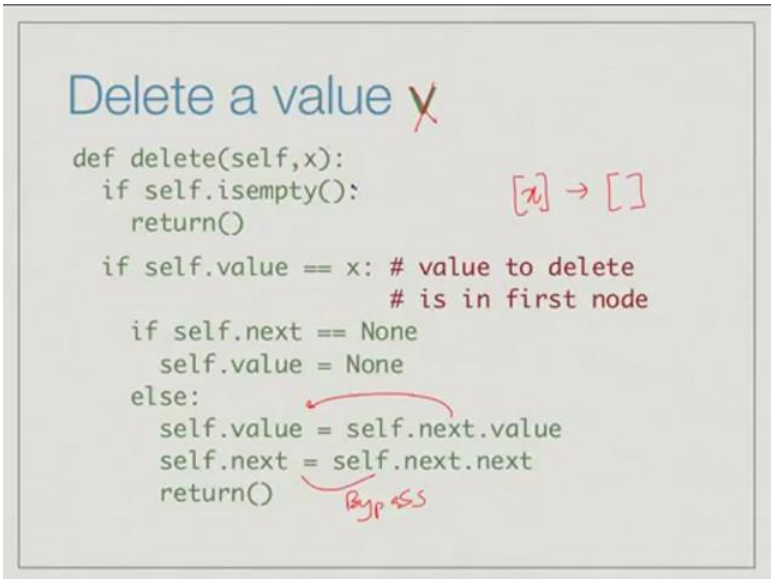

What we will actually do is we will scan by looking at the next value. If the self dot next dot value is v that is if the next node is to be deleted then we bypass it by saying the current node's next is not the next node that we had, but the next node's next. So, self dot next is reassigned to self 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)




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.

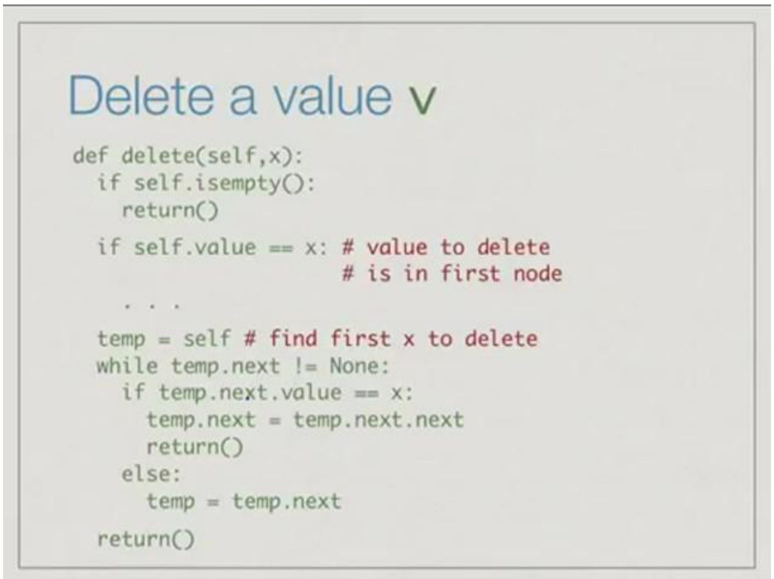
(Refer Slide Time: 14:07)




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.

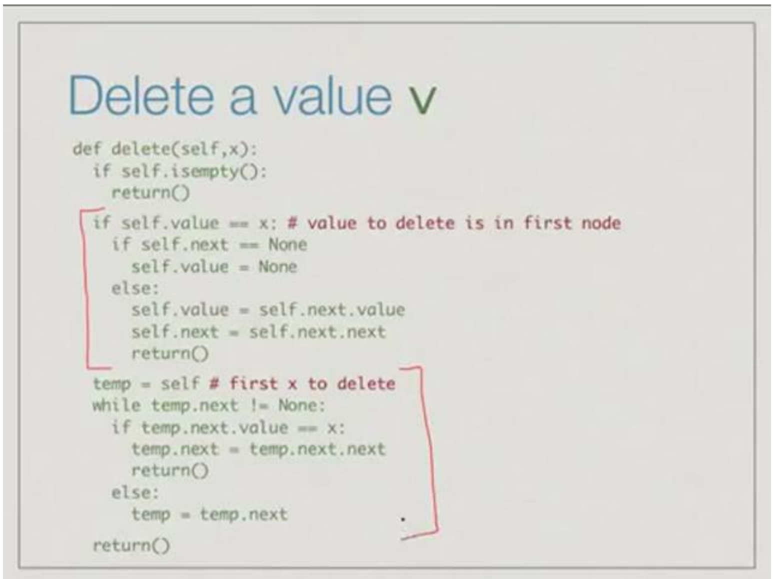
This is the easy case, but if it is not the first node, I mean, it is the first node and this is not also the only node in the list then what we do is we do what we said before. We copy the next value. We pretend that we are deleting the second node. So, we copy the second value into the first value 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)



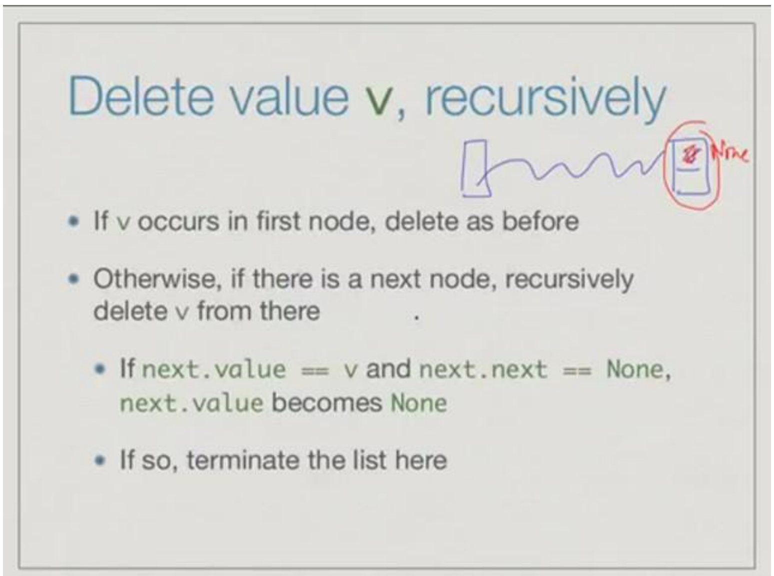

And if this is not the case then we just walk down and find the first x to delete. We start as… this is like our iterative append. We start pointing to self and so long as we have not reached the end of the list if we find the next value is x and then we bypass it and if you reach the end of the list, 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)




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.

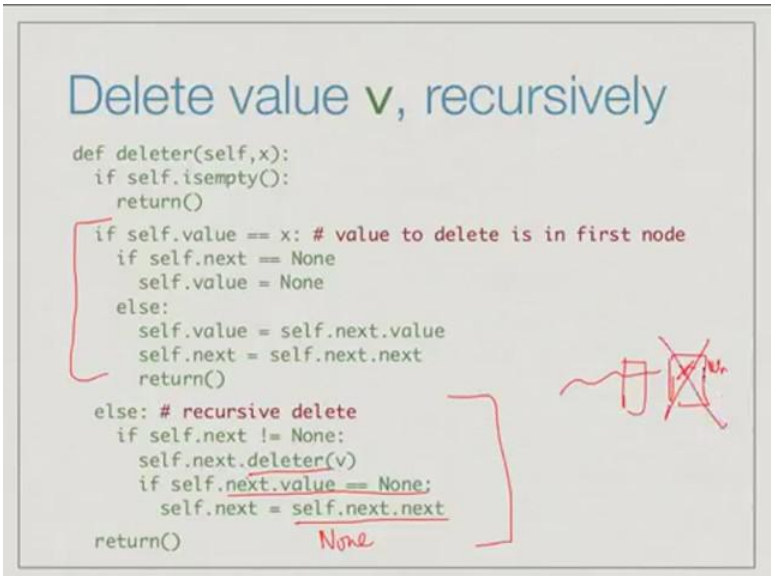
(Refer Slide Time: 16:09)




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)

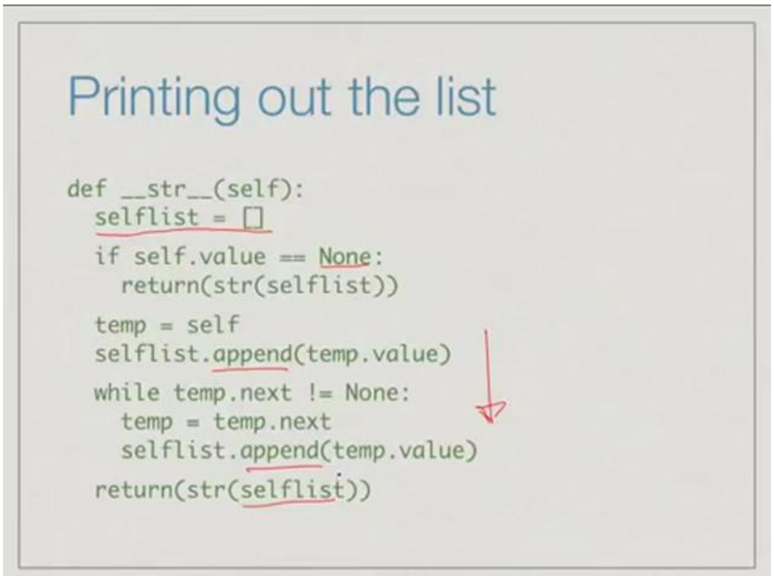



This part is the earlier part and now this is recursive part. So, recursive part is fairly straight forward. So the first part is when we delete the first element from a list, but the recursive part we check if self dot next is equal to none then we delete recursively that is fine. So, this is the delete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

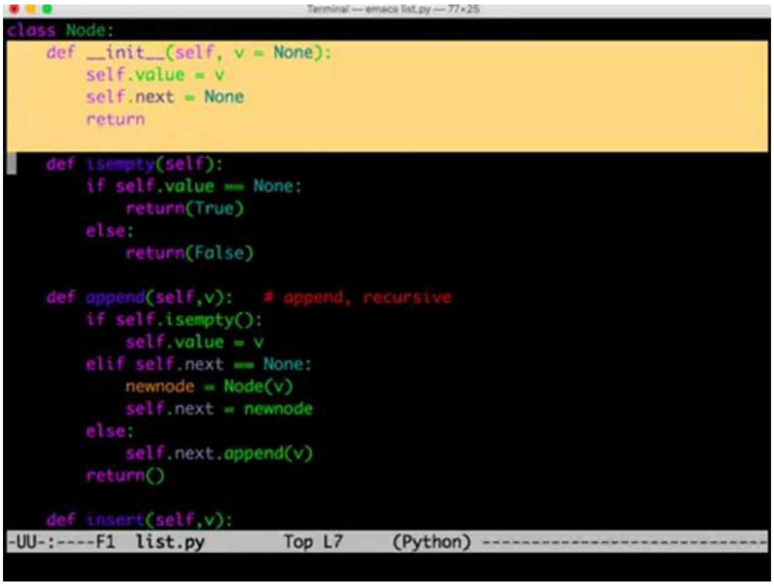
(Refer Slide Time: 18:34)




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.

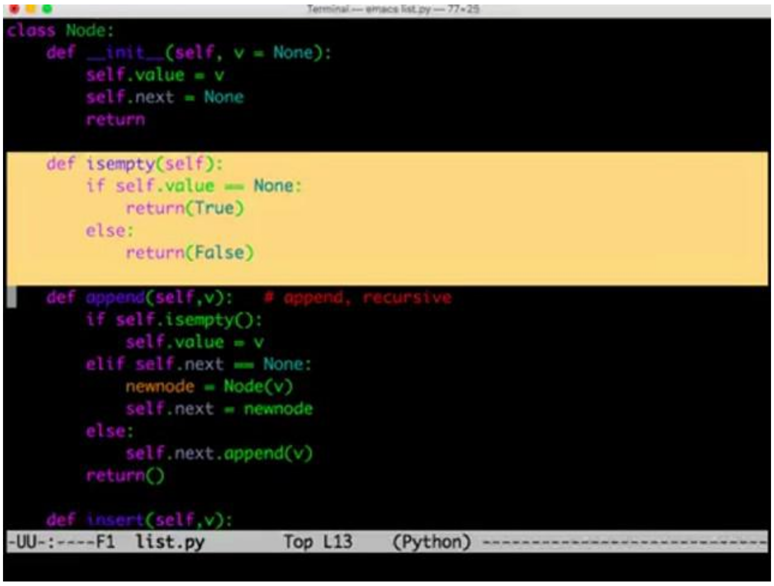
If our list, the node itself has nothing then we return the string value of the empty list, otherwise we walk down the list and we keep adding each value using the append function. So, we keep appending each value that we have stored in each node building up a python list in this process and finally, 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)



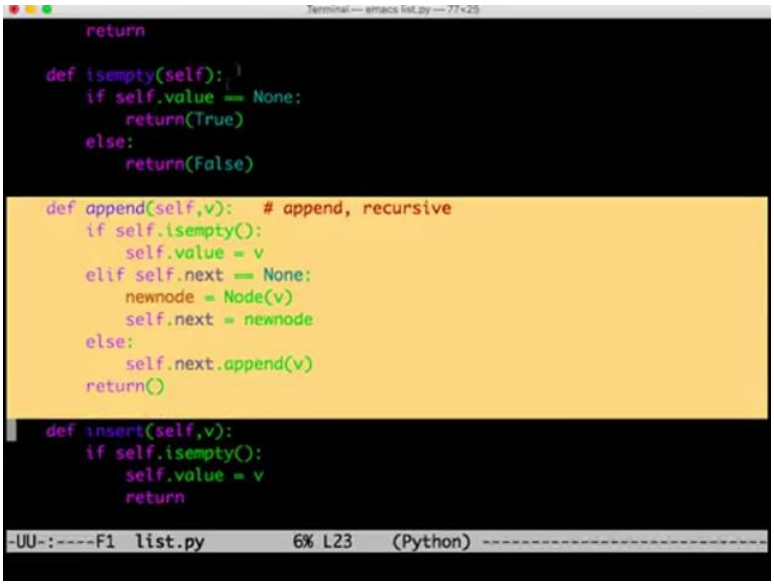

Here we have code which exactly reflects what we did in the slides. We have chosen to use the recursive versions for both append and delete. So, we start with this initial initialization which sets the initial value to be none by default or otherwise v as an argument provided.

(Refer Slide Time: 19:44)



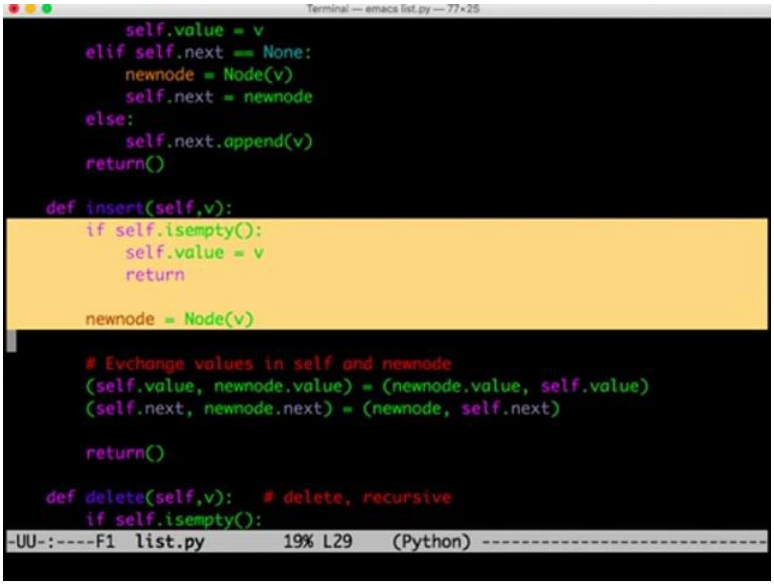

Then isempty just checks whether self dot value is none, we had written a more compact form in the slide by saying just return self dot value equal to equal to none, but we have expanded it out as an if statement here.

(Refer Slide Time: 19:56)



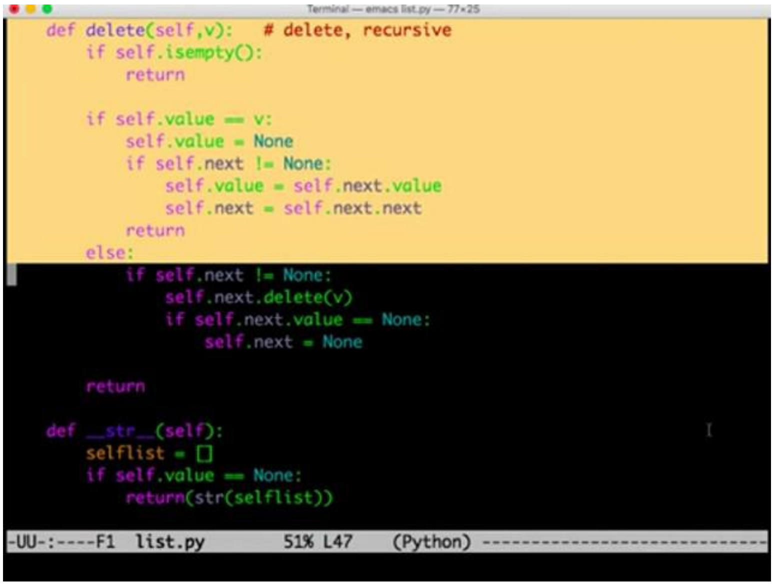

Now, this is the append function. So, append just checks if the current node is empty then it puts it here otherwise it creates a new node... if we have reached the last node it creates a new node and makes the last node point to the new node, otherwise it recursively appends. Then we have this insert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

(Refer Slide Time: 20:29)




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.

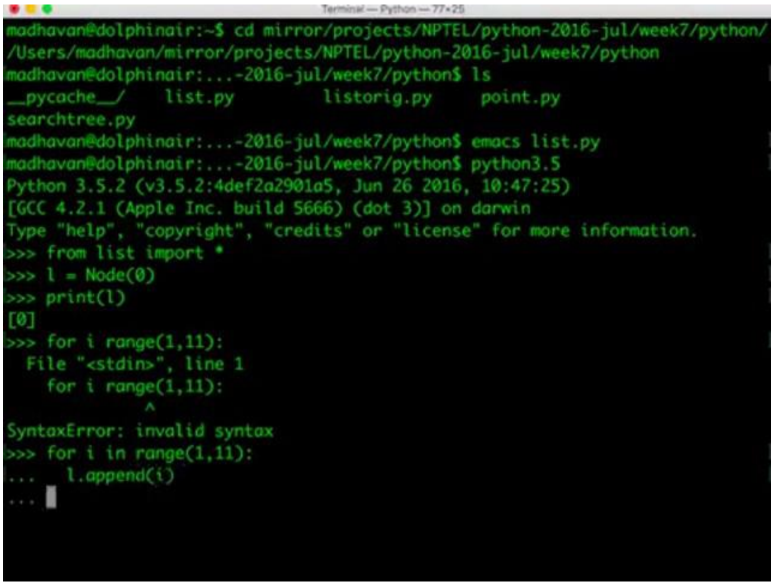
(Refer Slide Time: 21:02)




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.

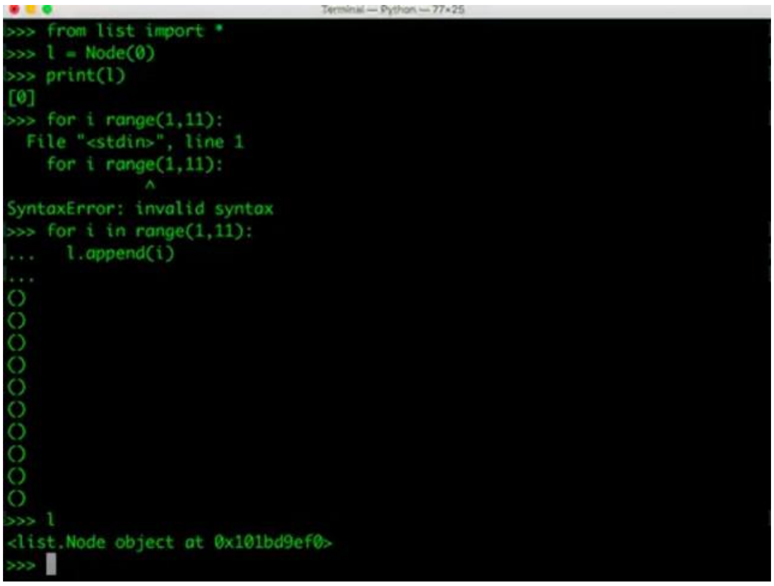
So, these 2 lines here just make sure that we do not leave a spurious empty node at the end of the list. And finally, we have this str function which creates a python list from our values and eventually returns a string representation of that list.

(Refer Slide Time: 21:54)



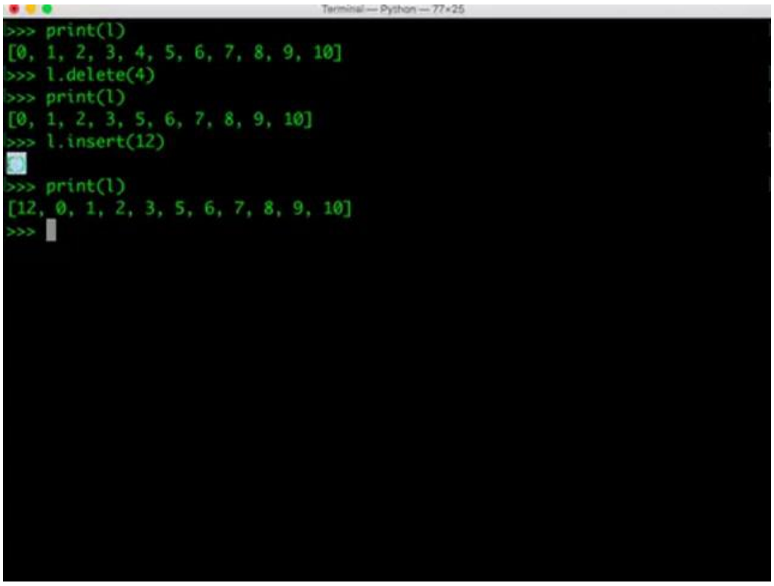

If we now run this by importing, then we could say, for instance, that l is a list with value 0 and if we say print l then we will get this representation 0, we could for instance put this in a loop and say for i in range 1 say 11, l dot append i.

