PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Prof. Madhavan Mukund
Computer Science & Engineering
Chennai Mathematical Institute

INDEX

Week 5

Week 6

Week 7

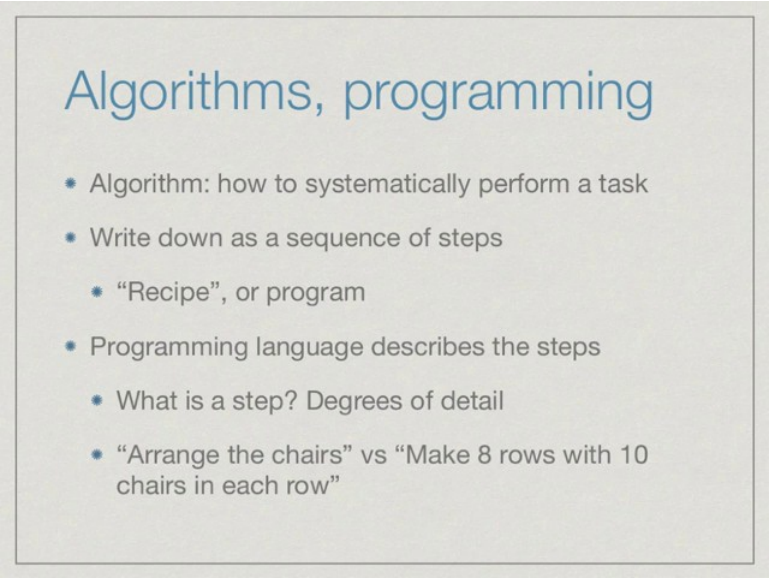
Week 8

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)



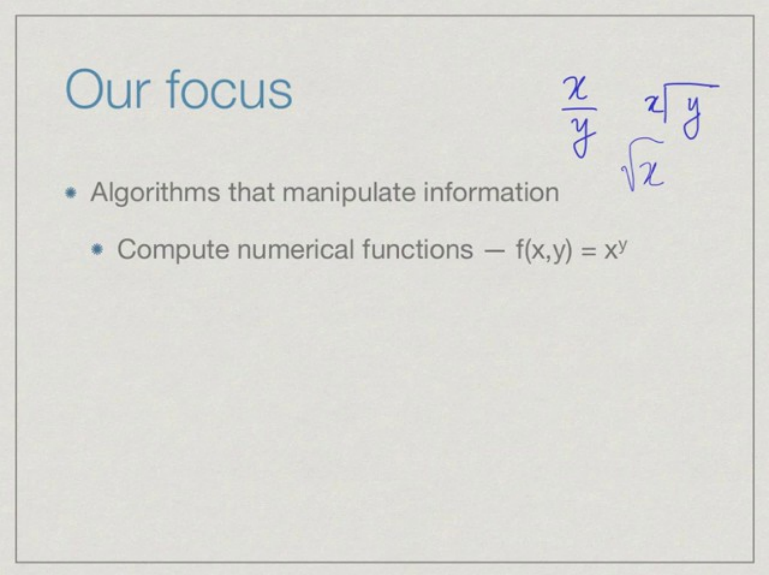
Let's start with the basic definition of what we mean by an algorithm and what programming is. As most of you probably know, an algorithm is a description of how to systematically perform some task. An algorithm consists of a sequence of steps which can we think of as a recipe in order to achieve something. 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.

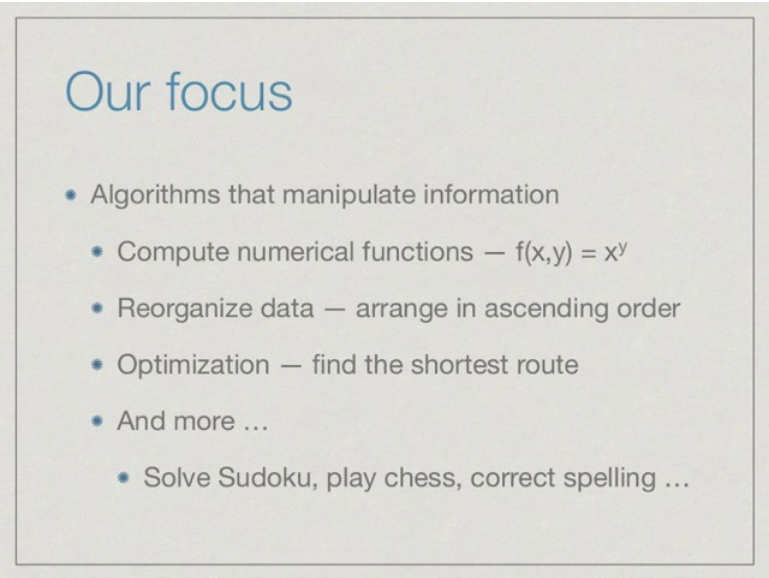
(Refer slide Time: 02:44)



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)

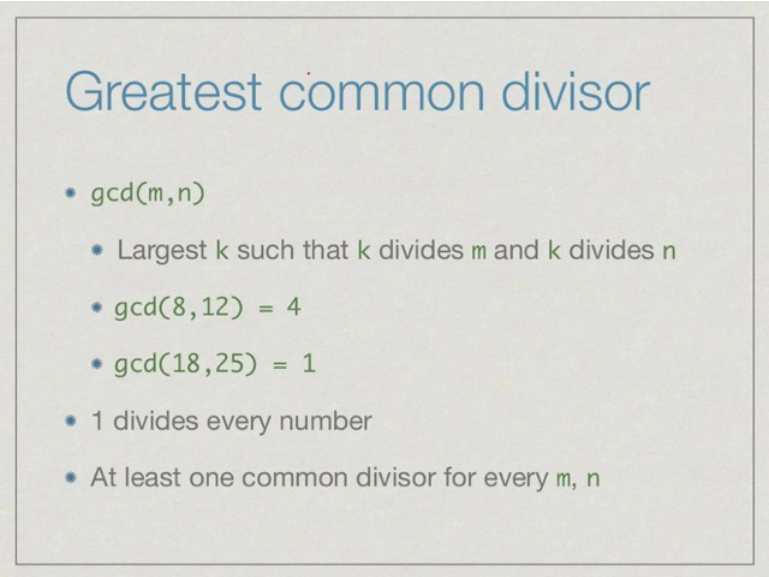


But all of us who have used computers know that many other things also fall within the realm of computation. For instance, if we use a spreadsheet to arrange information and then we want to sort of column. So, this involves rearranging the items in the column in some order either in ascending order or 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)



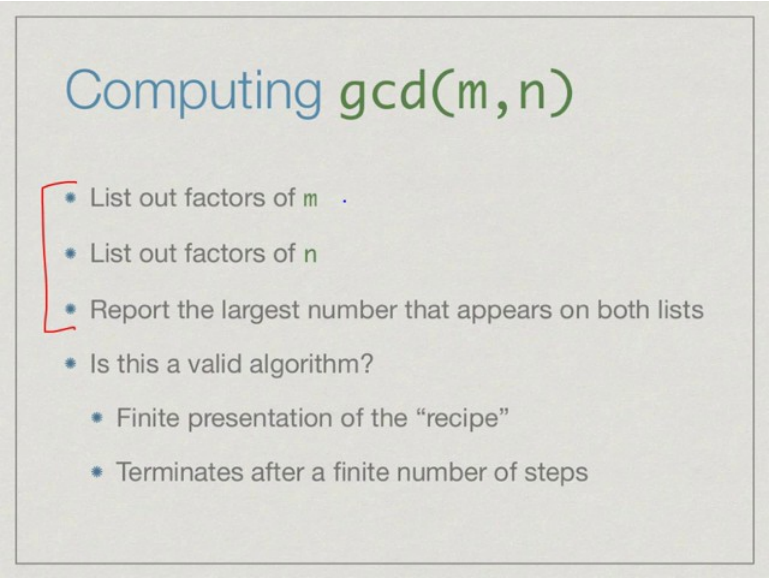
So to illustrate this let us look at the function which most of us have seen and try to understand the algorithmically. So, the property that I want to compute is the greatest common decide divisor of two positive integers m and n. So, as we know a divisor is a number that divides. So k is a divisor 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)



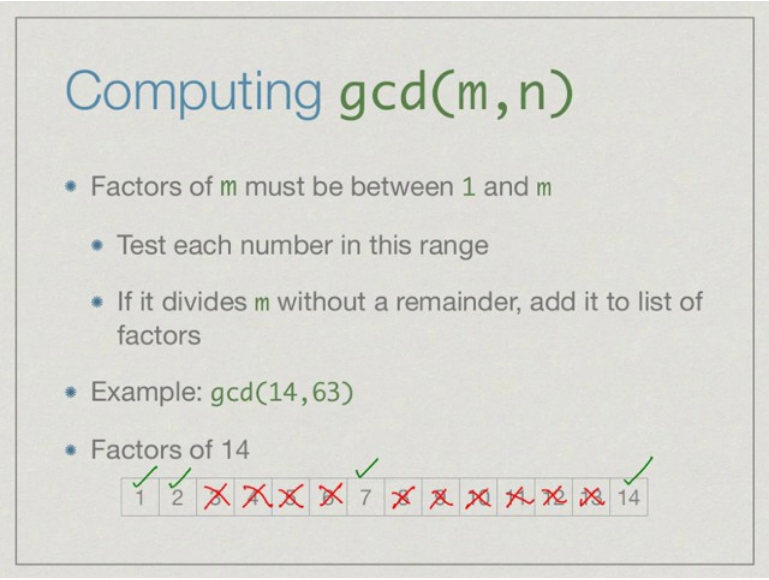
So, how would one go about computing gcd of m, n? So, this is where we come to the algorithmic way, we want to describe the uniform way of systematically computing gcd of m n for any m or any n. So, here is a very simple procedure. It is not the most efficient; we will see better once as we go along. 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)

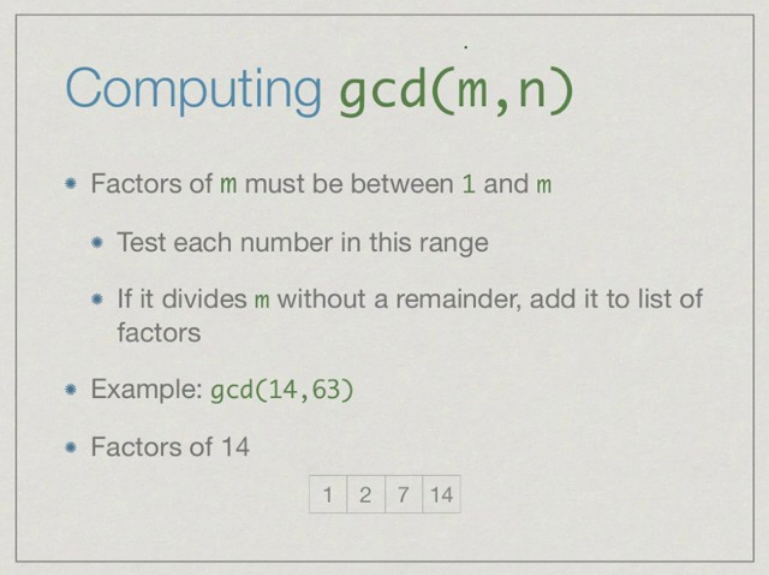


To argue that this process is well defined all we need to realize is that the factors of m must be between 1 and m. In other words, we although there are infinitely many different possibility as factors we don't have to look at any number bigger than m, because it cannot go into m evenly. So, all we 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.