(Refer Slide Time: 22:34)




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

(Refer Slide Time: 22:40)

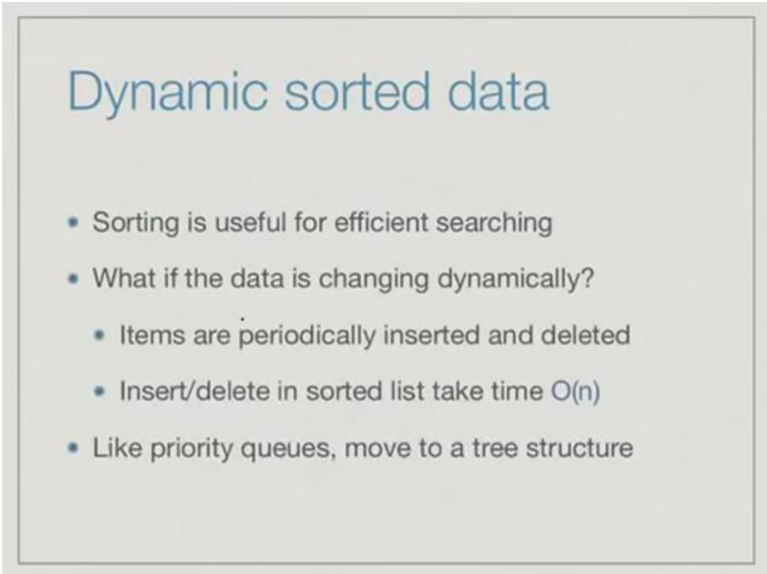



Now we say l dot delete 4 for instance and we print l then 4 is formed and so on. If we say l dot insert 12 and print l, then 12 will begin. So, you can check that this works. Notice that we are getting these empty brackets, this is the returned value. So, when we wrote this return, we wrote with the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

**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**

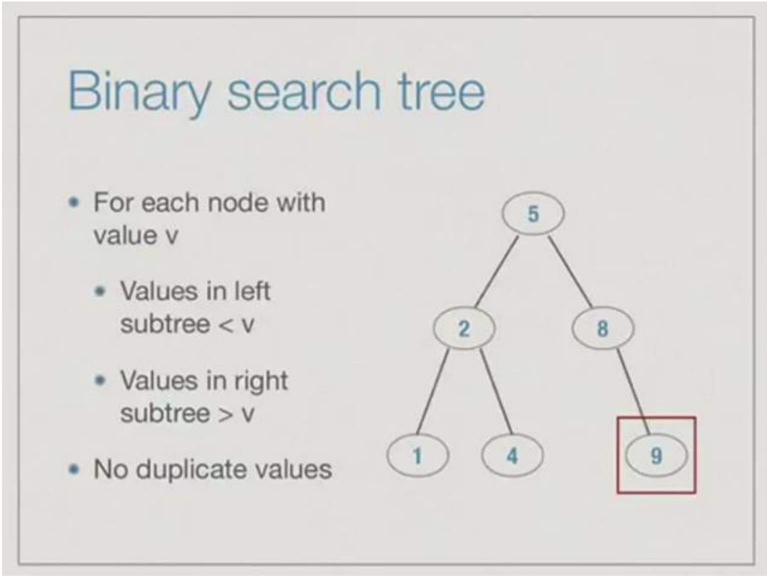
(Refer Slide Time: 00:02)




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)

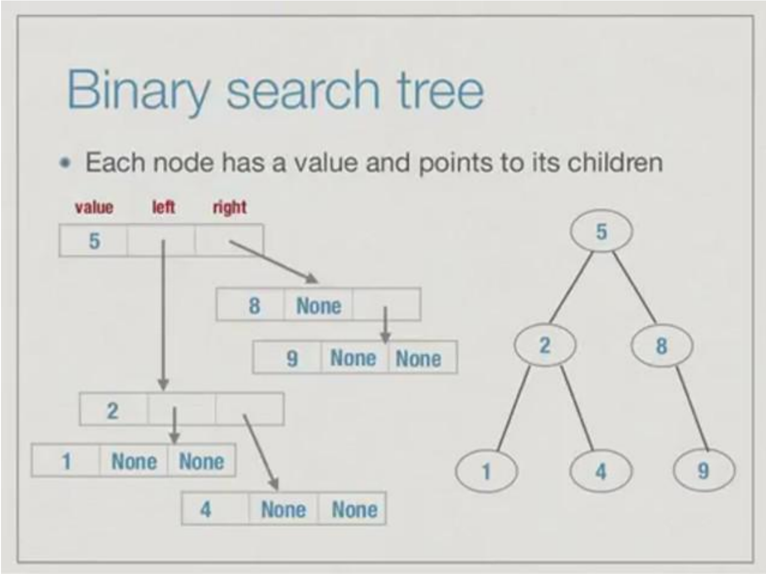



The data structure we have in mind is called a binary search tree. So, in a binary search tree we keep exactly one copy of every value. It is like a set we do not keep duplicates and the values are organized as follows for every node all values which are smaller than the current nodes value are to the 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.

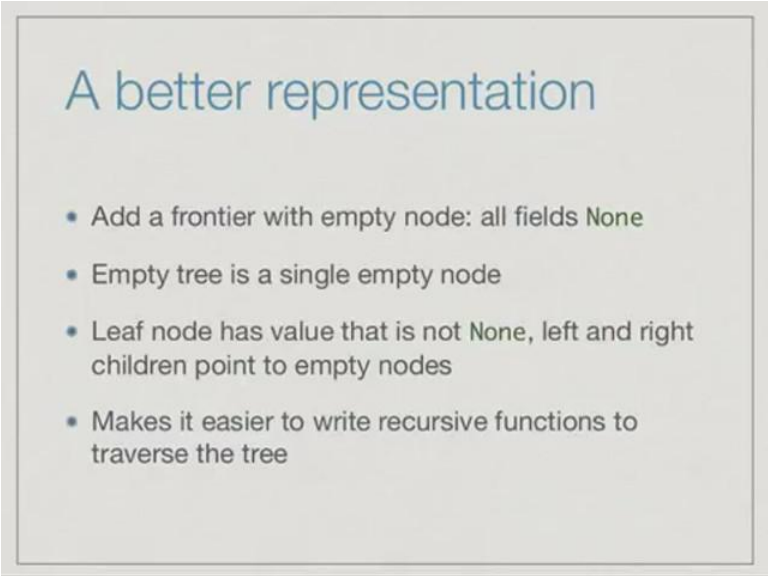
(Refer Slide Time: 02:14)




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.

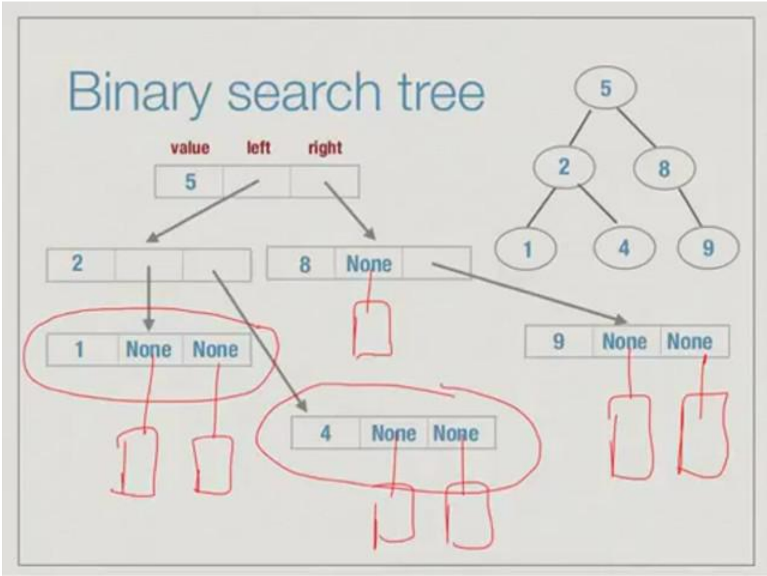
(Refer Slide Time: 03:10)




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 it is 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.

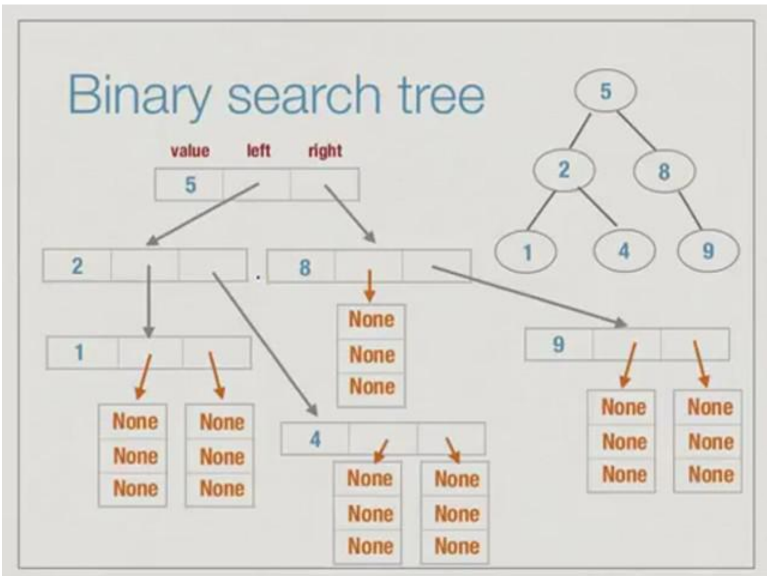
(Refer Slide Time: 03:56)




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.

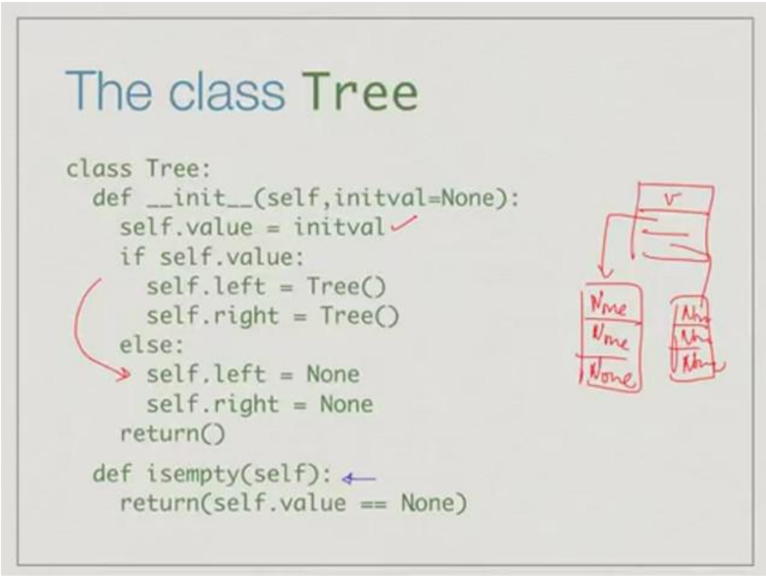
This of course, grows up the tree a little bit, but then this cost is not that much as we will as you can calculate. So, if we do this then we get a new tree which looks like this right

(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.

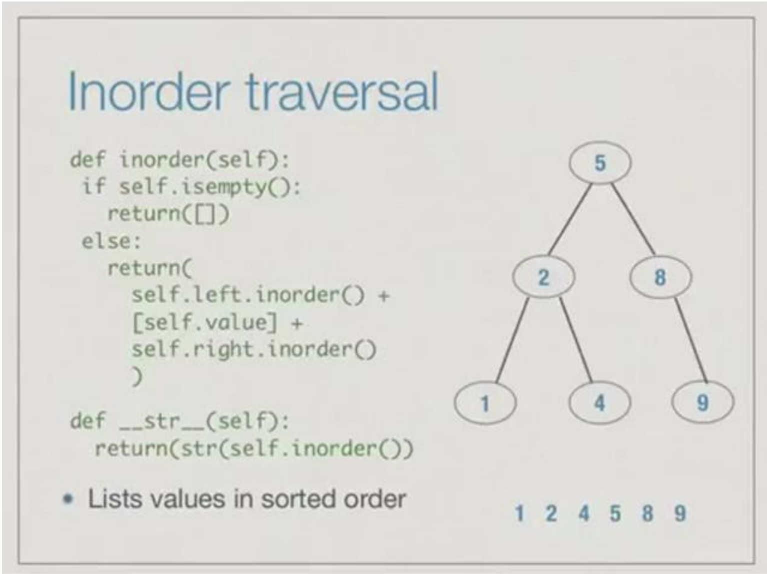
(Refer Slide Time: 04:58)




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)



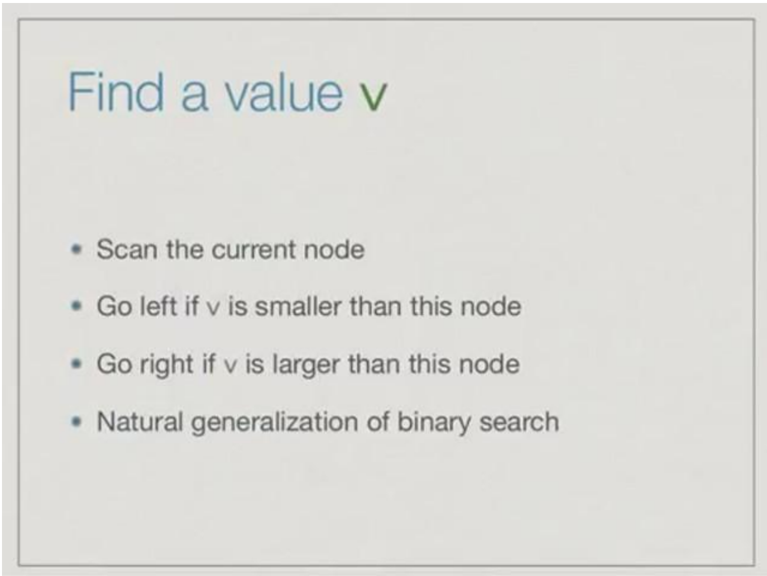

Let us first look at a way to systematically explore the values in a tree. These are called traversals and one traversal which is very useful for a binary search tree is an inorder traversal. So, what an inorder traversal does is that it first explores the left side. So, it will first explore this recursively 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.

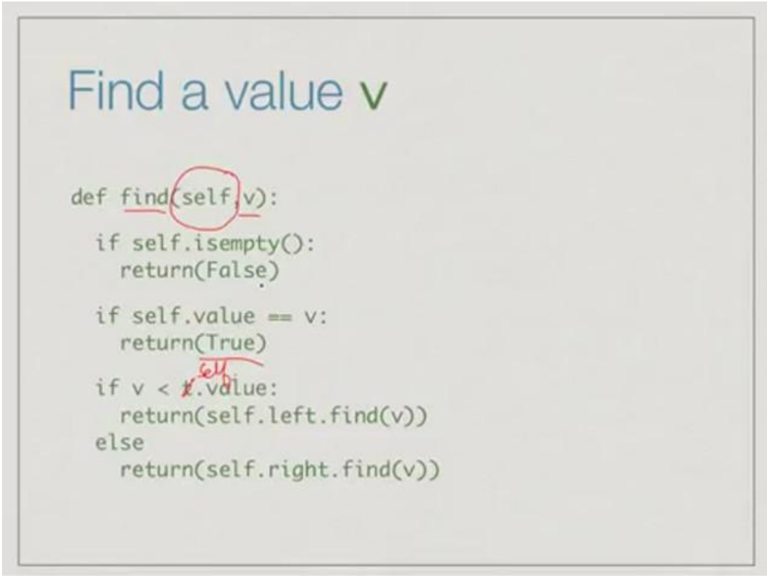
(Refer Slide Time: 08:16)




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.

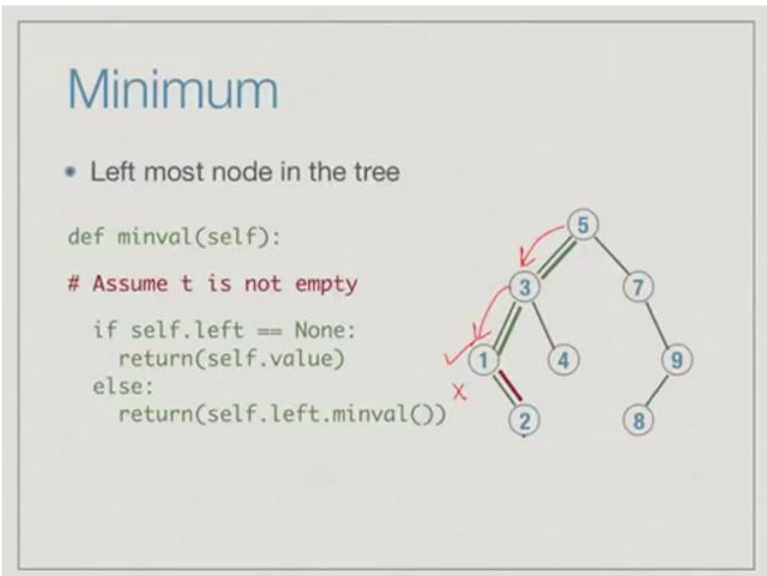
(Refer Slide Time: 09:04)




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, 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.

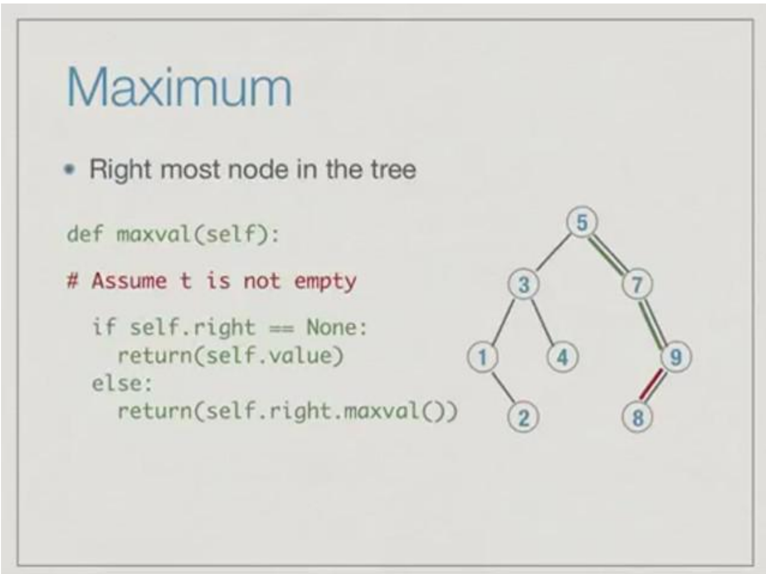
(Refer Slide Time: 09:43)



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.

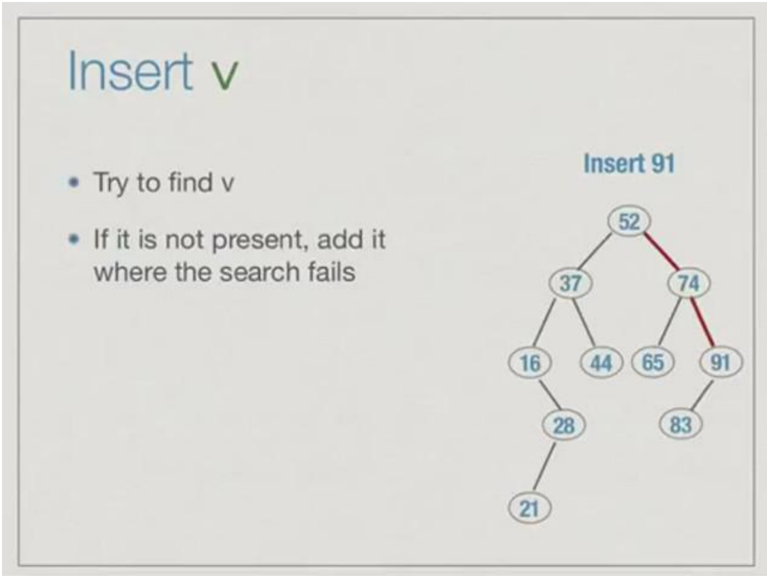
(Refer Slide Time: 10:50)




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)

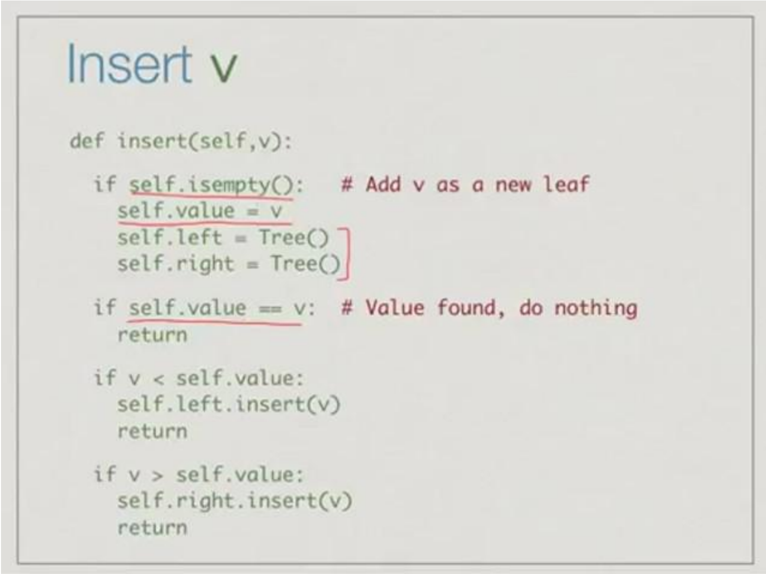



So, one of the things we said is that we need to be able to dynamically add and remove elements from the tree. The first function that we look for is insert, how do we insert an element in the tree well it is quite easy we look for it if we do not find it then the place where are search concludes is 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 out 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.

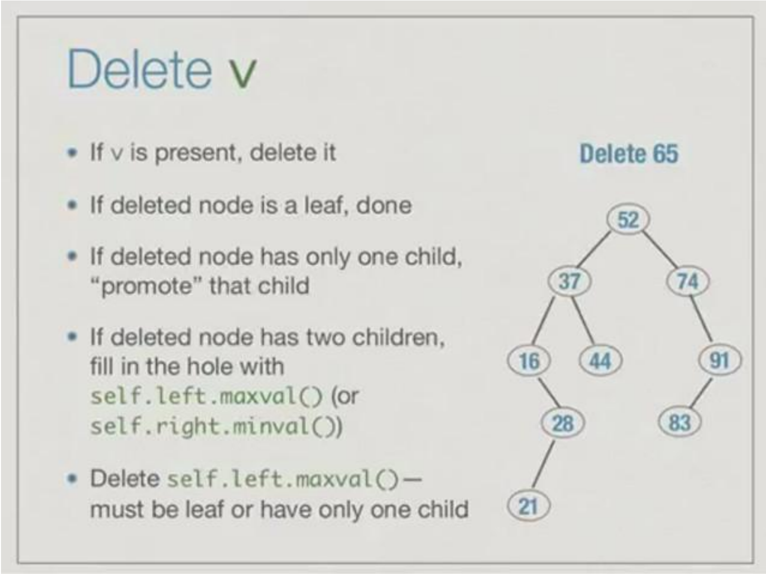
(Refer Slide Time: 12:50)




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.

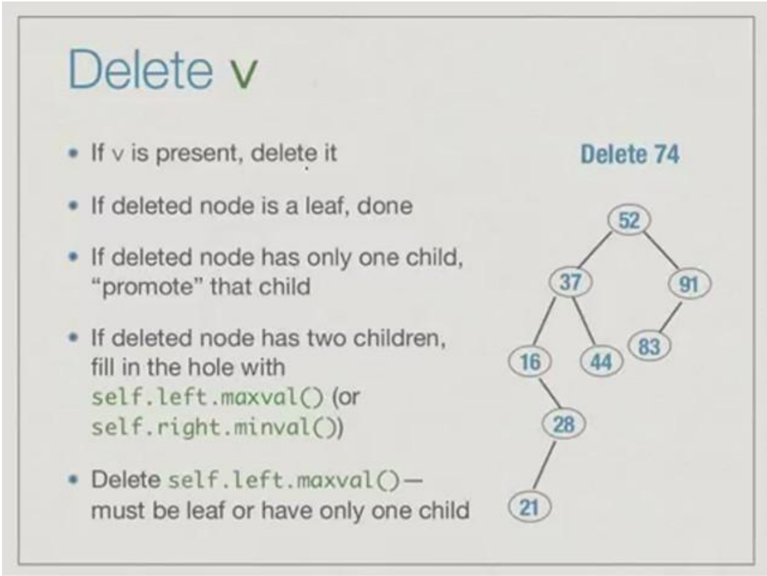
(Refer Slide Time: 13:32)




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.