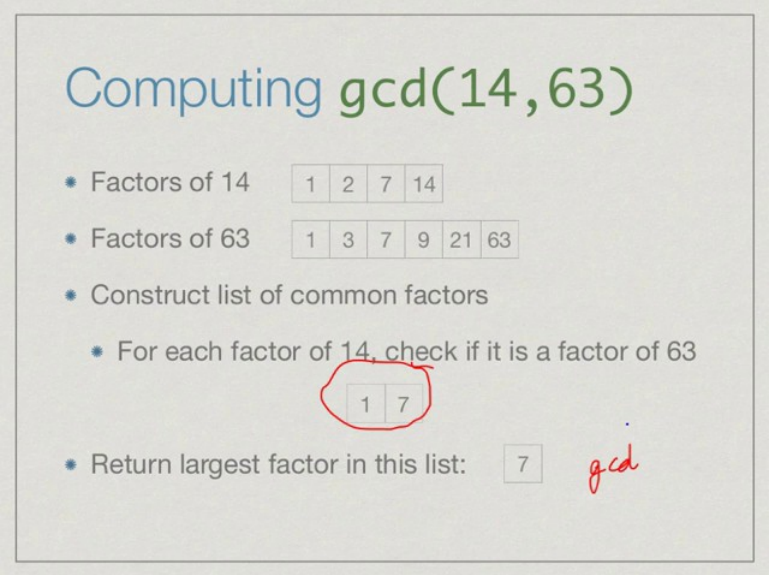
And finally, we find that the only other factor left is 14 itself. So for every number m - 1 and m will be factors and then there may be factors in between.

(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

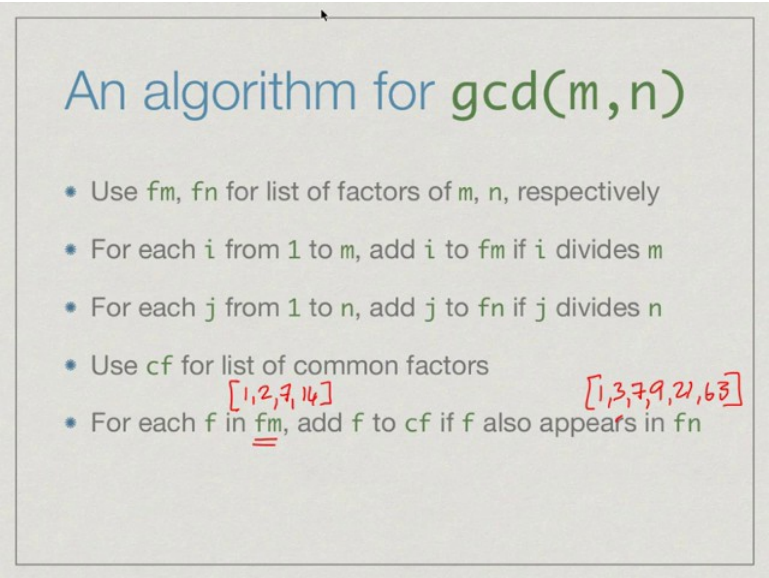
ones divide. So, again we will find that 1 divides, here 2 does not divide; because 63 is not even, 3 does divides, then we find a bunch of numbers here, which do not divide. Then we find that 7 divides, because 7 9's are 63. Then again 8 does not divides, but 9 does. Then again there are large gap of 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.

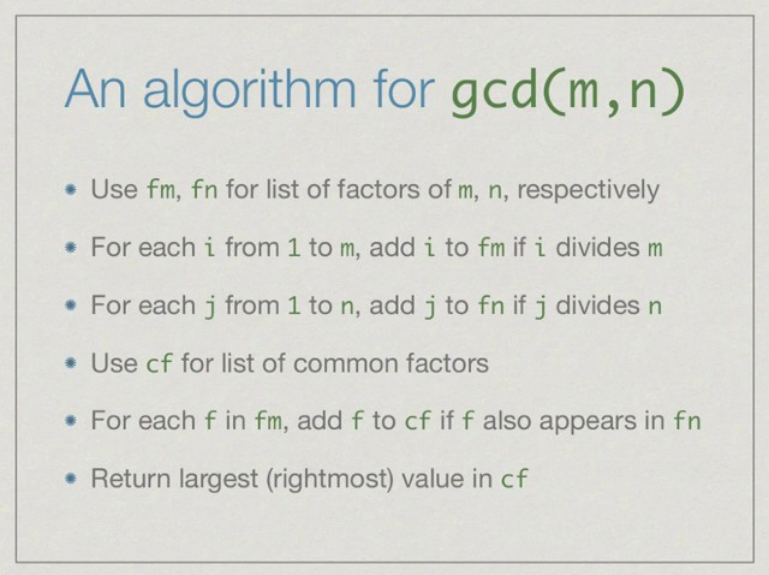
(Refer slide Time: 14:20)



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.

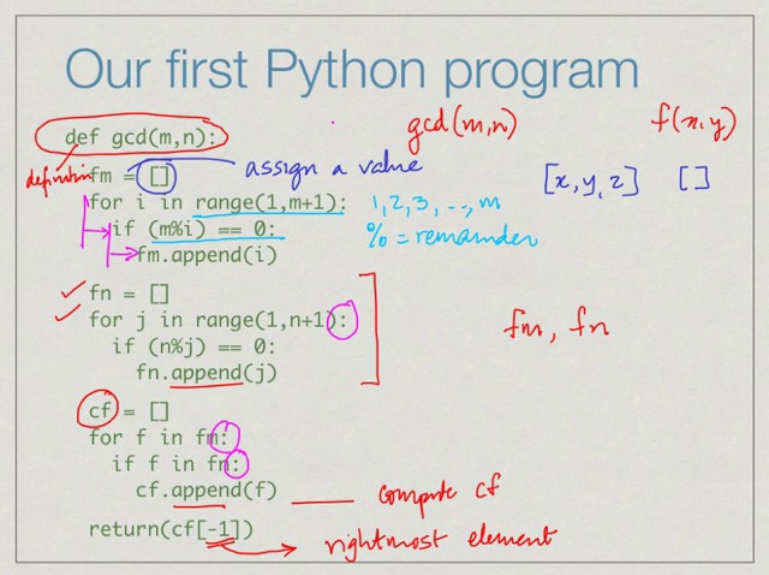
Now we have two lists fm and fn which are the factors of m and factors of n. Now we want to compute the list of common factors, which we will call cf. So, what we do is for every f that is a factor of a first number, remember in our case was 14 where each f so we ran through 1, 2, 7 and 14 in our case 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.

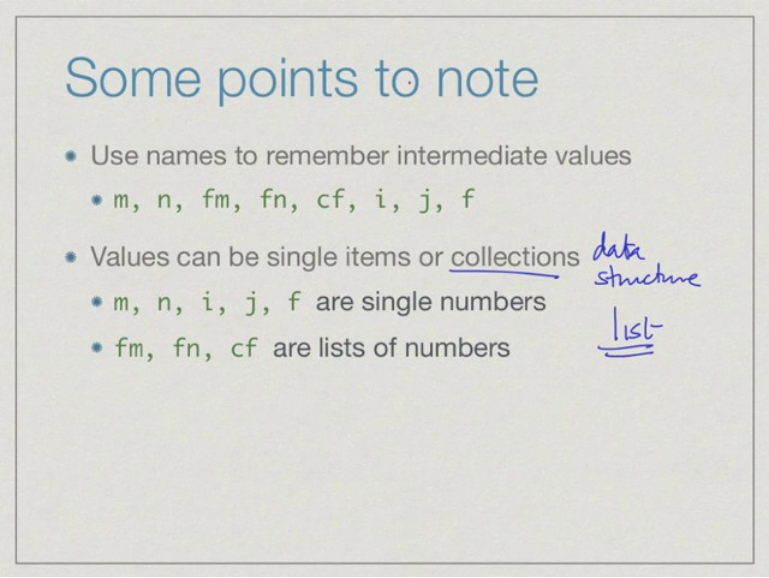
So, the same function append is being use throughout. Just take a list and add a value. Which value? We add the value that we are looking at now provided it satisfies the conditions. So, earlier we were adding provided the divisor was 0 uh the remainder was 0, now we are adding it provided it appears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



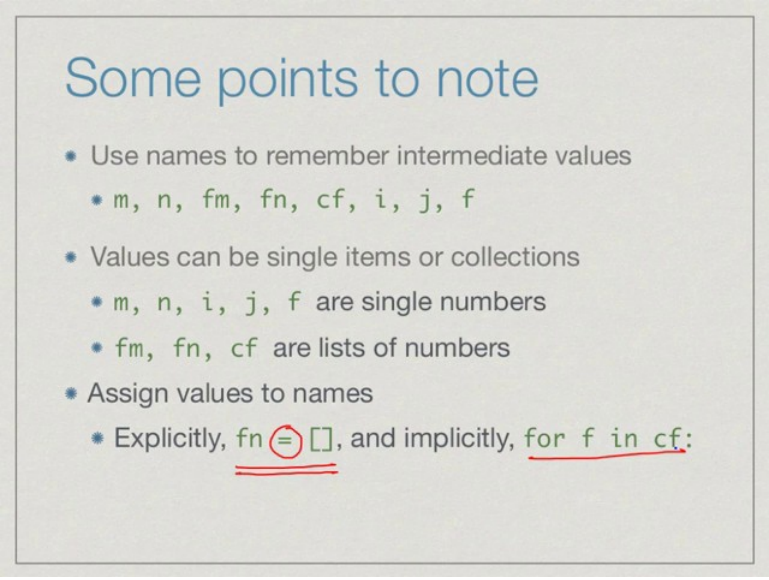
Let us note some points that we can already deduce from this particular example. So, the first important point is that we need a way to keep track of intermediate values. So, we have two names to begin with the names of our arguments m and n. Then we use these three names to compute this list of factors 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)

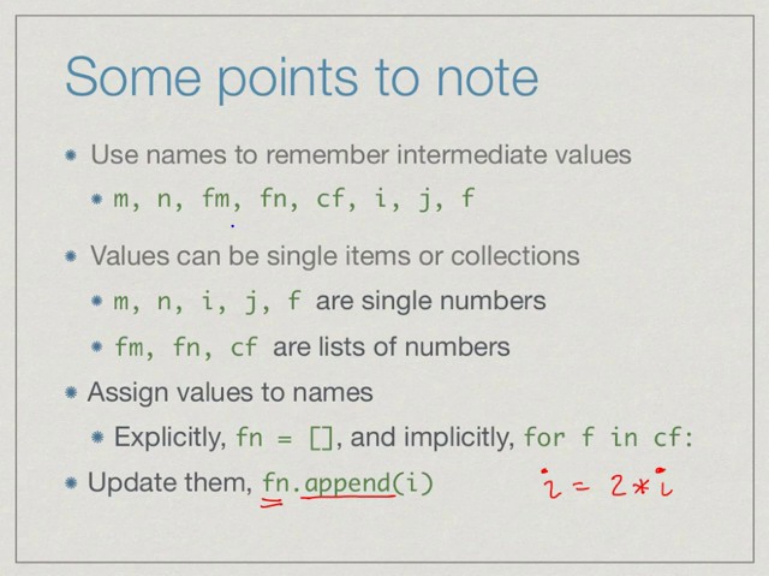


What can we do with these names and values well one thing is we can assigned a value to a name. So, for instance when we write fn is equal to the empty list we are explicitly setting the value of fn to be the empty list. This tells two things this says the value is