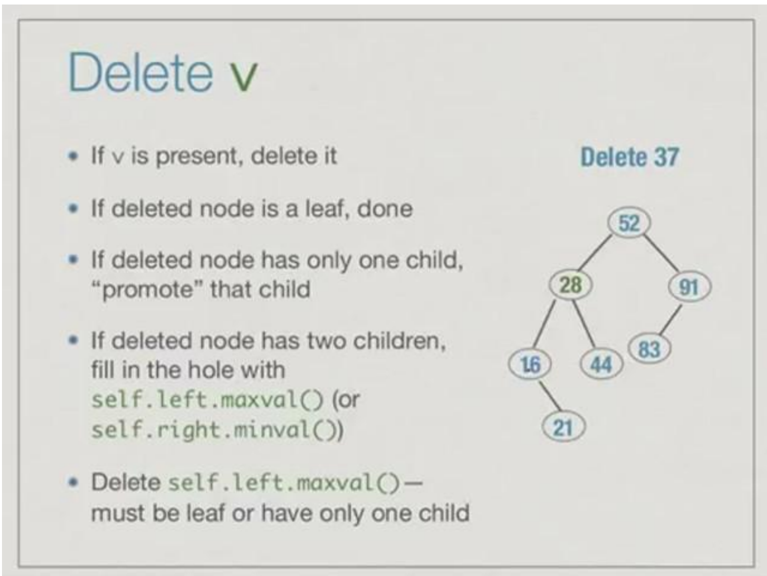
So, we search for v as usual now if the node that we are searching for is a leaf then we are done we just delete it and nothing happens, if it has only one child then we can promote child. If it has two children we have a hole we have a leaf we have a node which we have to remove value, but we have values 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)




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)

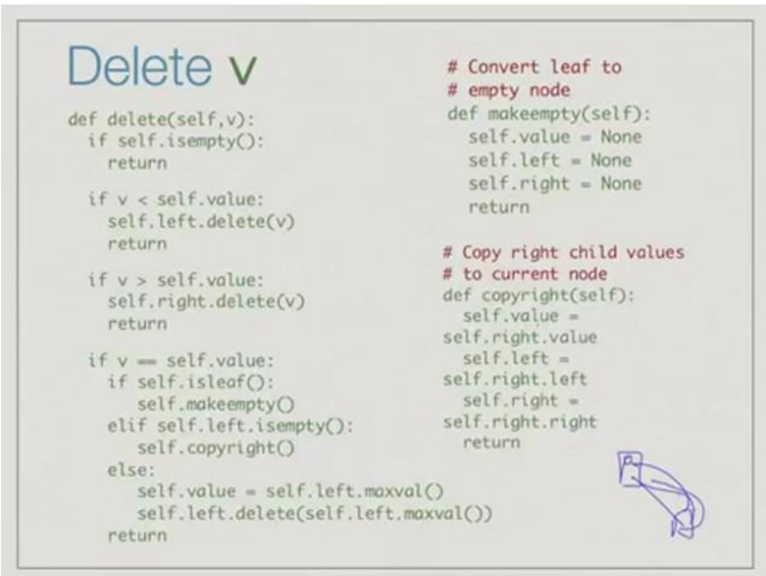


Now, finally, we have the difficult case which is you want to delete a node which is in the middle of the tree, in case this case 37. So, we come to 37 and we want to remove this. Now, if we remove this there will be a vacancy now, what do we fill the vacancy again and how do we adjust the shape of the 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 walked out remove the 28 and promote the 21. So, this is exactly how delete works.

(Refer Slide Time: 16:37)



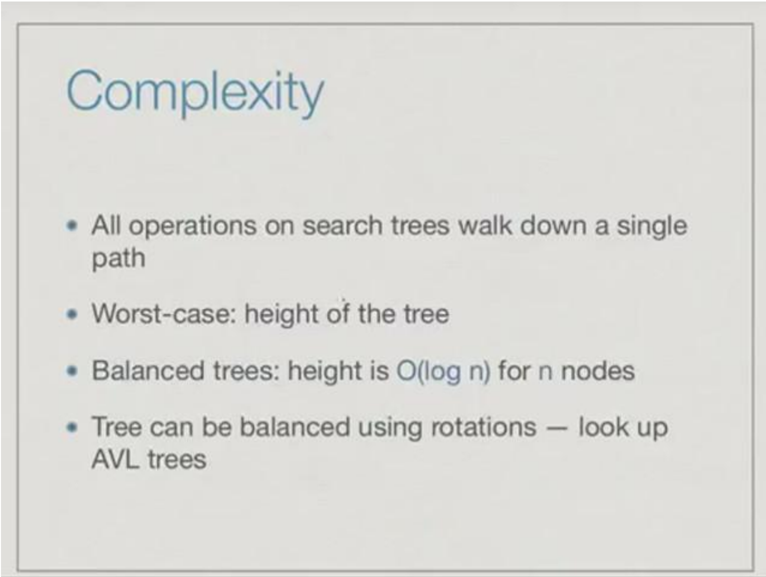
We can now look at the function. If there is no value v then we just return the easy cases are when we do not find we are the current thing. So, if it is less than the current value then we go to the left and delete if it is bigger than the current value we go to the right and delete. So, the hard work 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 of 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)

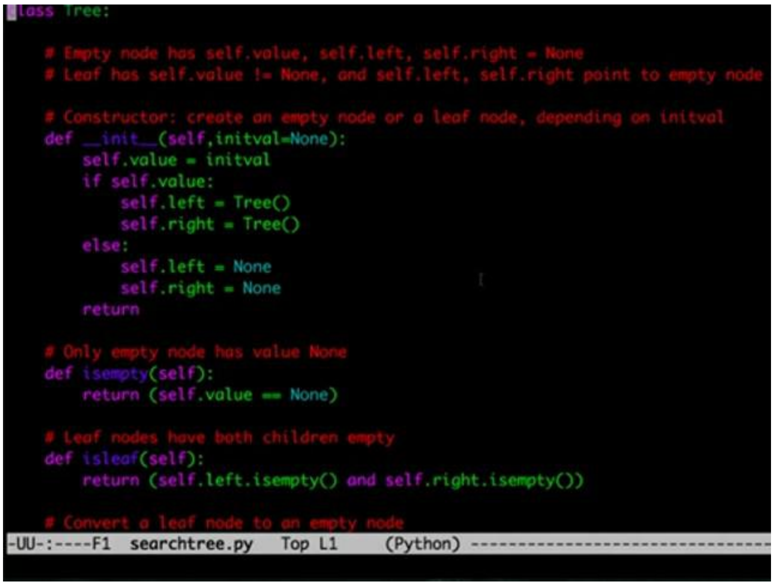



So, how much time do all these take well if you examined the thing carefully you would realize that in every case we are just walking down one path searching for the value and along that path either we find it or we go down to the end and then we stop. So, the complexity of every operation is actually 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 chair roughly have the same size.

If we have a balanced tree then it is not difficult to see then we have height logarithmic in an 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.

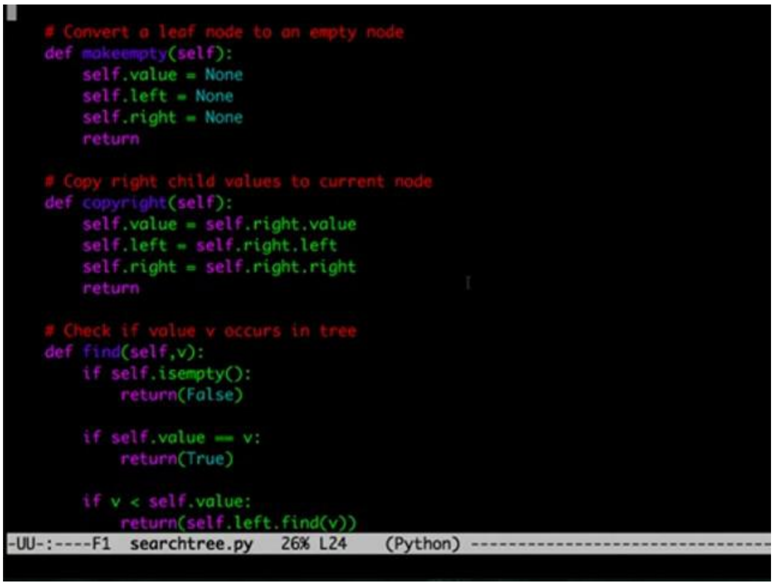
Let us just look at the code directly and execute it and convenience ourselves that all the things that we wrote here actually work as intended.

(Refer Slide Time: 19:55)



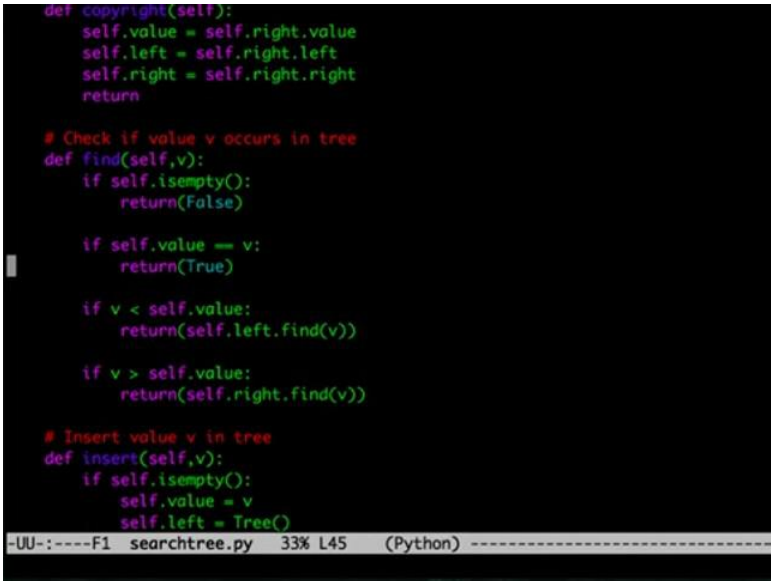

Here we have a python code for the class tree that we showed in the lectures. So, we are just added a comment about how the empty node is organized and the leaves are organized. So, there is the constructor which sets up either an empty node or a leaf node with two empty children then we have isempty and iseleaf we check whether the current value is none or both the left and right children are empty, respectively.

(Refer Slide Time: 20:26)



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)

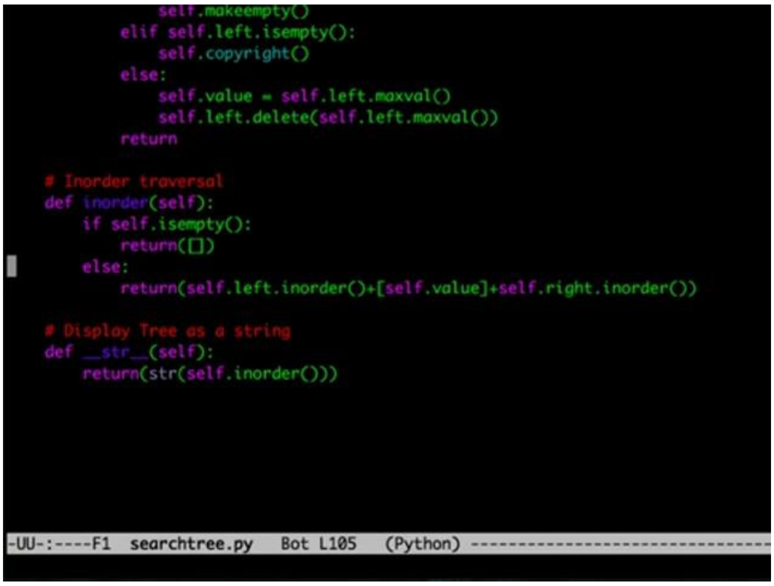



So, we start with find. So, find is the one which is equivalent to binary search then insert is like find and there it where it does not find it tries to insert.

(Refer Slide Time: 20:48)

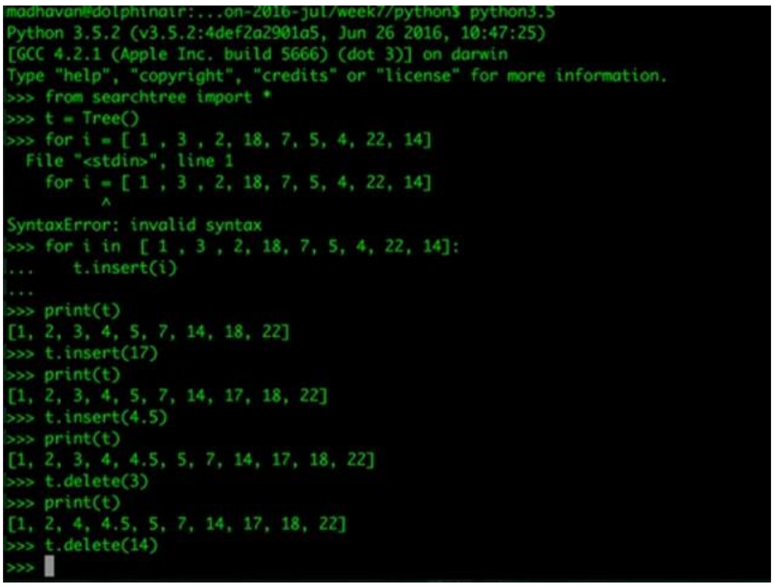

And finally, we have maxval which we made for delete and delete now when we reach the situation where we are found a value to that needs to be deleted if it is a leaf then we remove the leaf and make it empty if it is the left child is empty then we copy the right child up otherwise we delete the maximum from the left and promote that maximum value to a current.

(Refer Slide Time: 21:12)




Finally, we have this inorder traversal which generates a sorted list from the tree values and the str function just displays the inorder traversal.

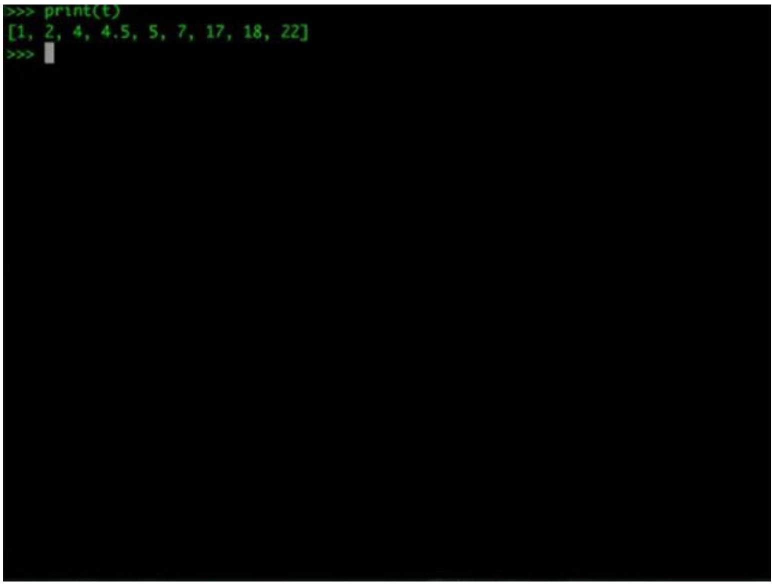
(Refer Slide Time: 21:22)




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 in 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)



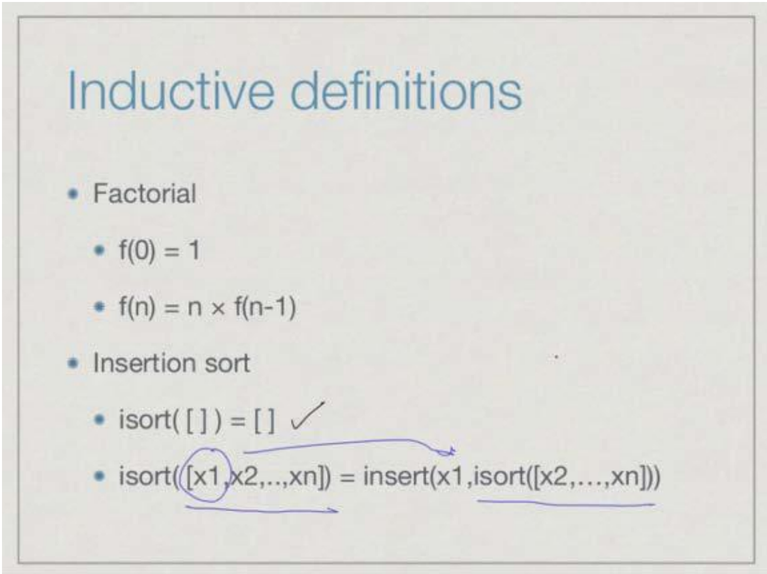

This incrementation definitely works, although we have a balanced it. If we do not balance it then the danger is that if we keep inserting and sorted sequence then we keep inserting larger values it keeps adding on the right child. So, the tree actually looks like a long path right. Then it becomes like 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**

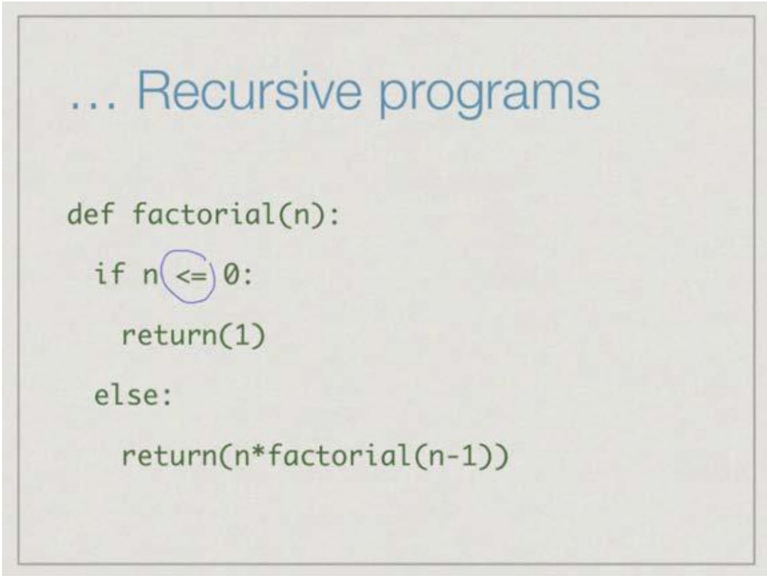
(Refer Slide Time: 00:02)




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.

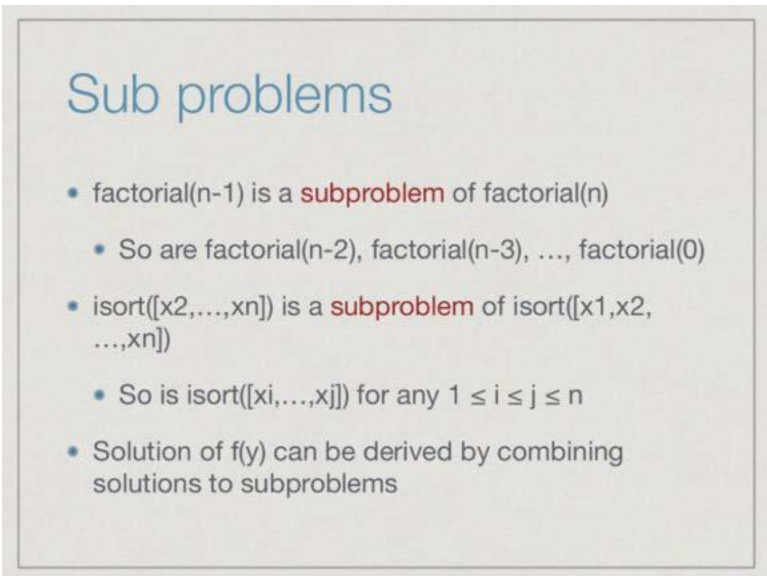
Insertion sort can be defined in terms of insert function as follows. So, isort of the base case for the empty case just gives us the empty list. And then if you want to sort a list with n elements, we pull out the first element right and then we insert it into the result of inductively sorting the rest. 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)




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)

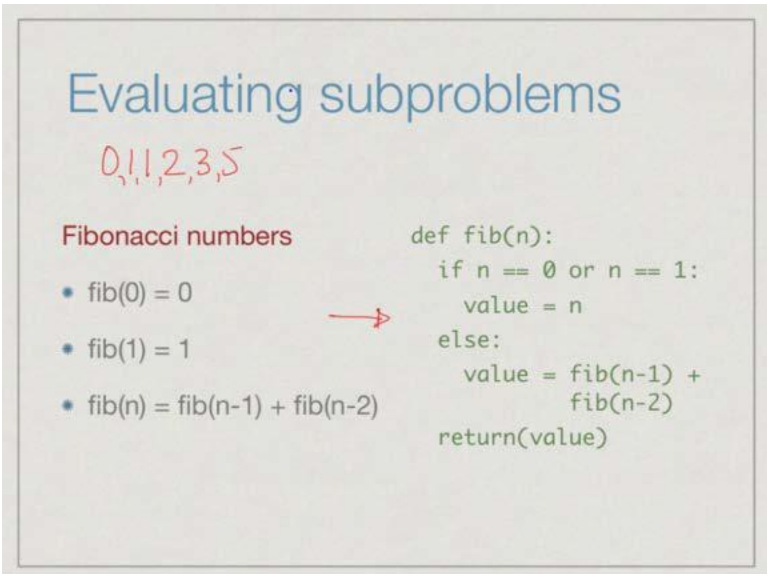


In general, when we have such inductive definitions, what we do is we have sub problems that we have to solve in order to get to the answer we are trying to reach. So, for instance, to compute factorial of n, one of the things we need to do is compute factorial of n minus 1. So, we call factorial of 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)



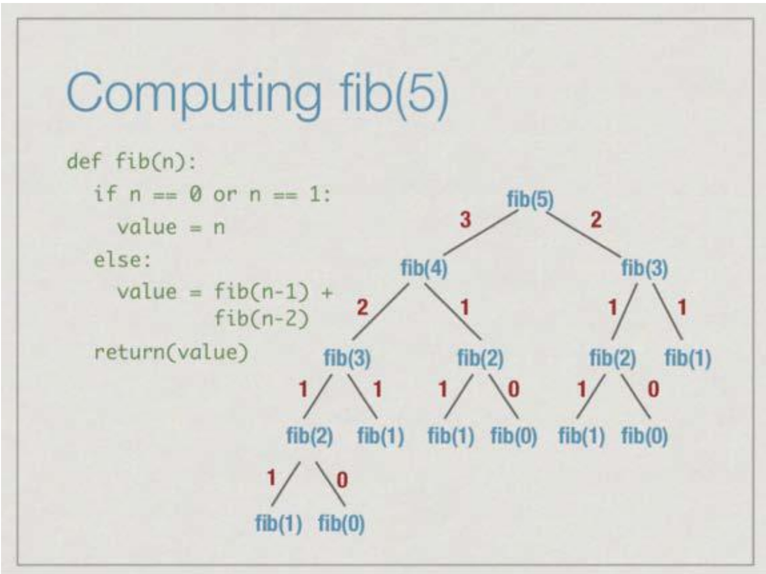

Let us look at one particular problem, which will highlight an issue that we have to deal with when we are looking at inductive specifications and naively translating them into programs. The Fibonacci numbers are a very famous sequence which were invented by

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.