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)

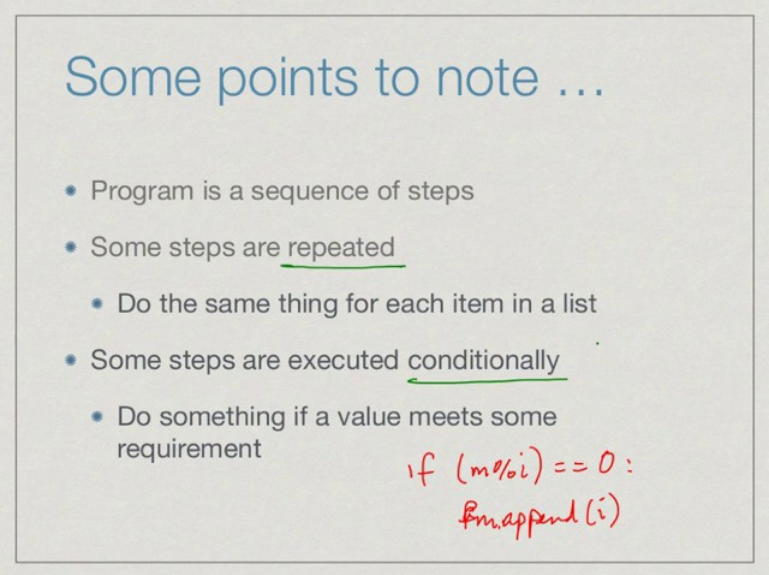


And having assigned a value we can then modify the value. For instance every time we find a new factor of n we do not want to through any old factor we want to take the existing list fm and we want to add it. So, this function append for instance modifies the value of the name fn to a new name which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

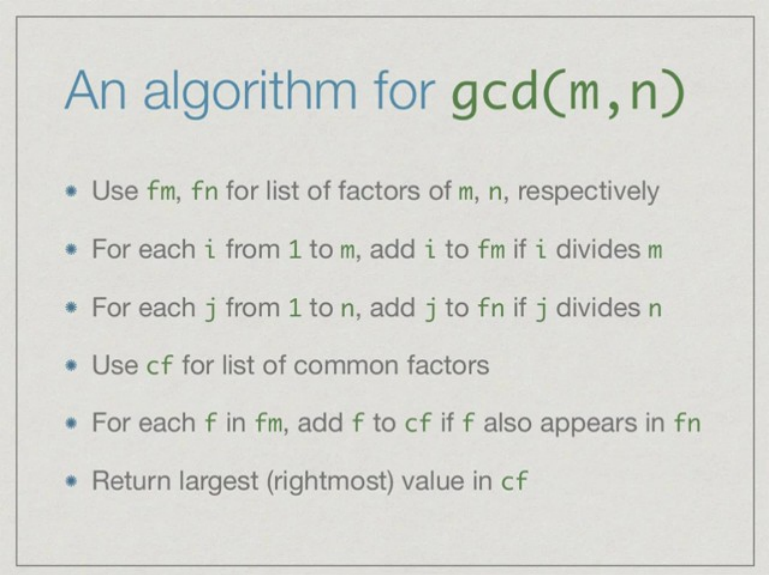
So, we will stop here. These examples should show you that programs are not very different from what we know intuitively, it is only a question of writing them down correctly, and making sure that we keep track of all the intermediate values and steps that we need as we long, so that we do not lose things. 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

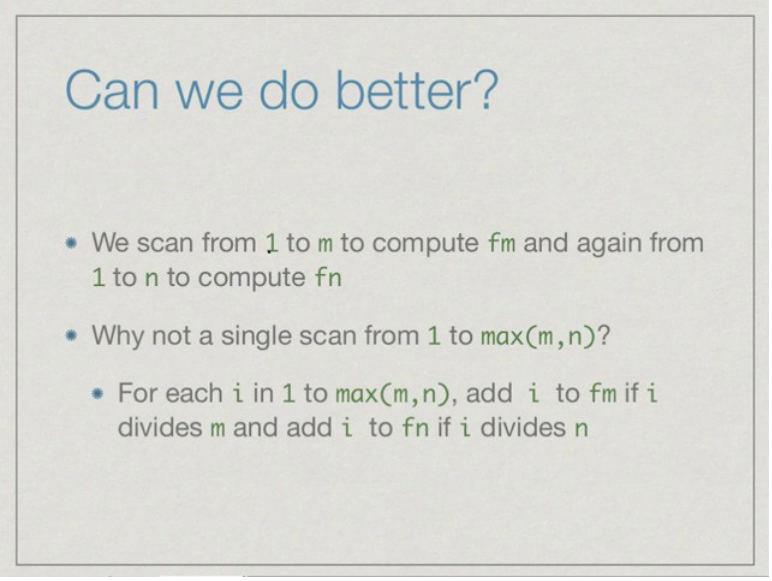
In the first lecture we used gcd as an example to introduce some basic concepts in programming. We will continue to look at the same example and see how to refine our program and explore new ideas.

(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.