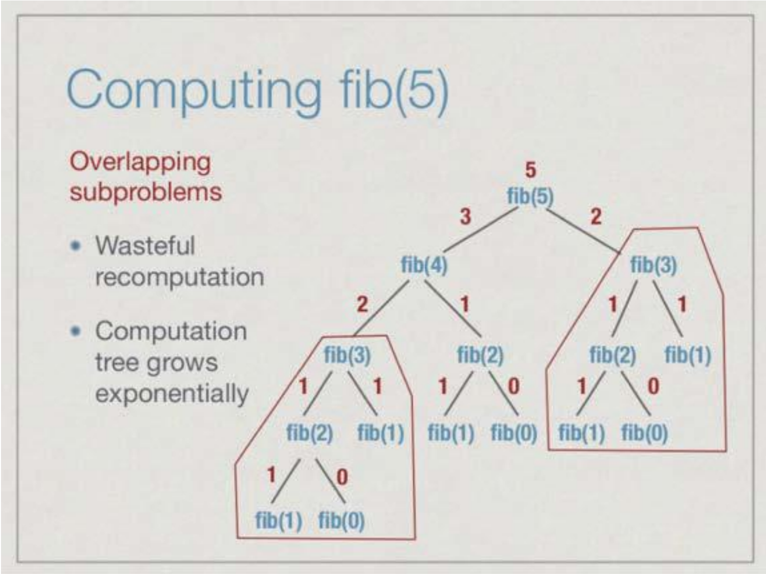
Similarly, we go to the left of the two sub problems Fibonacci of 3 requires 2 and 1; 2 requires 1 and 0. Now for 1 and 0 fortunately we can exit without making a recursive call; if n is equal to 0 or n is equal to 1, we just return the value. So, we get back Fibonacci 1 as 1 and Fibonacci 0 as 0. So, 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.

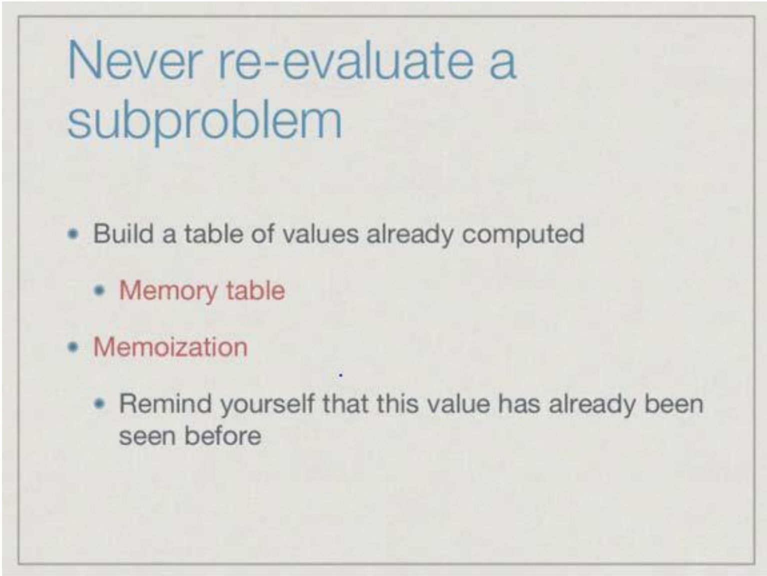
(Refer Slide Time: 06:45)




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.

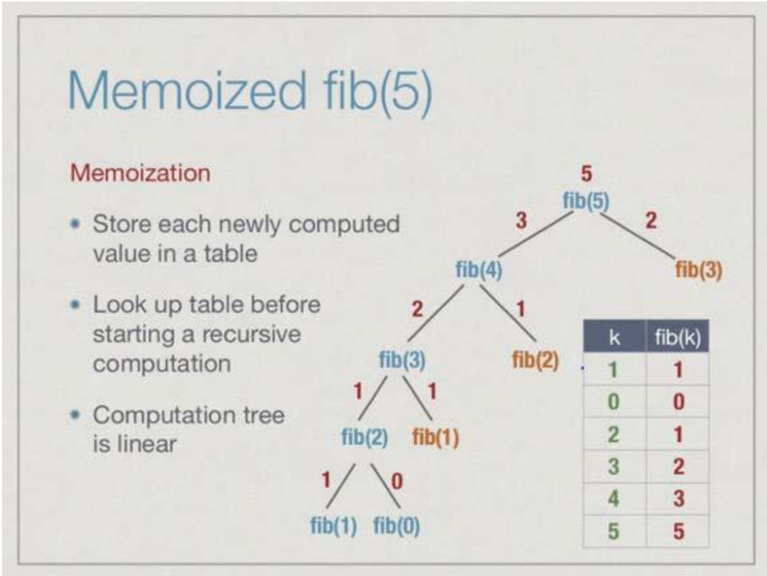
(Refer Slide Time: 07:26)




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.

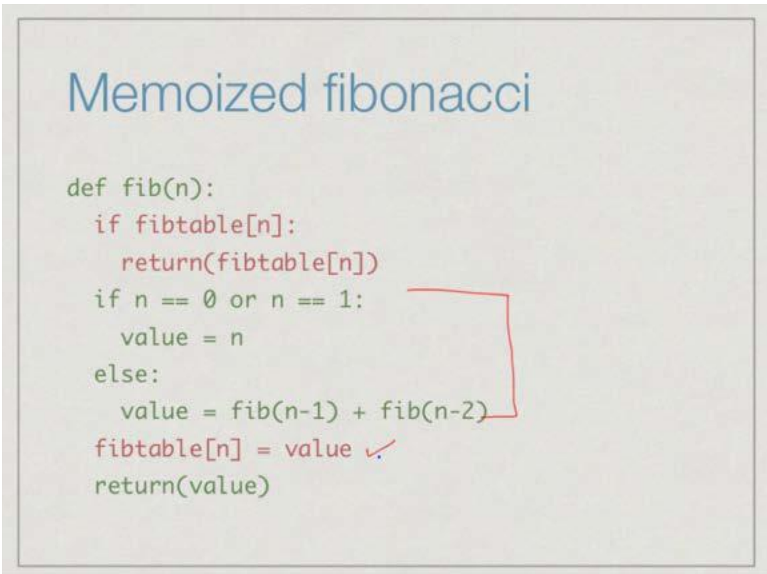
Now we come up and we realize that fib of 2 is now available to us, it is 1 plus 0 is 1. So, we store that in the table. We say for k equal to 2, fib of k is 1. Now we come back to fib of 3, and now we go down and it asks us to compute fib of 1 again. Now although this does not take us any work, because 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.

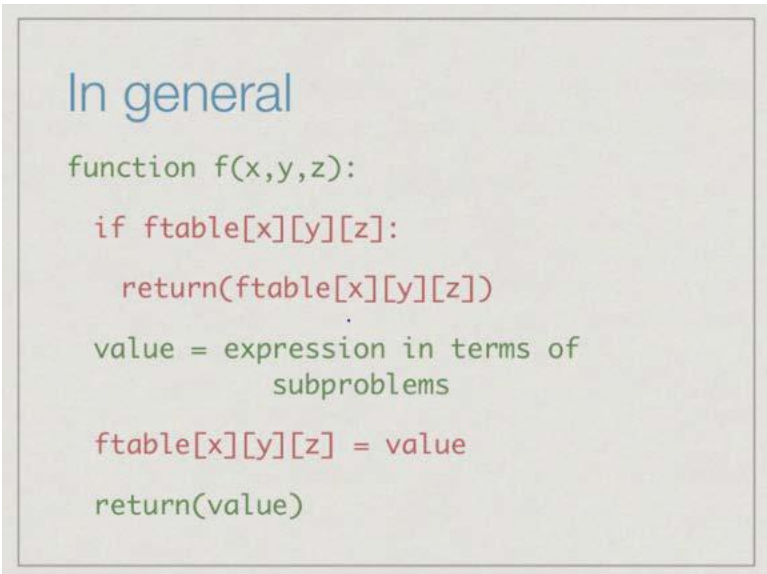
(Refer Slide Time: 11:32)



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.

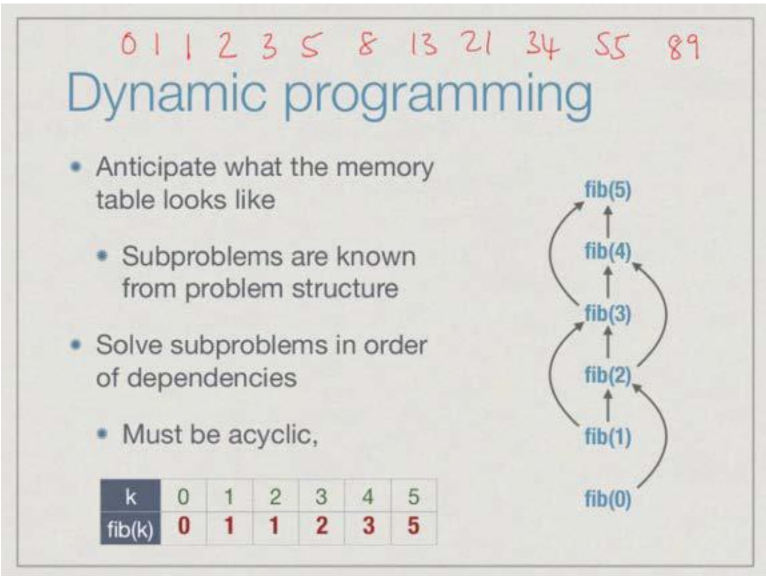
(Refer Slide Time: 12:49)




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.

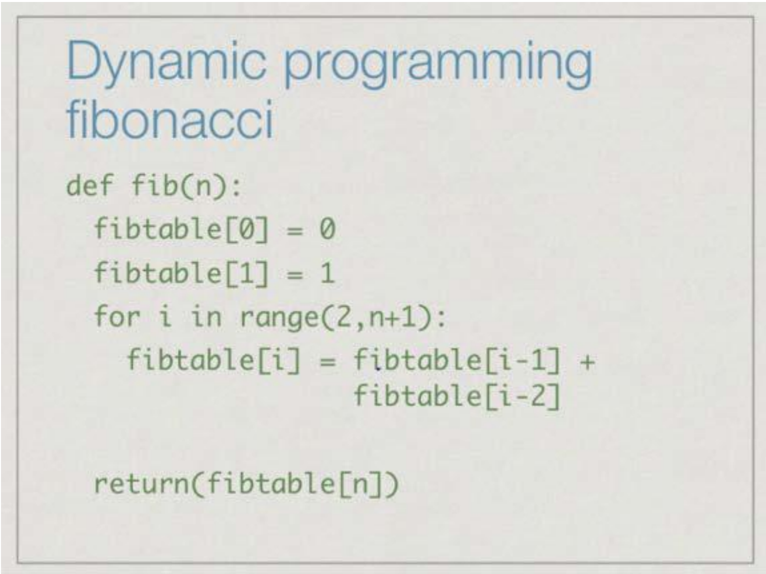
If we look at the sub problem for Fibonacci of 5, for example, it says that it requires all these sub problem but the dependency is very straightforward; 5 depends on 4 and 3; 4 depends on 3 and 2; 3 depends on 2 and 1; 2 depends on 1 and 0; and 0 and 1 have no dependencies. So, we can start at the bottom, 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.

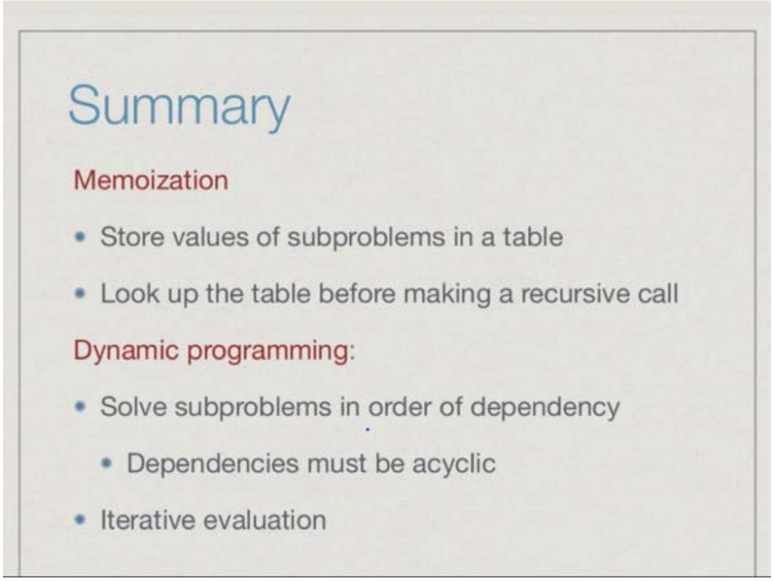
(Refer Slide Time: 17:25)




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)



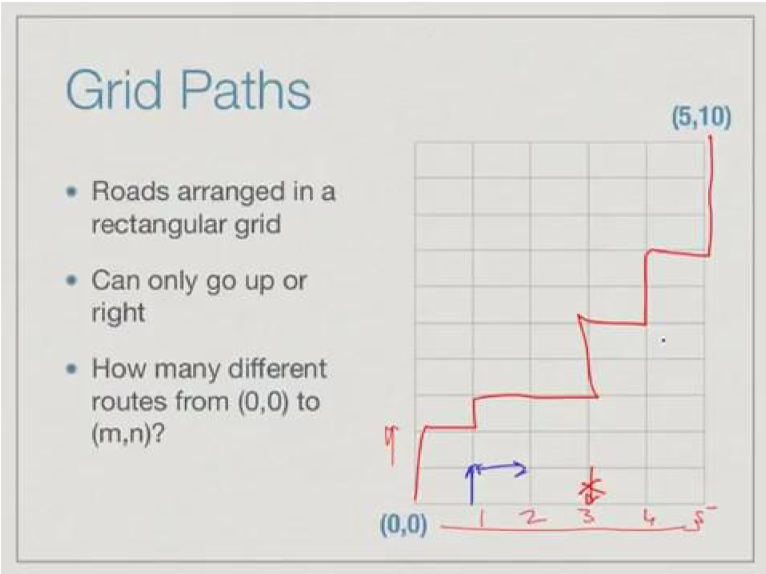

To summarize, the basic idea to make naïve recursion more efficient is to never compute something twice. And to never compute something twice, we store the values we compute in what we call a memo table this is called memoization. And we always look up the table before we make a recursive call.

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)

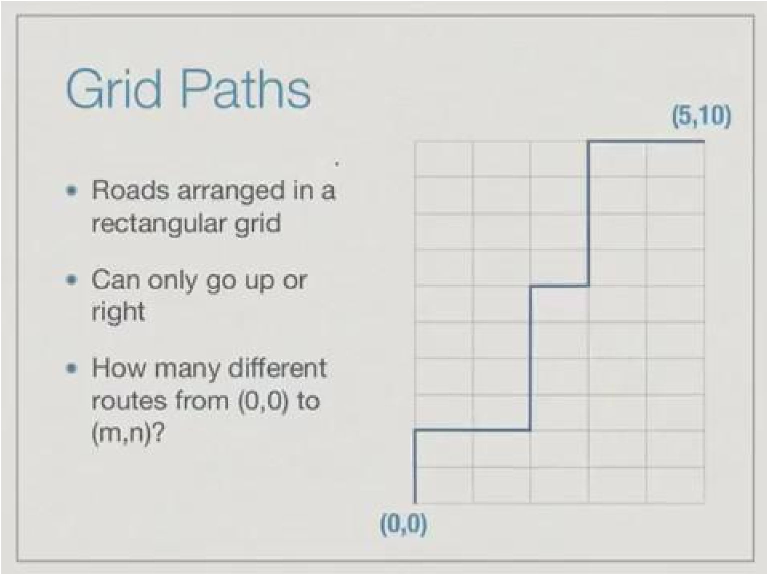



In the last lecture we looked at how to make iterative or inductive definitions more efficient than naïve recursion, and we saw memoization and dynamic programming as tools to do this.

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.

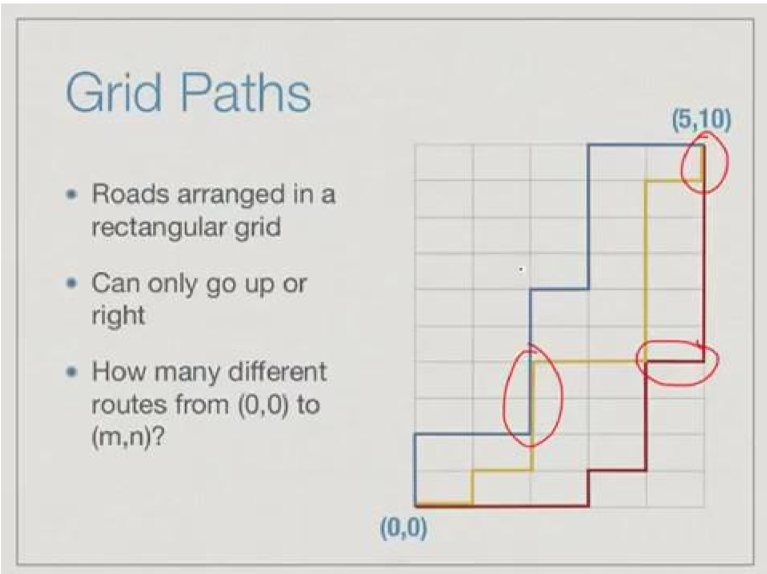
If these are roads the constraint that we have is that one can only travel up or right. So, you can go up a road or you can go right, but you cannot come down. This is not allowed. These are one way roads which goes up and right, and what we want to ask is how many ways there are to go from the bottom 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)



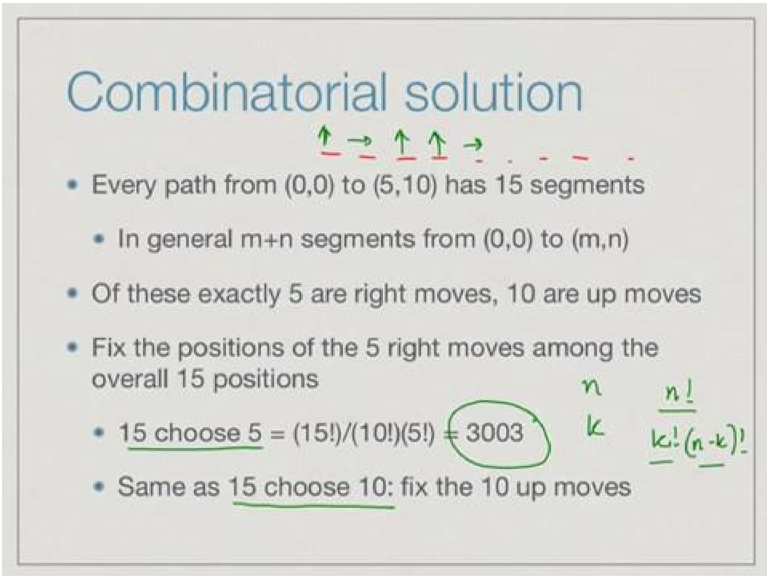

So, here is one path drawn in blue.

(Refer Slide Time: 01:32)




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)




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).

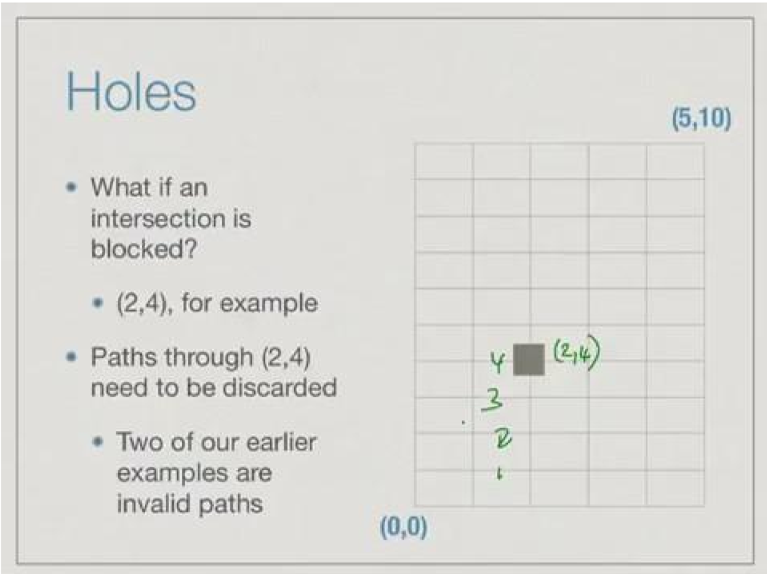
So, all we have to do since we know that these 5 steps are horizontal and 10 are vertical is to just demarcate which ones are horizontal and which are vertical. Now once we know which ones are horizontal we know what sequence they come in because the first horizontal step takes us from column 0 to column 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.

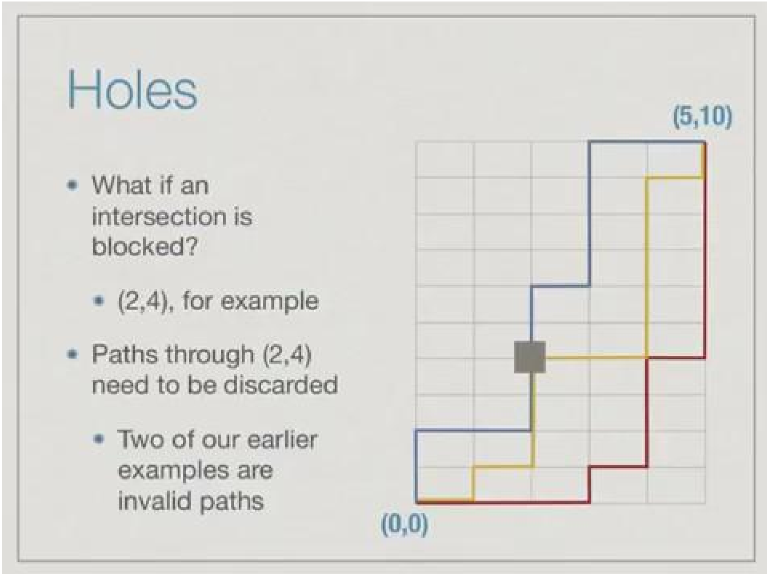
This k and n minus k basically says that 15 minus 5 is 10. So, we get a symmetric function in terms of k and n minus k. In this case we can apply this formula if you would like to call it that and directly get that the answer is 3003. There does not appear to be much to compute other than writing out large factorials and then seeing what the number comes.