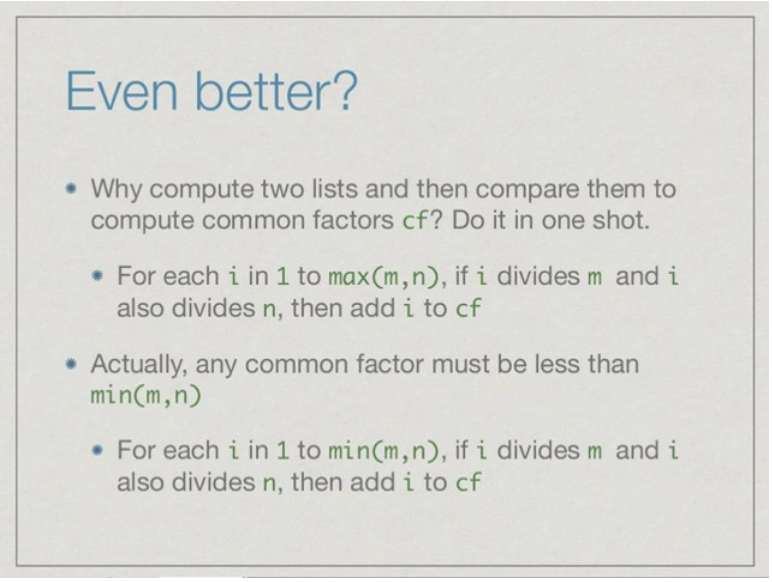
(Refer Slide Time: 00:51)



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.

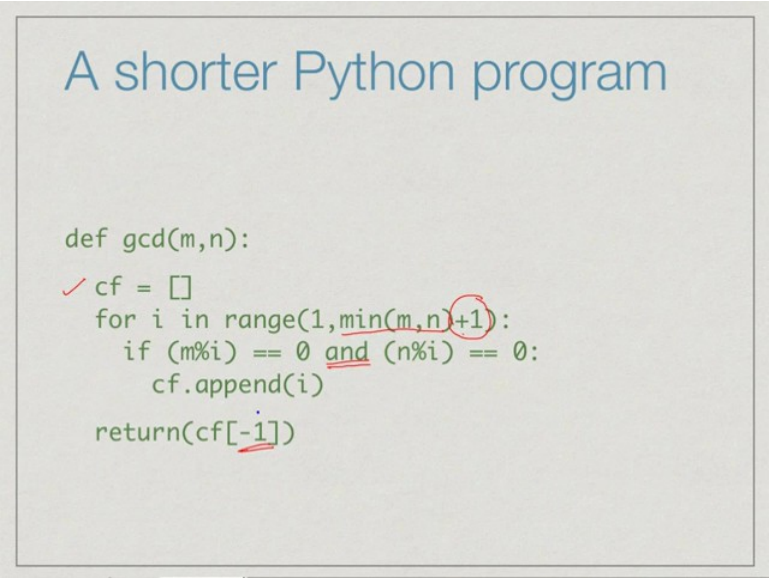
(Refer Slide Time: 01:46)



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.

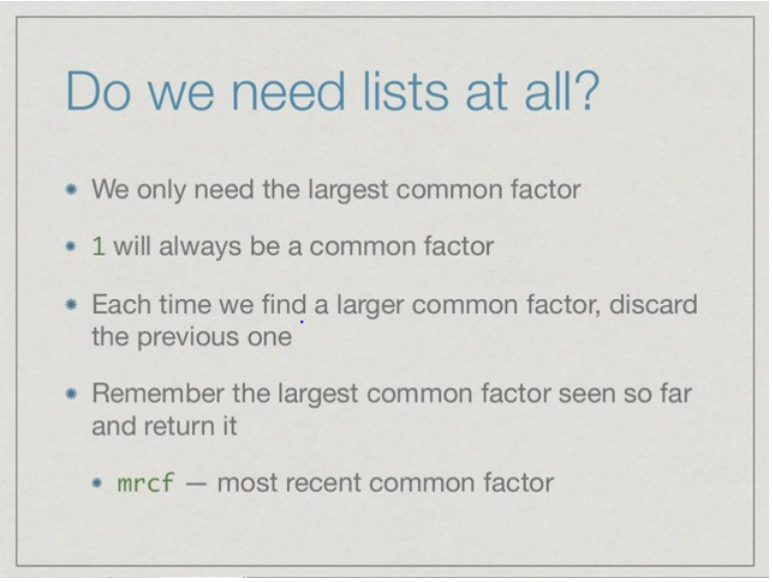
(Refer Slide Time: 03:03)



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.

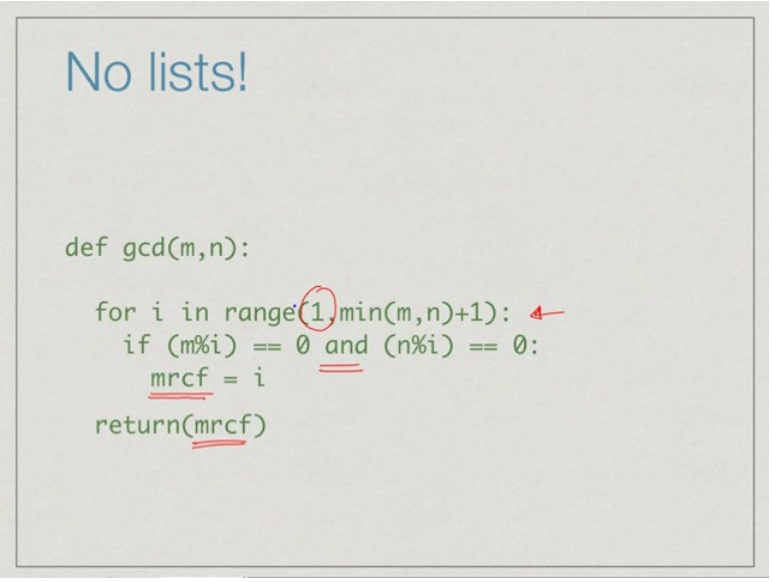
(Refer Slide Time: 04:15)



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.