(Refer Slide Time: 04:11)



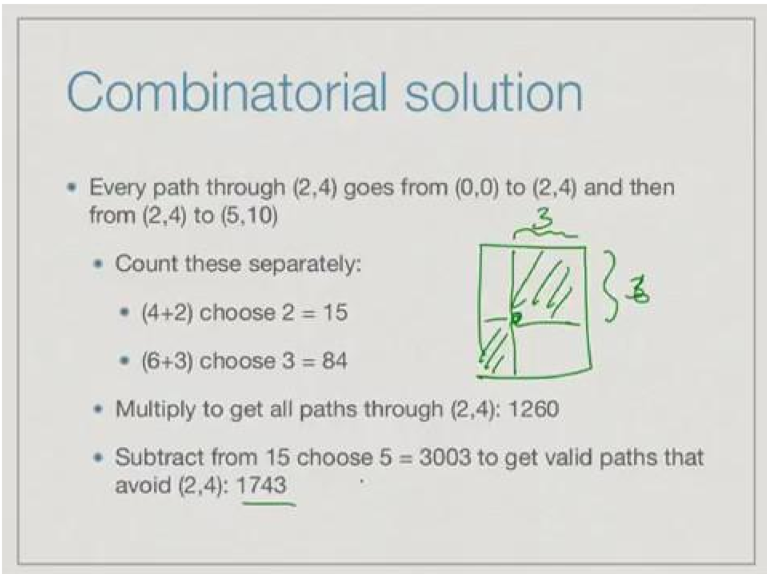

But the problem becomes more interesting, if we constrain it by saying that some of these intersections are blocked for instance, supposing there is some road work going on and we cannot go through this intersection (2, 4). This is the intersection 2 comma 4 second column and the fourth row counting 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)




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)



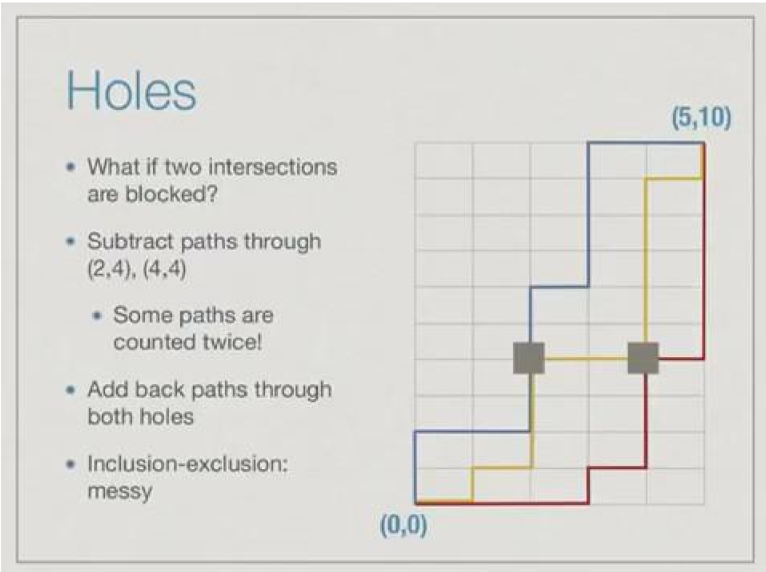

So, again we can use a combinatorial argument in order to be blocked a path must go to (2, 4) and then from (2, 4) to (5, 5). If we could only count how many paths go from (0,

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.

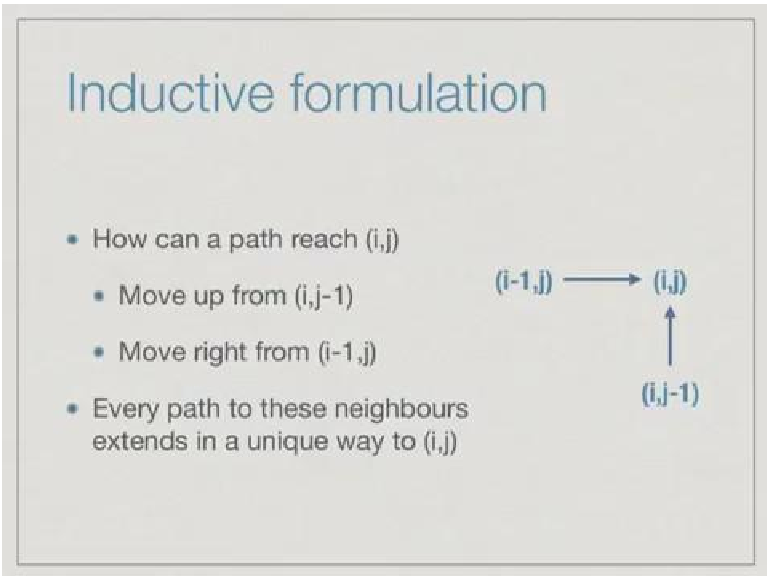
(Refer Slide Time: 06:32)




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).

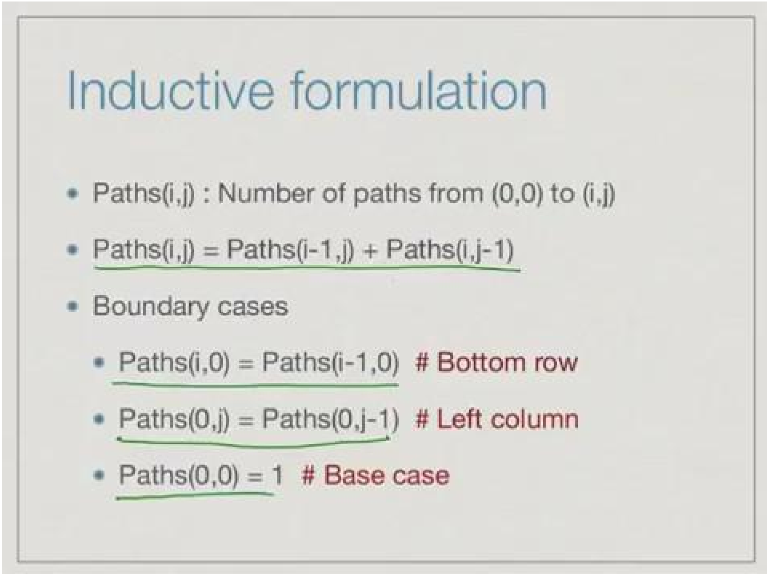
So, we need a third count we need to count paths which pass through both of these and make sure we do not double count them. So, one way is that we just add these back. This is something which is called in combinatorics inclusion and exclusion. So, when we have these overlapping exclusions, then we have 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)




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 it'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)

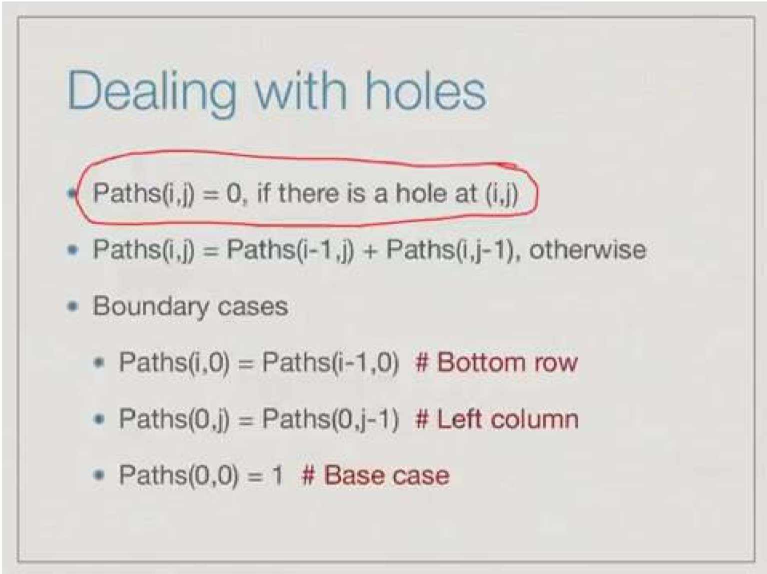



In other words if we say that paths(i, j) is the quantity we want to compute, we want to count the number of paths from (0, 0) to (i, j). These paths must break up into 2 disjoint sets those which come from the left which recursively or inductively if you prefer to say is exactly the quantity paths(i-1, 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 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, paths(i, j) is just the sum of paths(i-1, j) and paths(i, j-1). Then we need to of course, investigate the base cases: in this case the real base case is just 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.

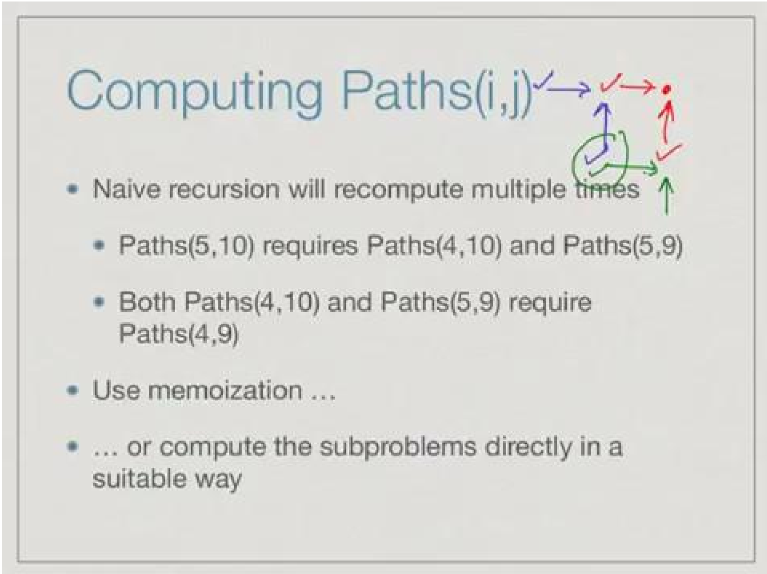
So, all the paths to (0,j) must be extensions of paths which have come from below up to (0,j-1). Similarly if you are on the bottom row there is no way to come from below because we are already on the lowest set of roads. So, paths(i, 0) can only come from the left, from paths(i-1, 0).

(Refer Slide Time: 09:56)




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 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 paths(i, j) at that point is 0.

(Refer Slide Time: 10:29)

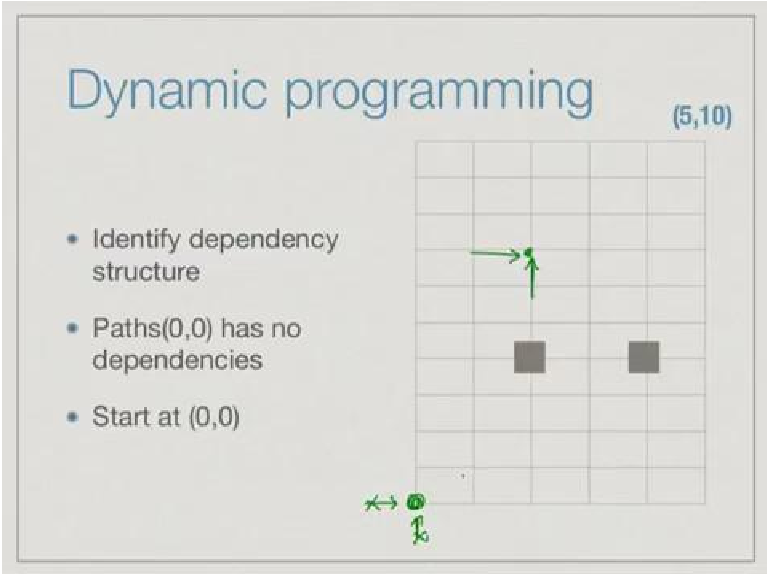



So, once again if we now apply this and do this using the standard translation from the inductive definition to a recursive program, we will find that we will wastefully recompute the same quantity multiple times for instance paths(5,10). If we have paths(5, 10), it will require me to compute this and 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).

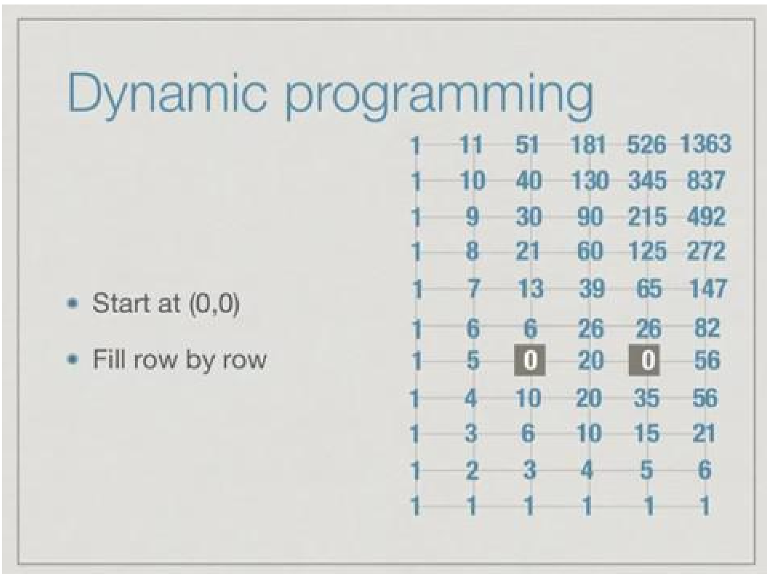
So, as we saw before we could use memoization to make sure that we never compute (i,j) twice by storing a table i comma j, and every time we compute a new value for i comma j we store it in the table and every time we look up, we need to compute one we first check the table, if it is already there we 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)




In general a node the value depends on things to it 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)



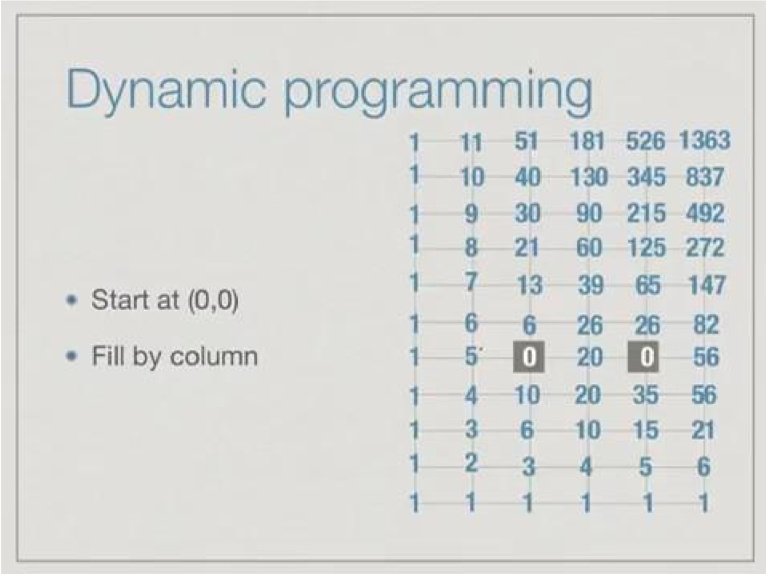

Remember that the base value at (0, 0) is one, and now once we have done this it turns out: you remember the road dependency, it said (i, 0) is (i-1, 0). So, we can fill up this, because this has only one dependency which is known now. In this way I can fill up the entire row and say that all along this 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.

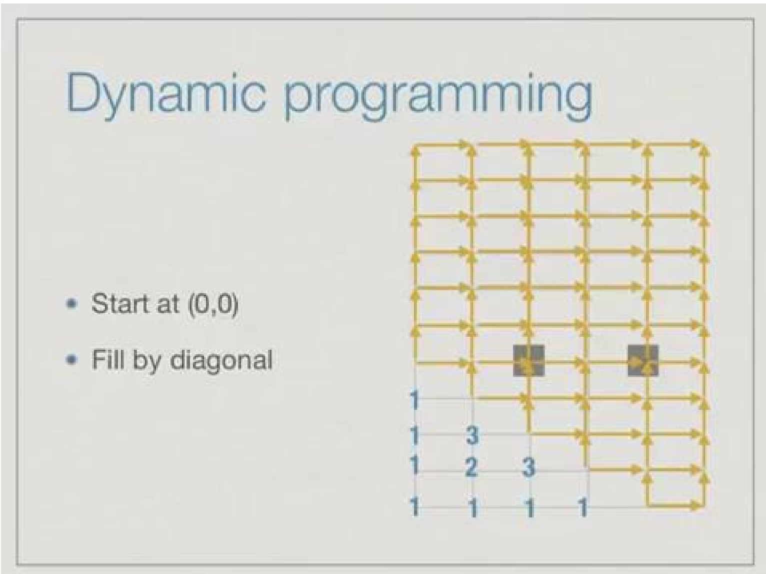
So, we can continue doing this row by row, and eventually we find look there are 1363 paths which avoid these two.

(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.

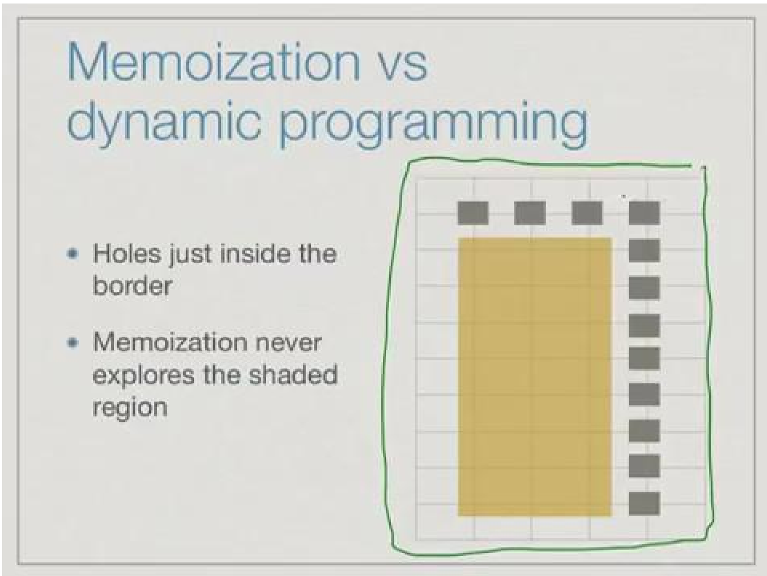
(Refer Slide Time: 14:39)




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)



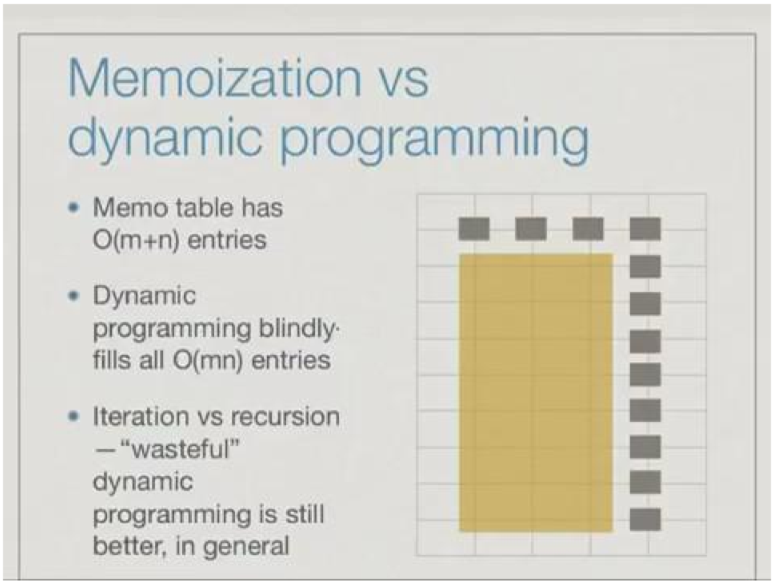

So, one small point, so we have said that we can use Memoization or we can use dynamic programming. One of the advantages of using dynamic programming is it avoids this recursive call. So, recursion we had mentioned earlier, also in some earlier lecture, comes with a price because whenever you make a 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 fib of n minus 1 or 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)




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.

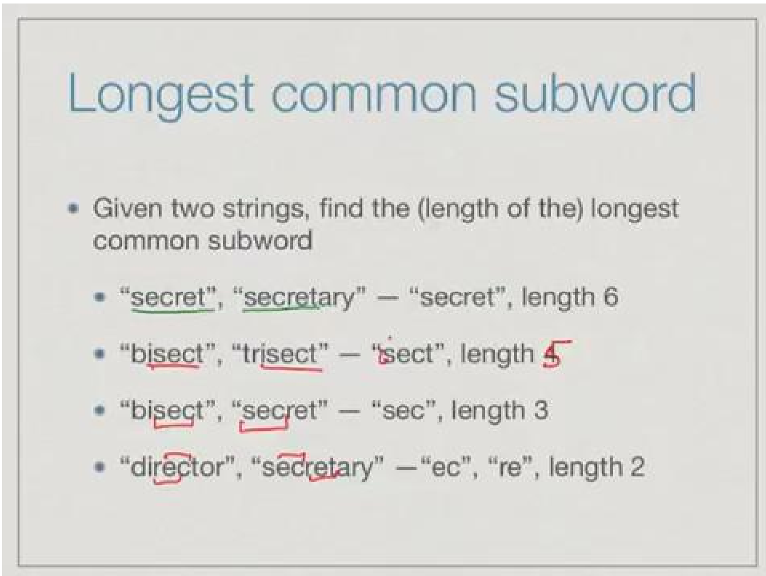
The reality is that these recursive calls will typically cost you much more, than the wastefulness of computing the entire table. In general even though you can analyze the problem and decide that memoization will result in many fewer new values being computed than dynamic programming. It is usually 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)



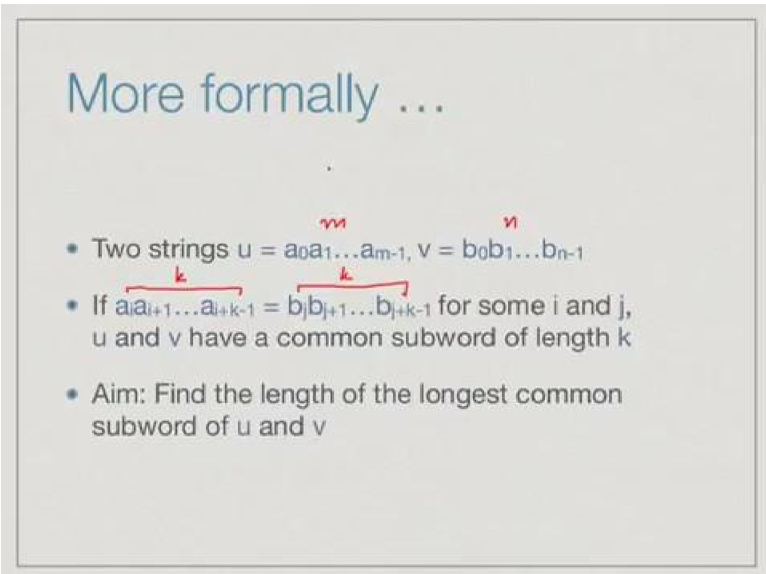

So, we are looking examples of problems where the main target is to identify the inductive structure and once you identify the inductive structure then the recursive structure of the program becomes apparent from which you can extract the dependencies and figure out what kind of memo-table you have and 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.