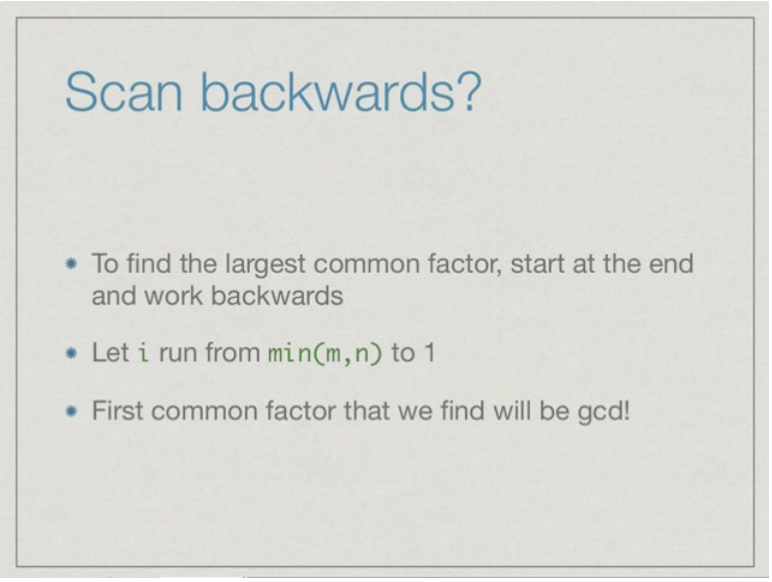
(Refer Slide Time: 05:17)



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.

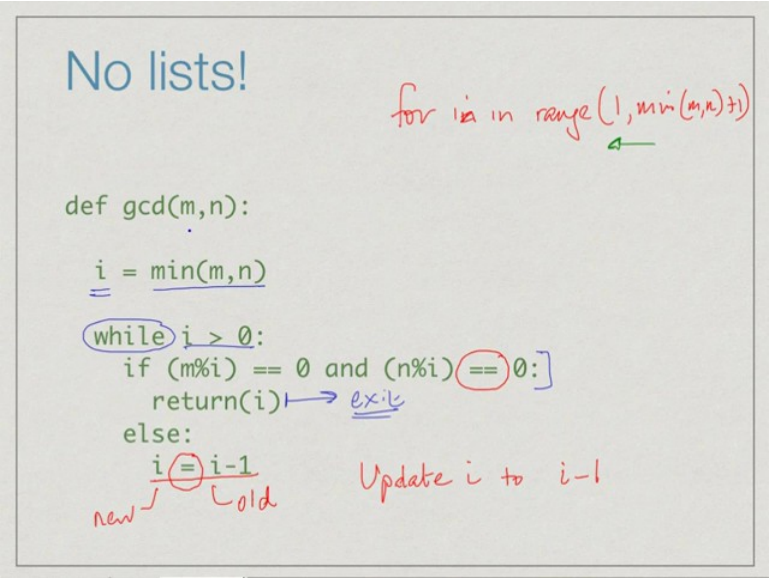
(Refer Slide Time: 06:18)



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.

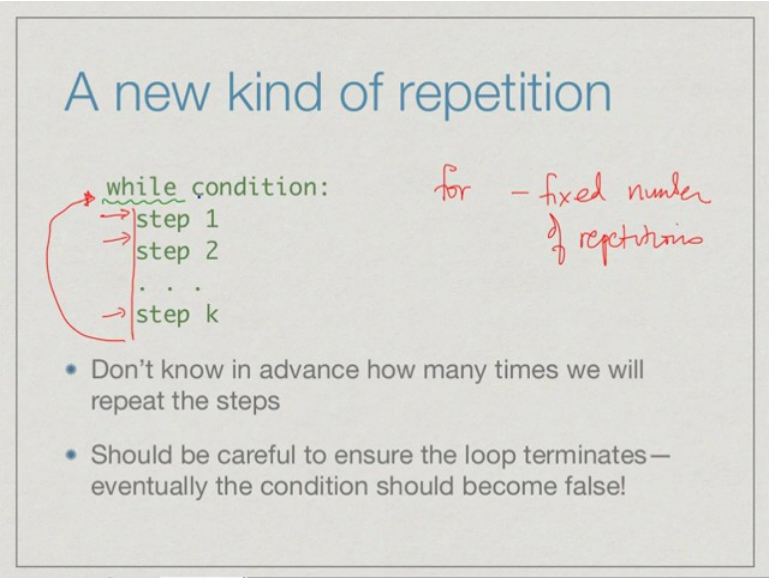
We start by assigning to the index i, the value that we want to start with namely the minimum of m and n, remember we want to start at the largest possible value for i and go backwards. So what we have is, a new word instead of for called while, so while as the English word suggests is something that 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)