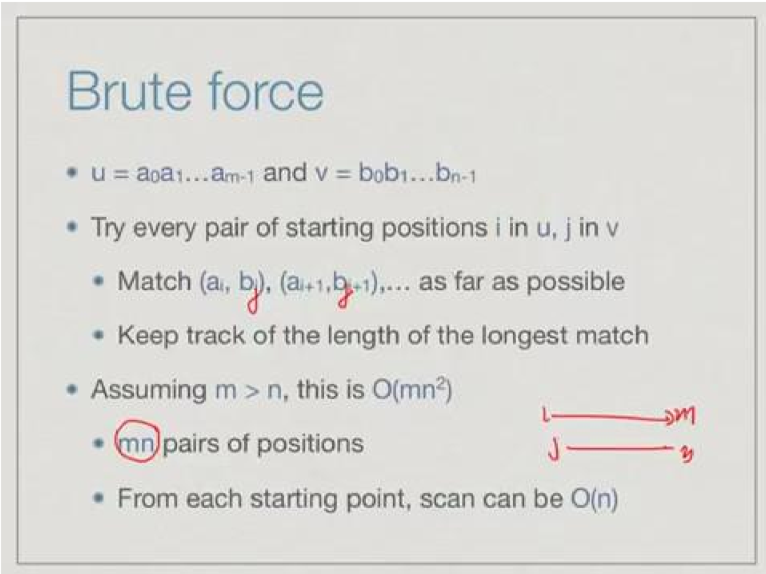
When we say subword, of course we do not mean a word in the sense; we just mean a sequence of letters. So, s e c is the longest common subword in has length 3 and if you have two very different words like director and secretary, sometimes you might have only small things, for example, here r e and e 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)




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)

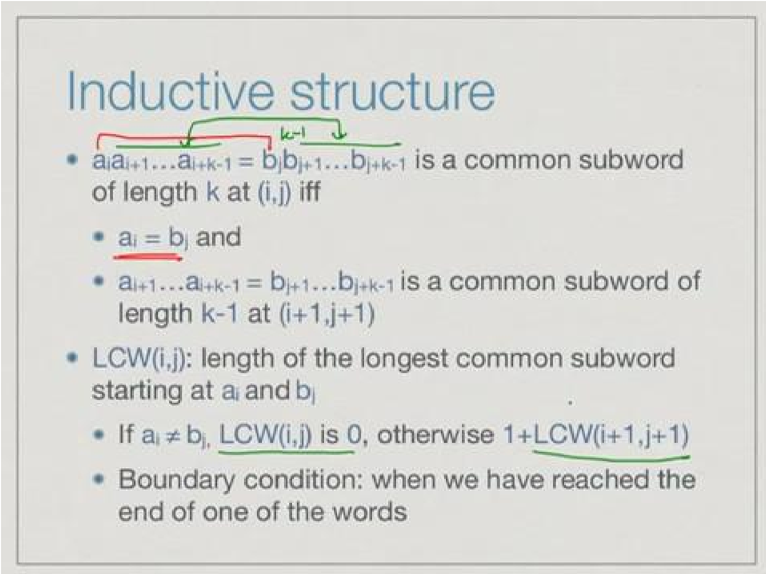



There is a brute force algorithm that you could use which is you just start at i and j in two word. In each word you can start a position i in u j in v and see how far you can go before you find they are not. So, you match a i and b j right. So, if a i and b j work then its fine. So, it should be b j 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)

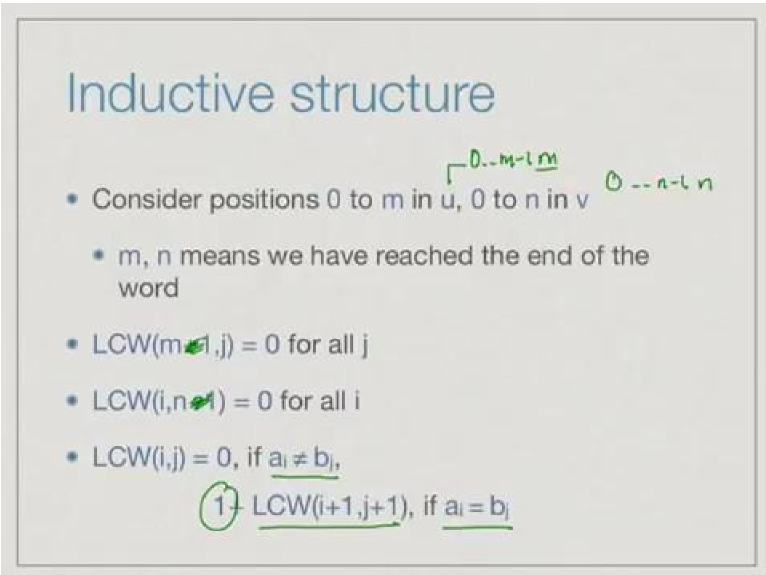



Our goal is to find some inductive structure which makes this thing computationally more efficient. So, what is the inductive structure? Well we have already kind of seen it when can we say that there is a commons subword starting at i j of length k, the first thing is that we need this a i to be the 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.

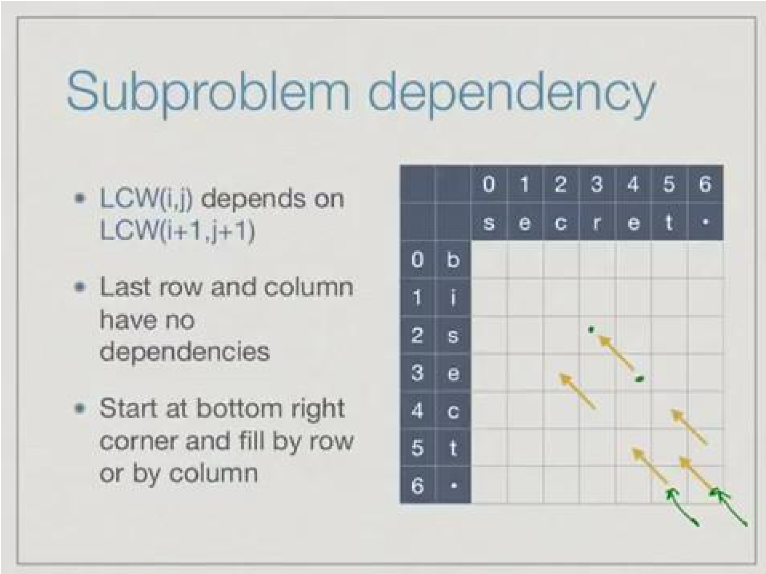
(Refer Slide Time: 05:58)




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 lcw of n comma j 0 is 0 because we have gone past the length u.

Similarly, if you reach n, but 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.

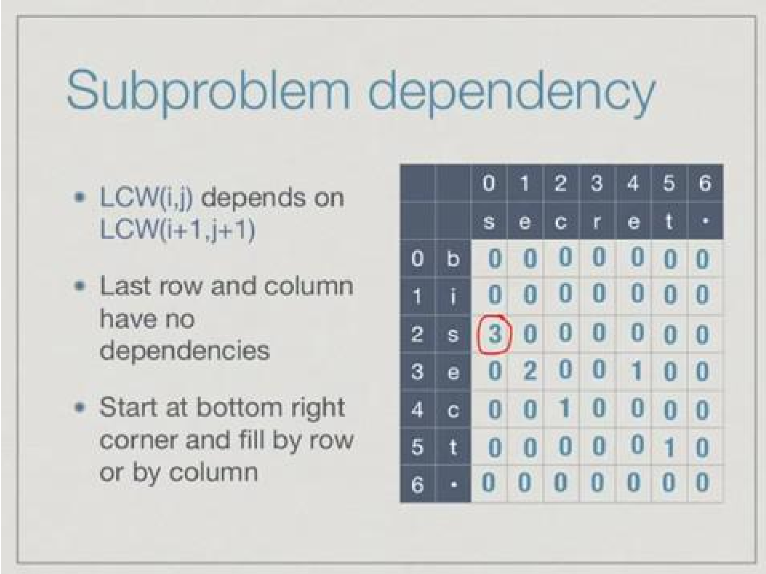
(Refer Slide Time: 07:18)




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 plus 1 j plus 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.

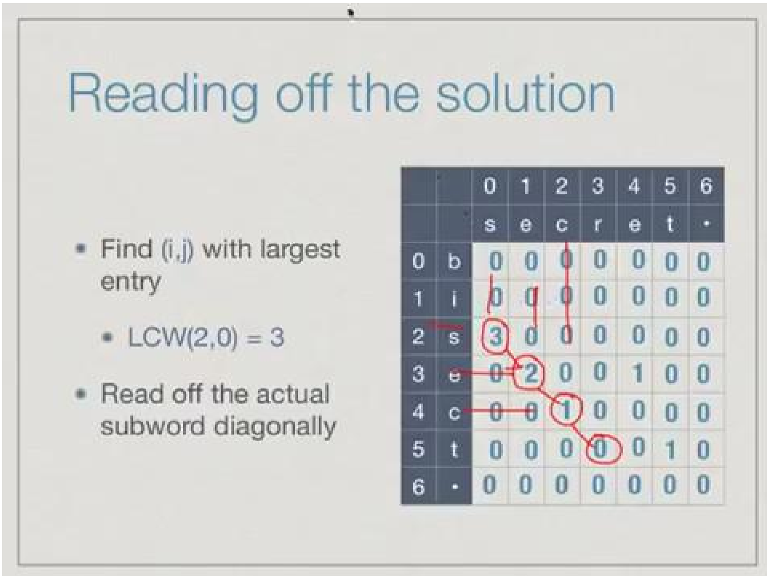
(Refer Slide Time: 08:08)




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.

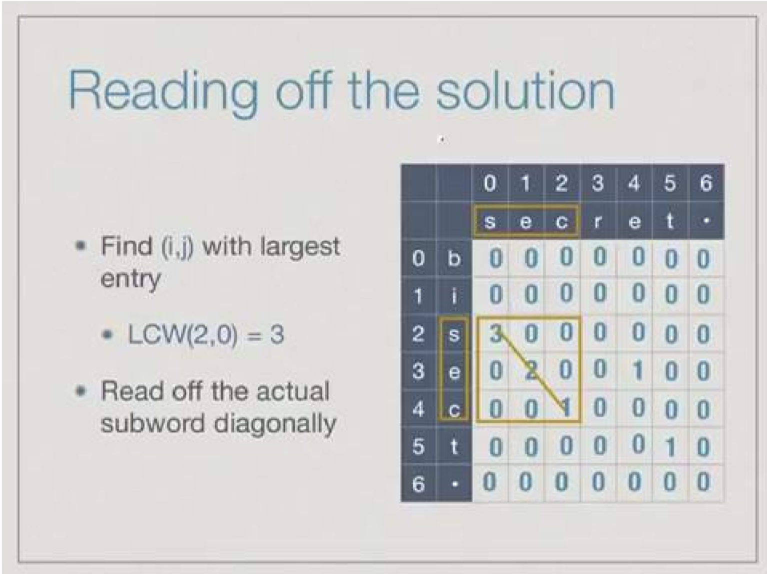
I do not even have to look at i plus 1 j plus 1, I directly says that 0 because is not there. So, in this way I can fill up this column. This is like our grid pack thing I can fill by column by column even though there the dependency was to the left and bottom and here the dependence is diagonally bottom 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)



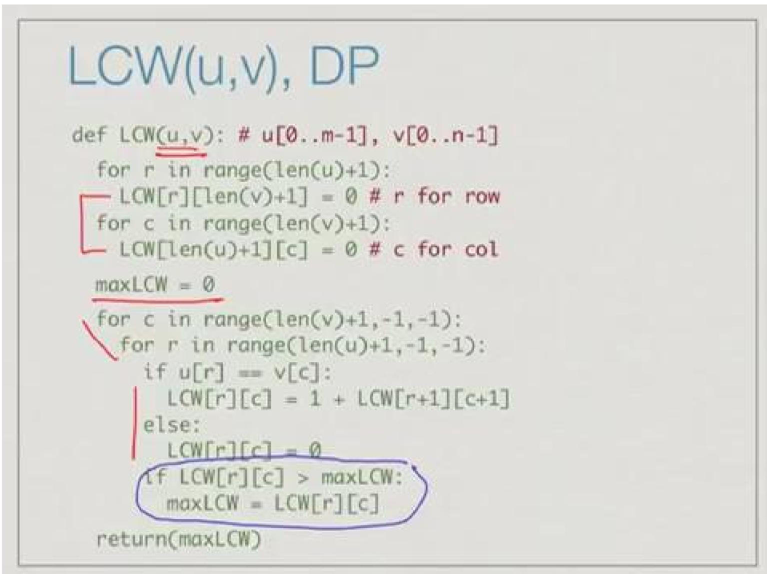

And now we said earlier that we are focusing on the length of the longest common subword not the word itself in the reason. We need to; we can afford to do that is because we can actually read of the answer once we have got the lengths. So, we ask ourselves why we did we get a 3 here. We got 3 here because 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)




If you walk down from that magical value, the largest value and we follow the sequence then we can read of and the corresponding row or column because they are the same, your actual subword which is the longest common subword for these two.

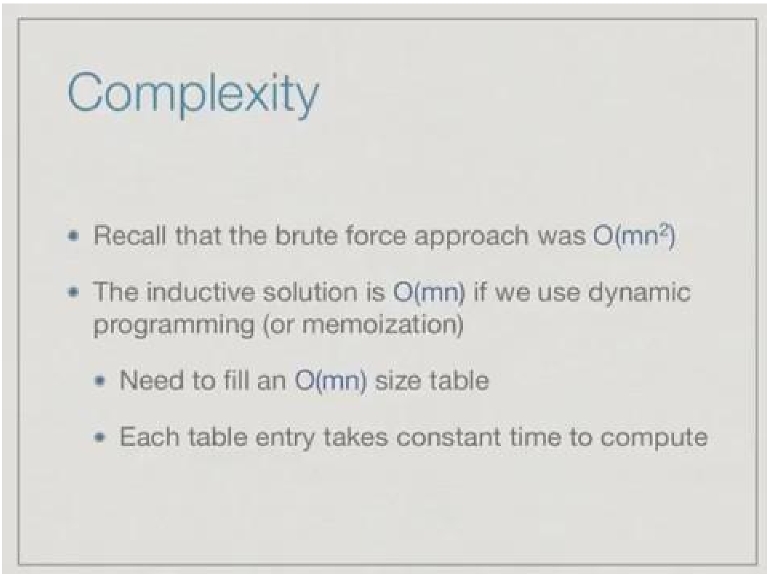
(Refer Slide Time: 10:10)



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.

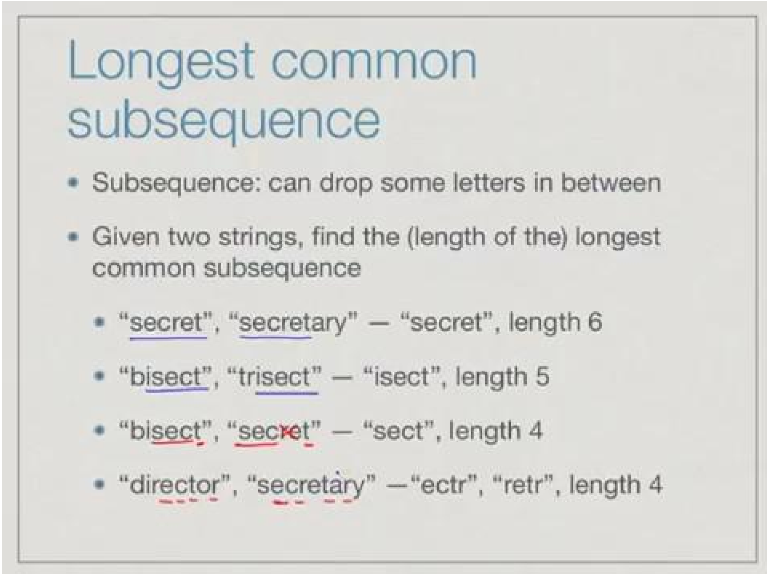
For each column, then for each row and that column you fill up the thing using the equation, if it is equal i to 1 plus otherwise as it say 0 and if i see and a new value this is the thing where I update if i see and new entry which is bigger than the entry which is currently the maximum, I update the 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)




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.

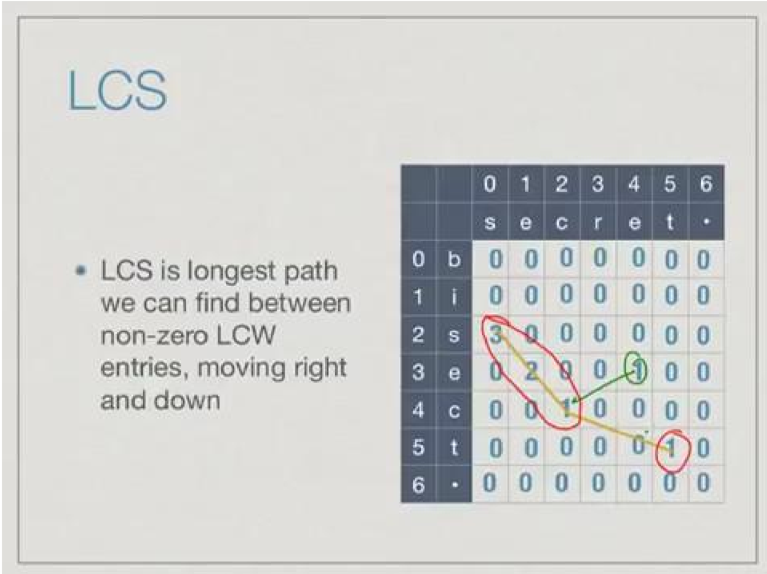
(Refer Slide Time: 11:43)




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?

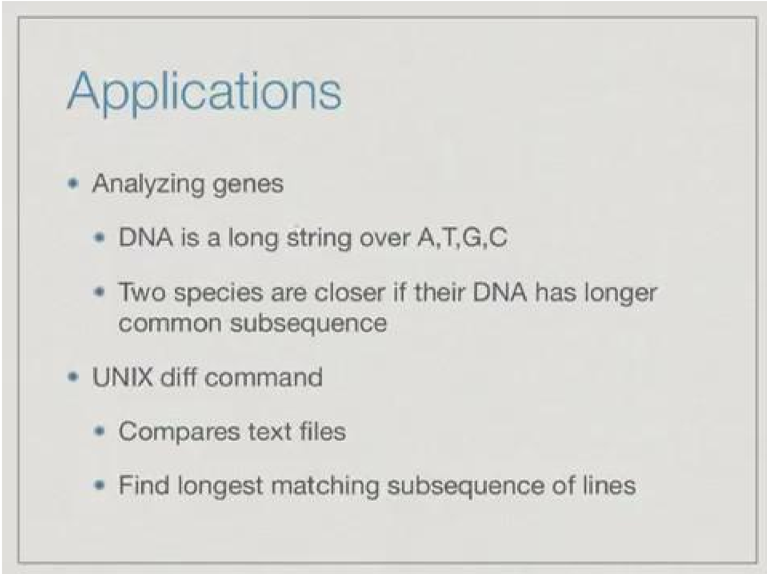
(Refer Slide Time: 12:54)




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)

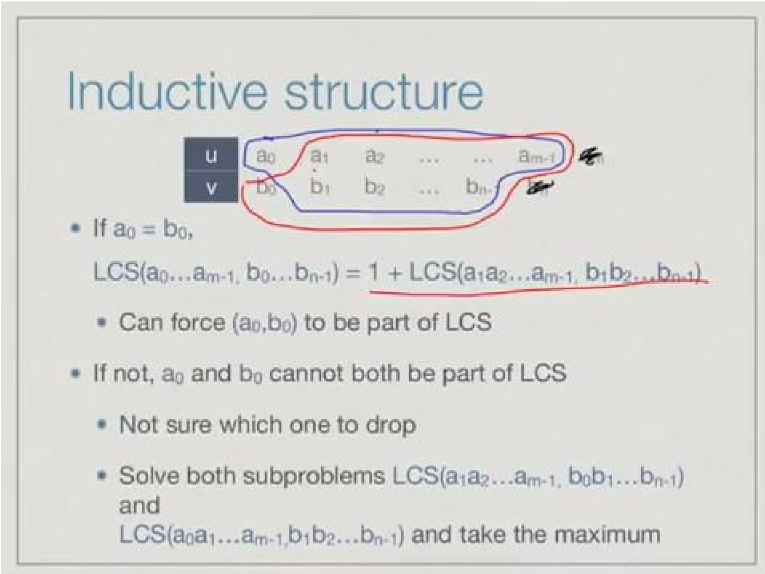



The motivations, well one of the big motivations for subsequence matching comes from things like genetics, for instance, when we compare the genes sequence of two organisms they are rarely equal. So, what we are looking for are large matches, where there might be some junk genes in between which you 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 individuals 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 go 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)




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.

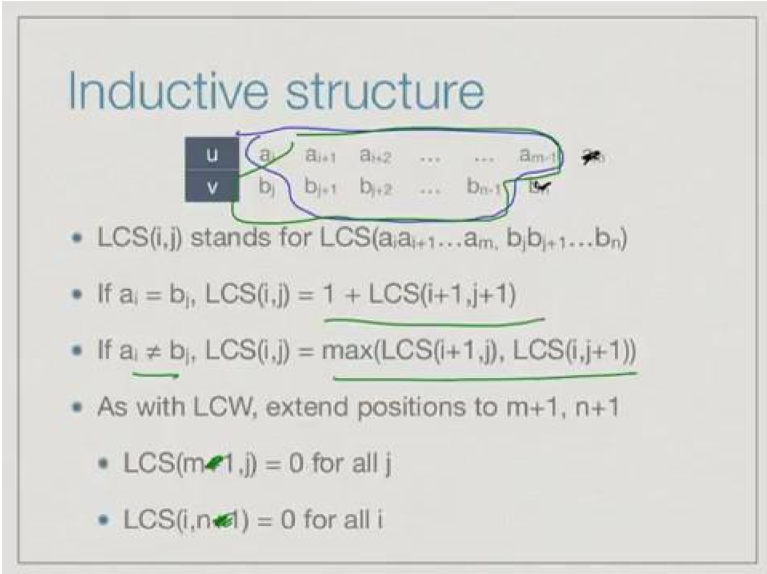
What is the subsequence, the subsequence actually is some kind of a matching, it says that you know it will say that this matches this and then this matches this, these are the same and this match is this and then this match is this and so on. Only thing is that these lines cannot over lap, they must be kept going from left to right without over lap. So, this kind of paring 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.

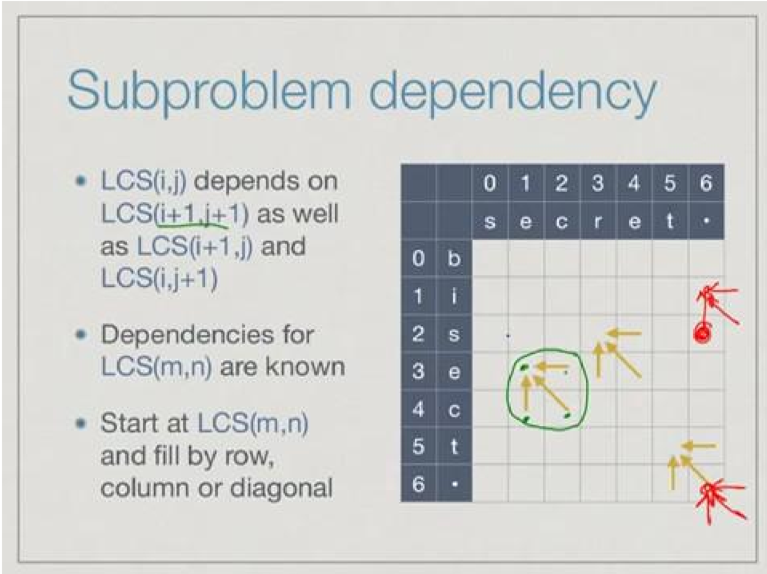
(Refer Slide Time: 18:45)




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.

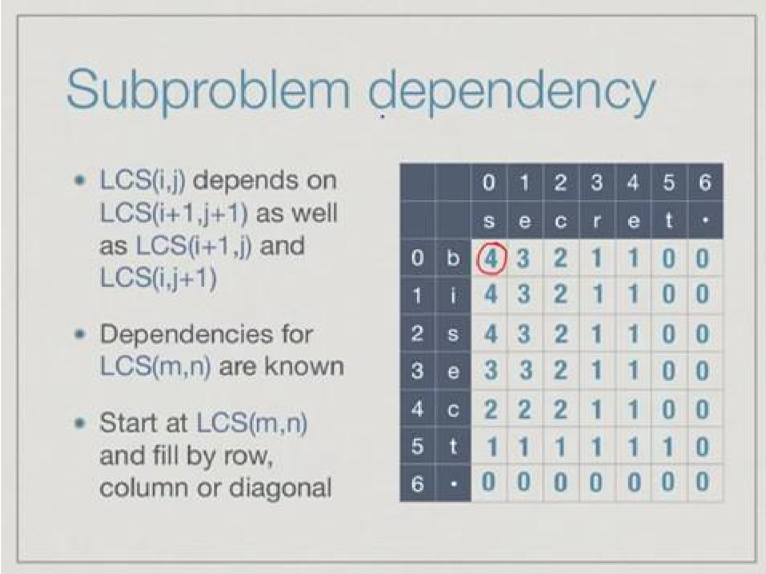
(Refer Slide Time: 20:01)




So, here the dependency is slightly more complicated because depending on the case, I either have to look at i plus 1 j plus 1 or I have to look at i plus 1 j or i j plus 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)

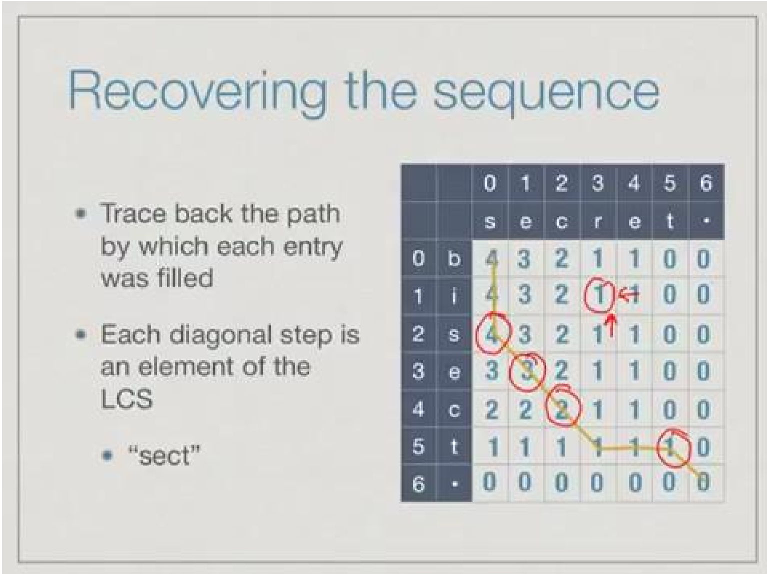



| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | | s | e | c | r | e | t | • |
| 0 | b | 4 | 3 | 2 | 1 | 1 | 0 | 0 |
| 1 | i | 4 | 3 | 2 | 1 | 1 | 0 | 0 |
| 2 | s | 4 | 3 | 2 | 1 | 1 | 0 | 0 |
| 3 | e | 3 | 3 | 2 | 1 | 1 | 0 | 0 |
| 4 | c | 2 | 2 | 2 | 1 | 1 | 0 | 0 |
| 5 | t | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 6 | • | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

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)



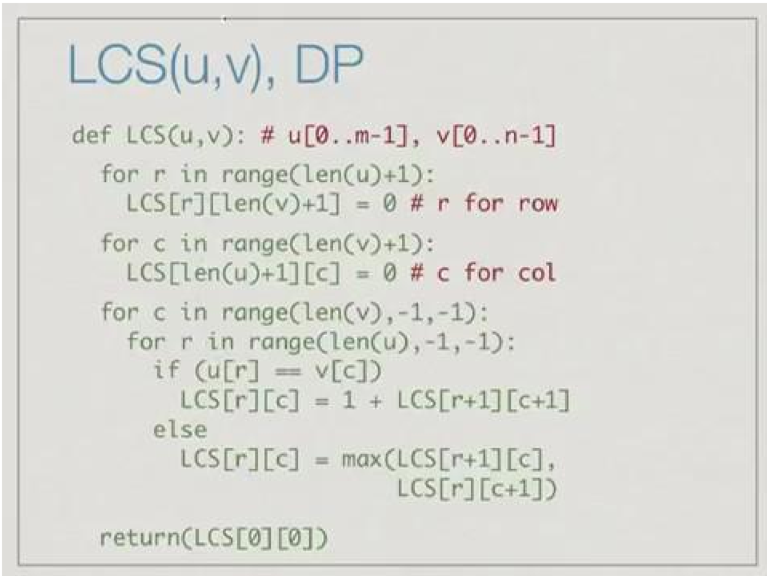

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| | | s | e | c | r | e | t | • |
| 0 | b | 4 | 3 | 2 | 1 | 1 | 0 | 0 |
| 1 | i | 4 | 3 | 2 | 1 | 1 | 0 | 0 |
| 2 | s | 4 | 3 | 2 | 1 | 1 | 0 | 0 |
| 3 | e | 3 | 3 | 2 | 1 | 1 | 0 | 0 |
| 4 | c | 2 | 2 | 2 | 1 | 1 | 0 | 0 |
| 5 | 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.

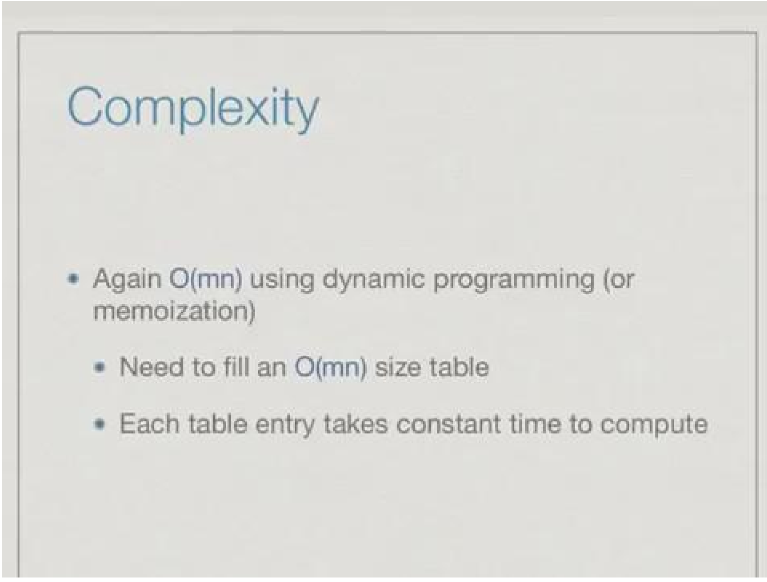
So, we can read of the diagonal steps along this kind a explanation of the longest number largest number we got and each diagonals step will contribute to the final solution, now there could be more than 1, because we have not got any example in this case, but sometimes the max could be one of in both 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)




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)



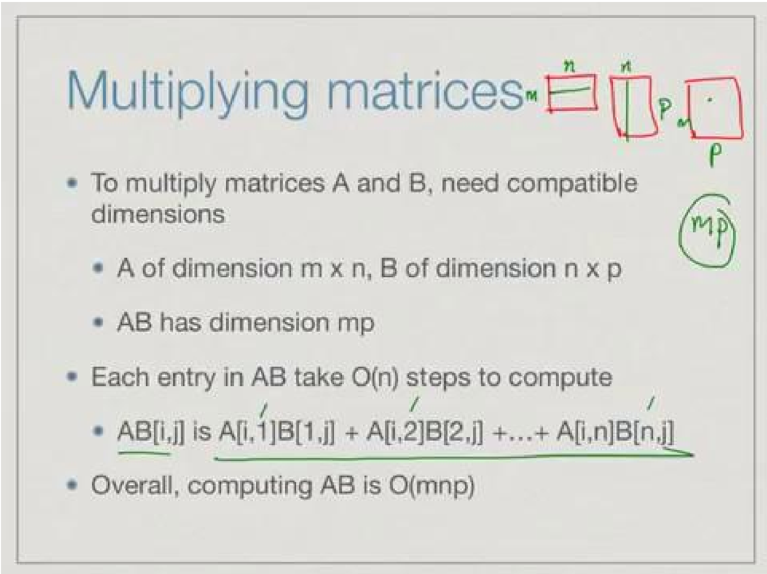

Just like the longest common subword, here once again we are filling in a table of size m times n. Each entry only requires you to look at most do three other entries. So, one to the right one to the bottom right in that the one below. So, it is a constant amount of work. So, m n entries constant amount 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.

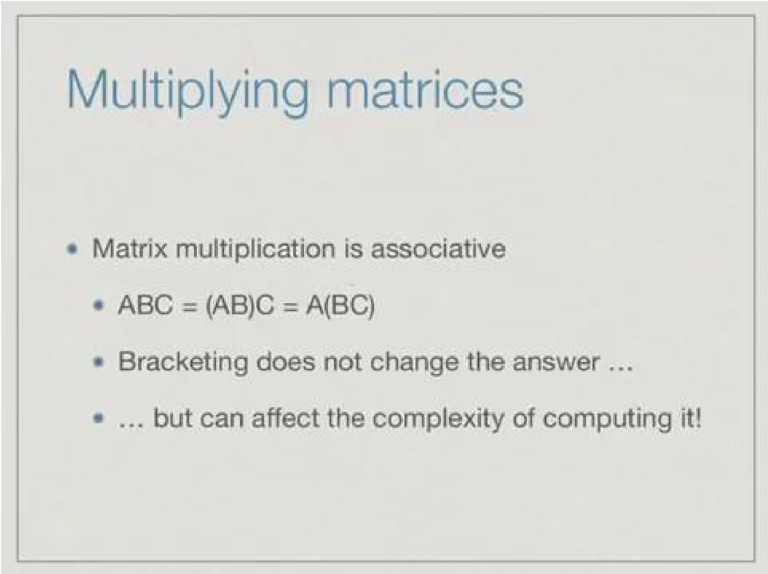
(Refer Slide Time: 00:02)




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.

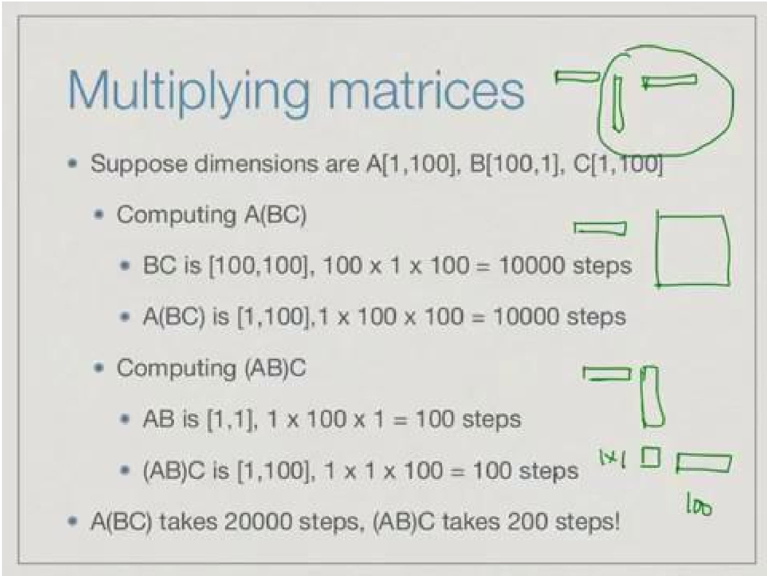
So, AB i j is this long sum and this sum has 1, 2, 3 up to n entries. Computing matrix multiplication for two matrices has a very straight forward algorithm which is a product triple nested loop which takes order m times n times p, if all of the dimension are the same this is an order n cubed algorithm. 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)




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)

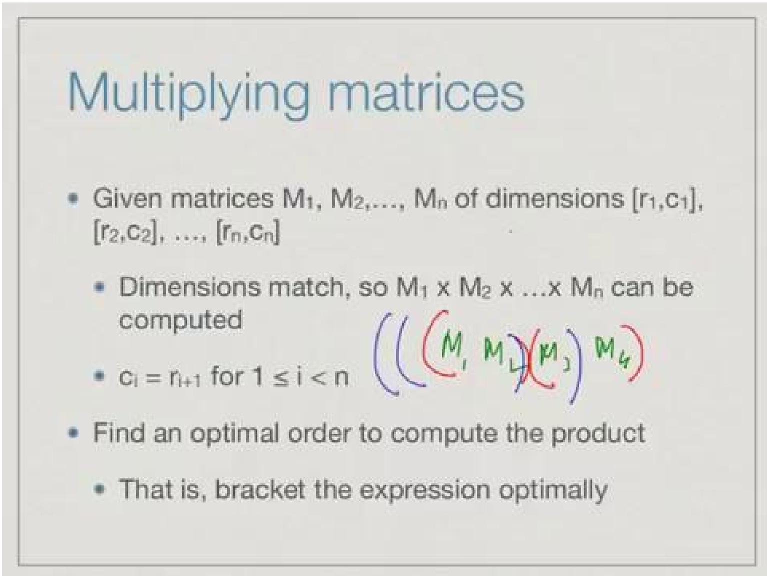



Suppose, we have these matrices A, B and C, and A and B have these columns and rows. A is basically got 1 by 100, A just has 1 row and 100 columns and B has a 100 rows and 1 column. And C has again 1 row and 100 columns. These are matrices which look like this. Now what happens is that when I multiply 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)



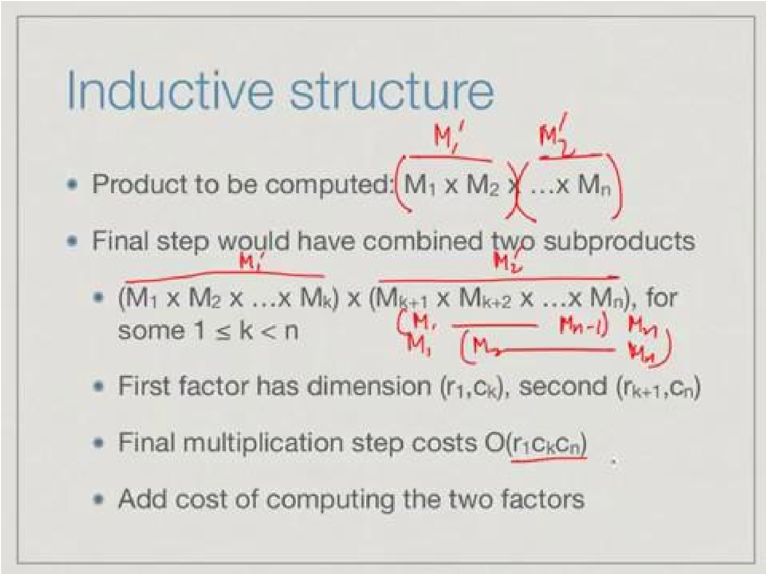

In general, we have a sequence M 1 to M n and each of them has some rows and columns, and what we are guaranteed is that each adjacent pair can be multiplied. So, r 1 c 1, r 2 c 2 the first two are such then c 1 is equal to r 2, the number of columns in the first matrix is equal to the number rows in 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 2 then I can combine that and do that combination with 3 or I can do M 1 2, M 3 4 and then do combination of M 1 2 multiplied by M 3 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)

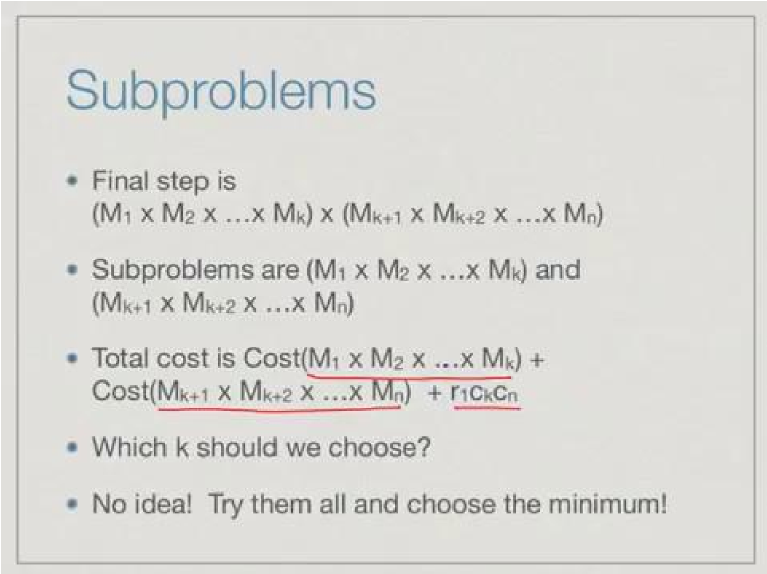



What is the inductive structure? Finally, when we do this remember we only do two at a time. At the end, we must end with the multiplication which involves some group multiplied by some group it must look like this, and must have this whole thing collapsing to some M 1 prime, and this whole thing collapsing 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 plus 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 doing 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 takes us to do M 1 prime how much time it will take us to do M 2 prime.

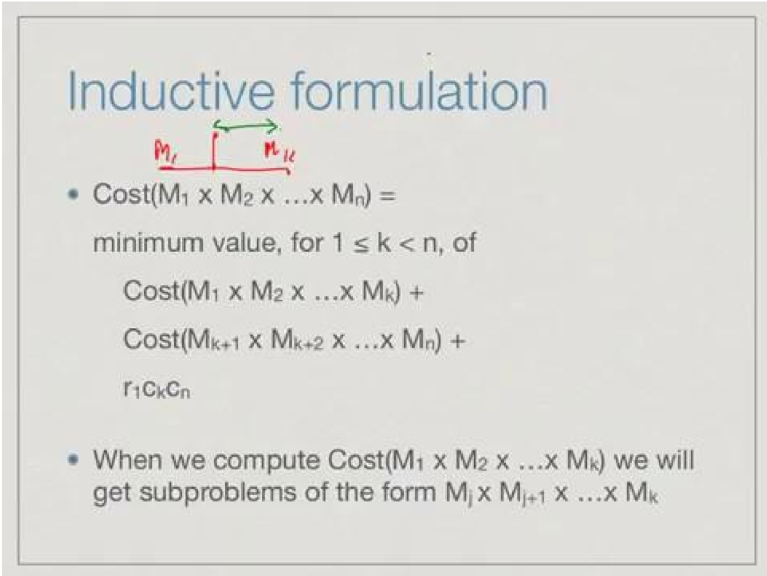
(Refer Slide Time: 07:31)




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.

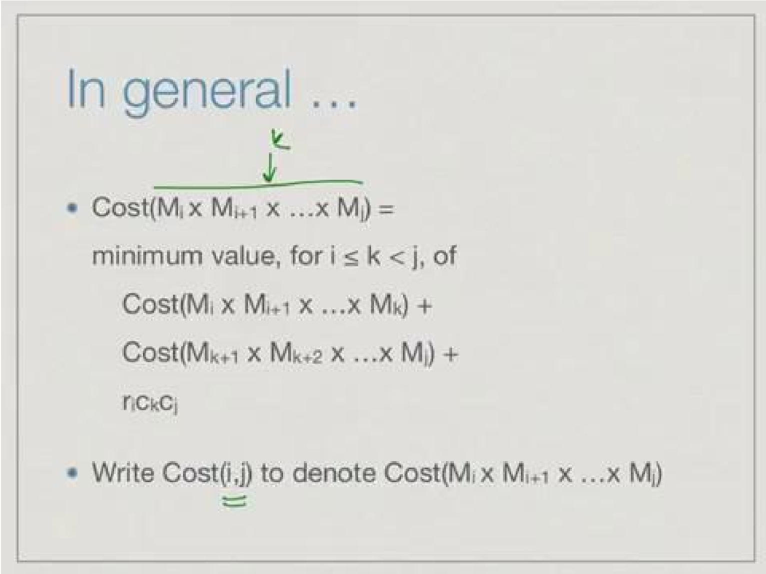
(Refer Slide Time: 08:37)




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.

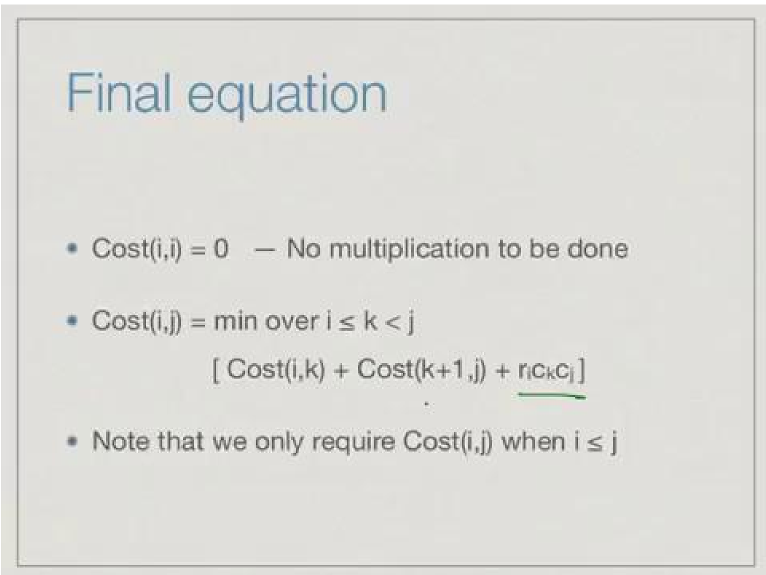
Then as before what will happens is that we will have to split this somewhere, so we will we will now end up having some segment which is neither M 1 nor M n, it starts at some M j and goes to M k. In general, we need to express this quantity for arbitrary left and right point, we cannot assume that the left hand point is 1; we cannot assume the right hand point is n right.

(Refer Slide Time: 09:25)



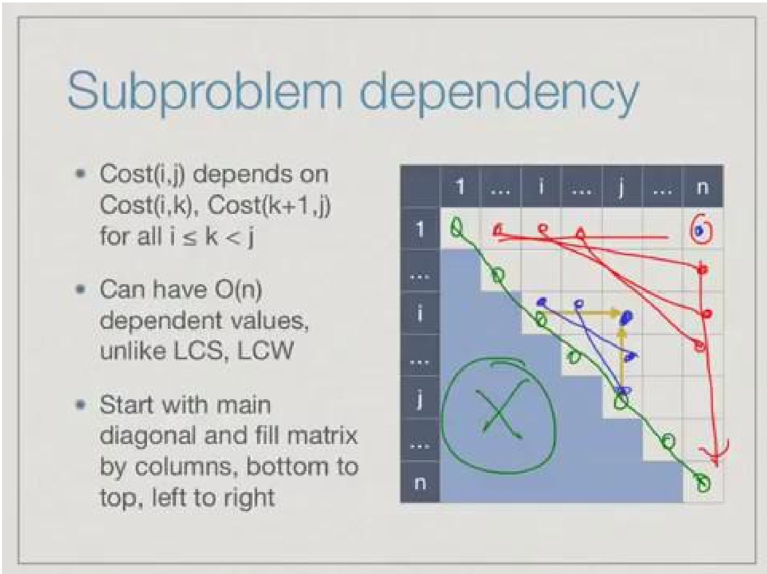

In general, if we have a segment from M i to M j, then we want the smallest value of among all the values of k from i to j minus 1, we want the minimum cost which occurs from computing the cost of M i to M k, and M k plus 1 to M j, and the cost of this final multiplication which is r i into c k into c j. This quantity we will write as cost i j cost i 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 cost i j, and we use this same recursive inductive definition choose the best.

(Refer Slide Time: 10:07)



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)

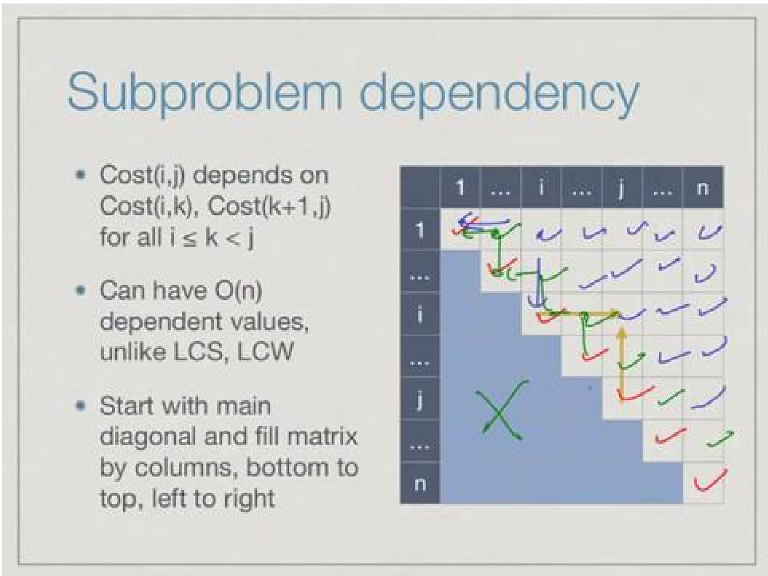



So, i is less than or equal to j, and we look at cost i j as a kind of matrix then this whole area where i is greater than j is ruled out. So, we only look at this diagonal. And the entries along the diagonal are the ones which are of the form i comma i, so all these entries are initially zero right. 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.

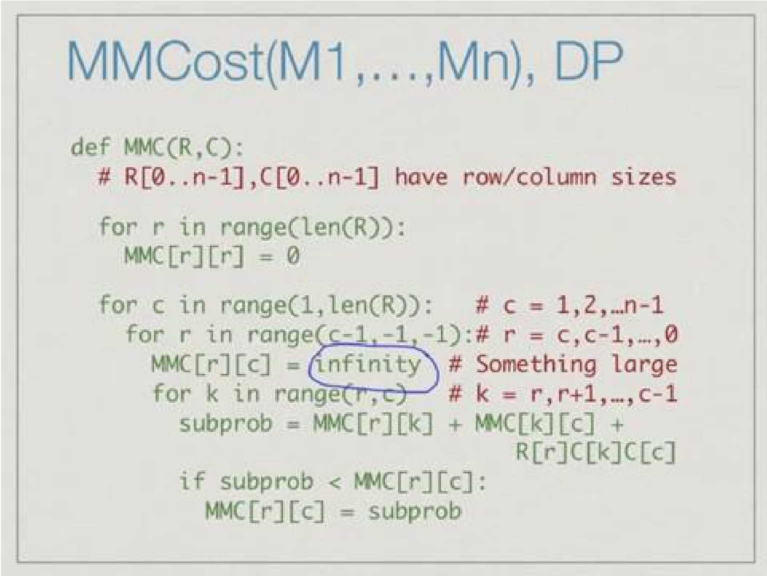
In general, if I have a thing there, I will say if I choose this entry as my k point then I must add this entry to get it up, and take this sum or I have to take this entry and add this entry and then add this and so on. In general, we could have order n values I need to compute for this I need to compute, I need all the values here and I need all the values here.

(Refer Slide Time: 12:40)



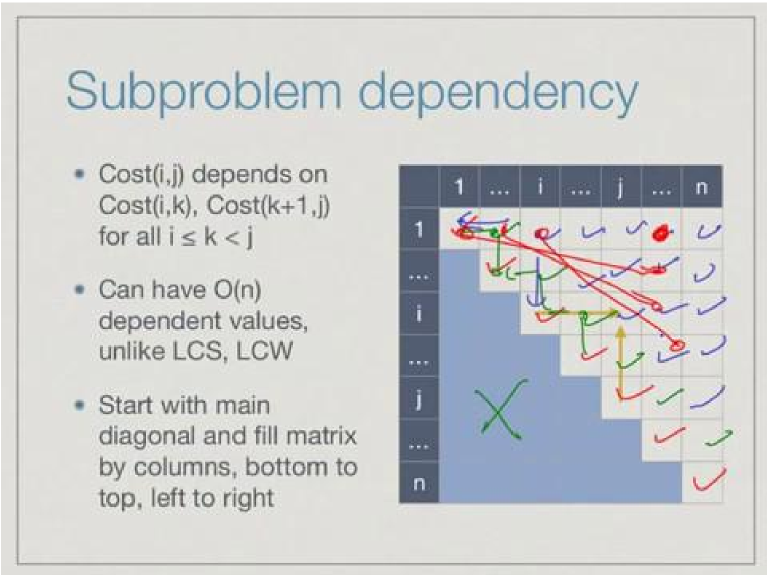

I need something to the left and to the bottom, but if I start in the diagonal then it becomes easy, because I can actually say that if I have initially these values filled in, this is the base case. Then to the left and below, I can fill in this because I know it is values to the left, I know this value 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)



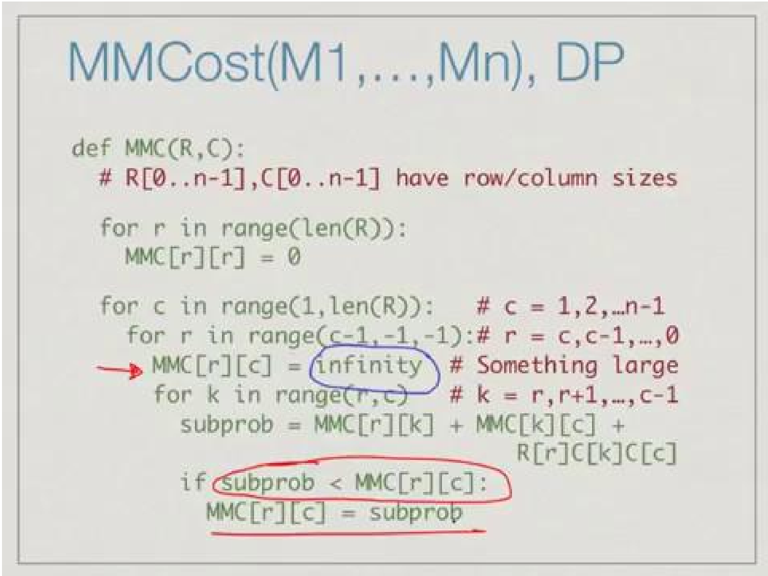

This is the code for this particular thing. So, you can go through it and just check the only thing that you need to notice that we have used some.

(Refer Slide Time: 13:31)




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.

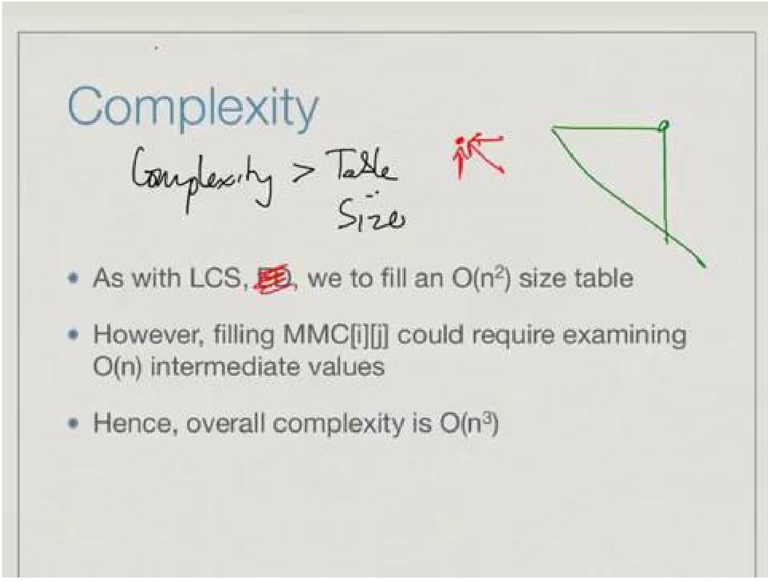
(Refer Slide Time: 13:57)




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)




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 it is 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 it is 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.

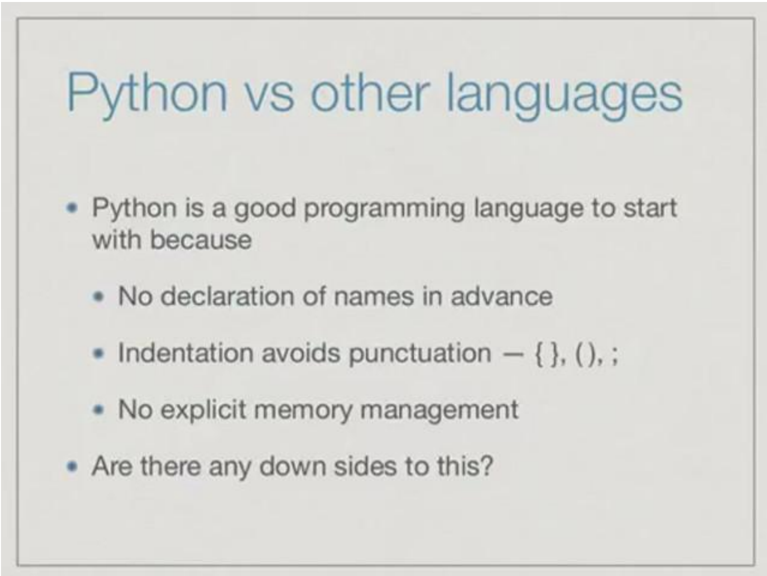
In all the examples we saw before, the Fibonacci we had a table which is linear in n and it took time linear in n to fill it. For the grid path and for longest common subsequence, we had tables, which are m by n, but it took only n by m time or m by n time to fill that. Here, we have a table which is 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)



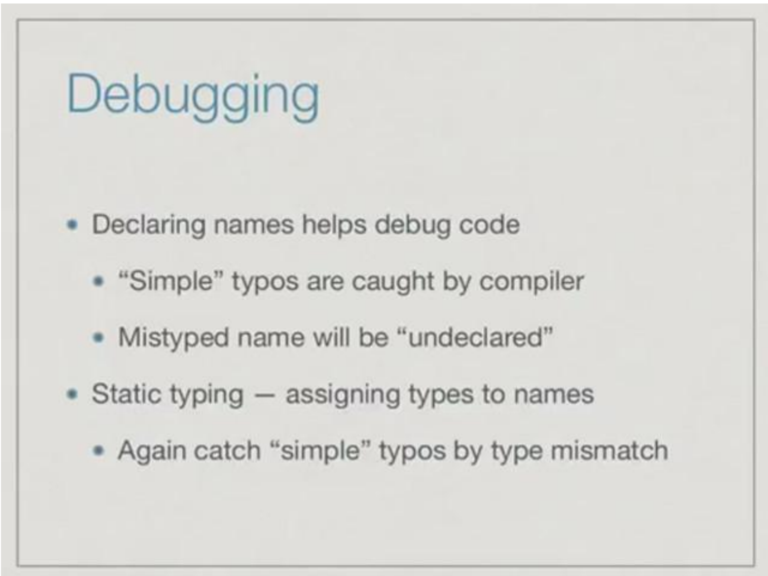

We have come to the last lecture of this course. So, instead of going into more features of Python of which there are many that we have not described. Let us take some time instead to reflect about Python as a language and compare it to some of the other languages which exist. Though you may not be familiar 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 it's 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)




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.

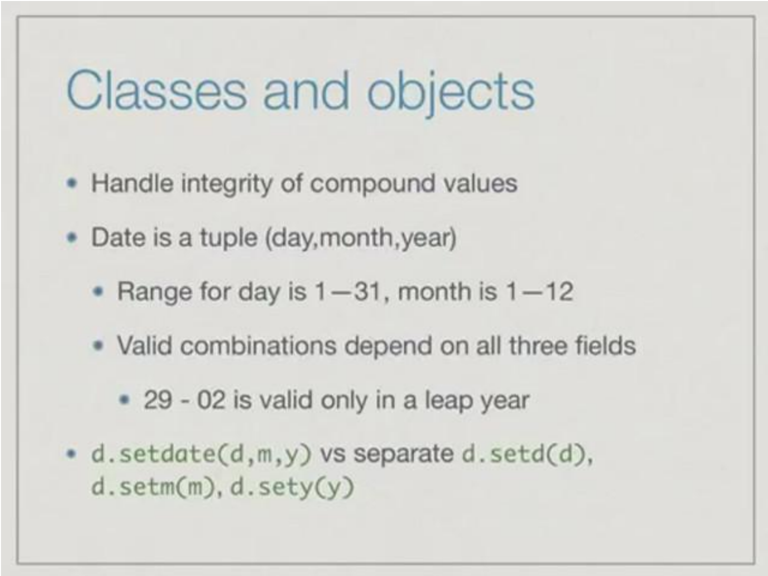
These kind of typos can be very very hard to find, and because you have this kind of dynamic name introduction with no declarations, Python makes it very difficult for you as a programmer to spot these errors. On the other hand if you declare these names in advance which happens in other languages like 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 typ ing. 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)




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 dot 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 dot x equal to 7.

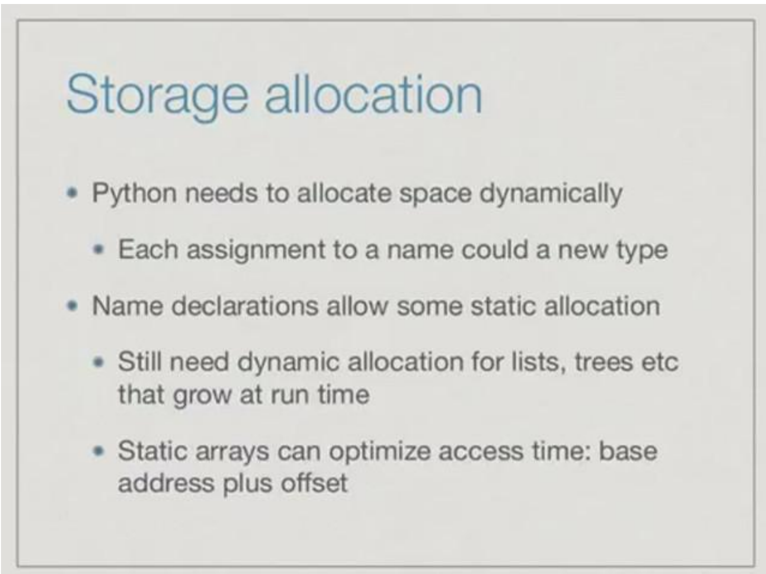
But in general this is the style that one would ideally advocate for object oriented programming - make all the internal names and variables private and only allow restricted access, so that they are used in the appropriate way and the use does not get compromised if the internal representation changes. If we move x, y to r, theta, get x and set x will still work, whereas p dot 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)



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.

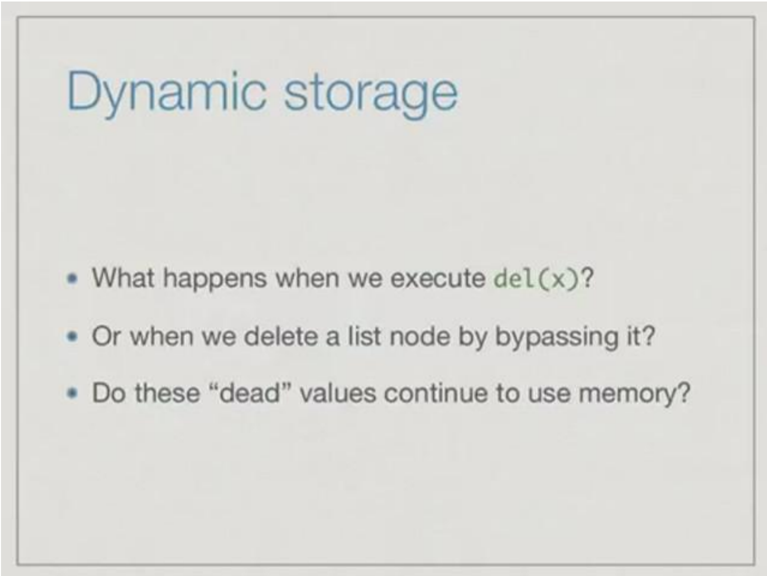
On the other hand if I have a static declaration, then if I say that i is an integer and j is an integer and k is an integer then the compiler can directly declare in advance some space in the memory to be reserved for i, j and k. Now this is particularly useful for arrays, we mentioned that in an earlier 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)




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)


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.

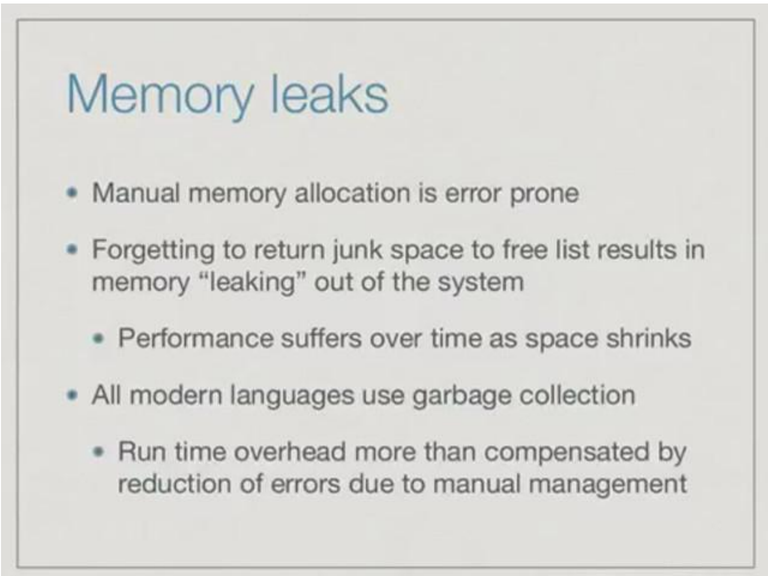
So roughly speaking how does garbage collection work? Well, what you do is you imagine that you have your memory and then you have somewhere some names, n, m, x, l and so on in your program and you know where these things point, this is somewhere here, this is somewhere here, this is somewhere here, 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, l 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)

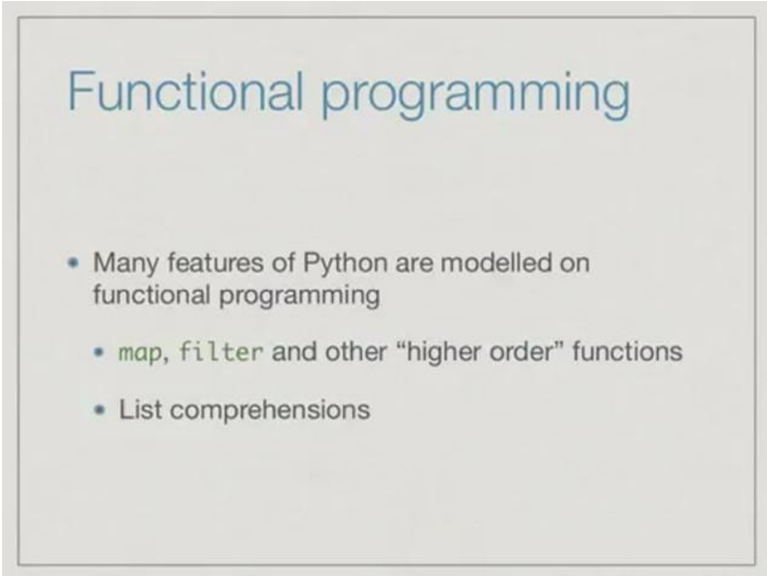


This is quite error prone. As you can imagine there might be just a simple case where a programmer writes a delete operation in a tree or a list and forgets to give the memory back. So, this means that every time something is added and removed from a list or a tree, in such a program, that thing will 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 refering to the fact that memory that has been allocated to a program is not being properly de allocated 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 de-allocated.

(Refer Slide Time: 21:58)




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.

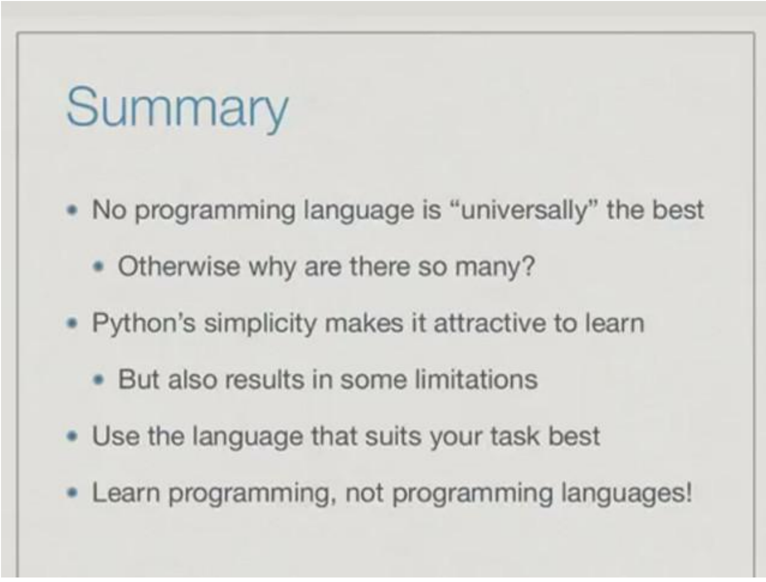
Functional programming is something which tries to take the kind of inductive definitions that we have seen directly at heart and just use these as the basis for computation. So, you will directly specify inductive functions in a declarative way what to conclude, so here is a typical declaration. For example, this is a Haskell style declaration. So, the first line with this double colon says that factorial is a function and this is it's 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 it's own uses and features.

Python has actually borrowed some of it's 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)




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