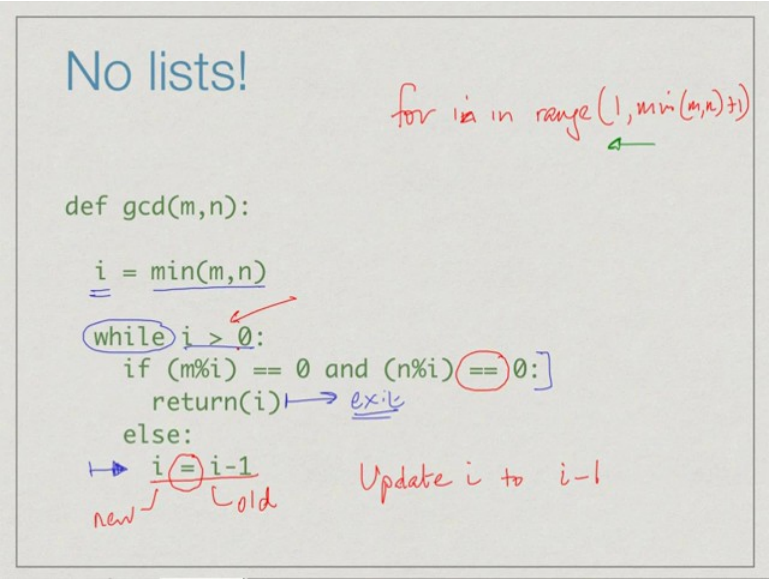
What we saw in this example is a new kind of loop. So, this new kind of loop has a special word while. And while is accompanied by condition, so long as the condition is true, we do whatever is within the body of the while. Now notice that Python uses indentation, so these statements here are offset 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.

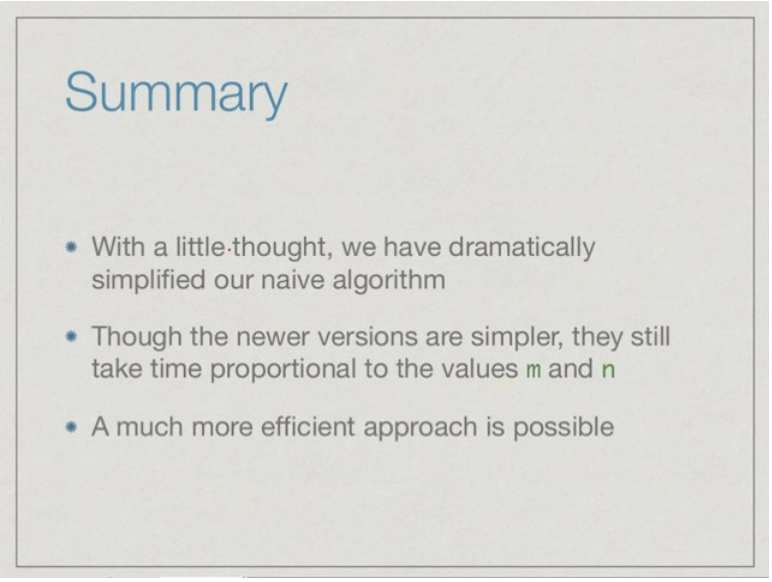
(Refer Slide Time: 11:51)



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.

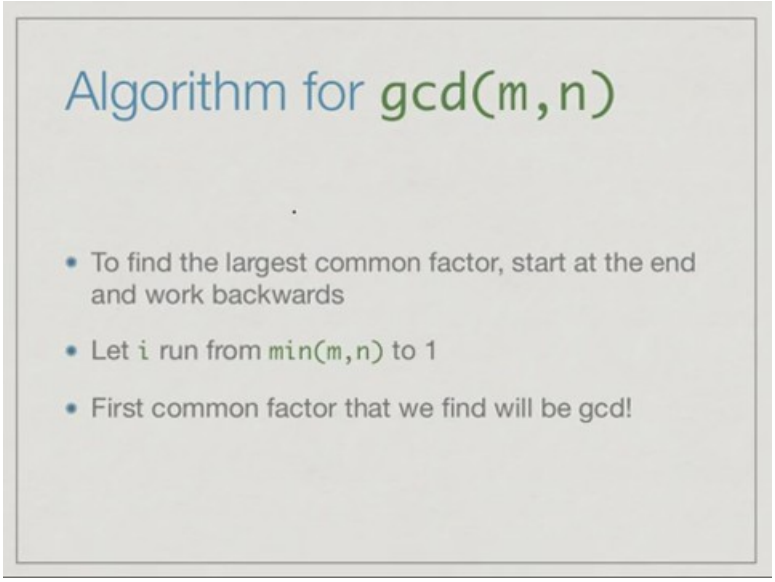
What we will see in the next lecture is that we can actually come up with a dramatically different way to compute gcd, which will be much more efficient.

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)

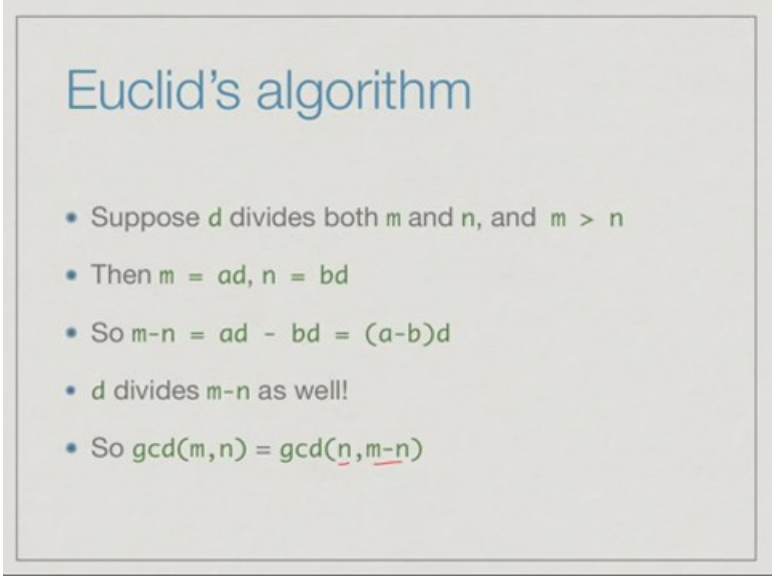


We started with the basic definition of gcd, which said that we should first compute all the factors of m, store it in a list, compute all the factors of n, store it in another list, from these two lists, extract the list of common factors and report the largest one in this common factor list. Our first 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)



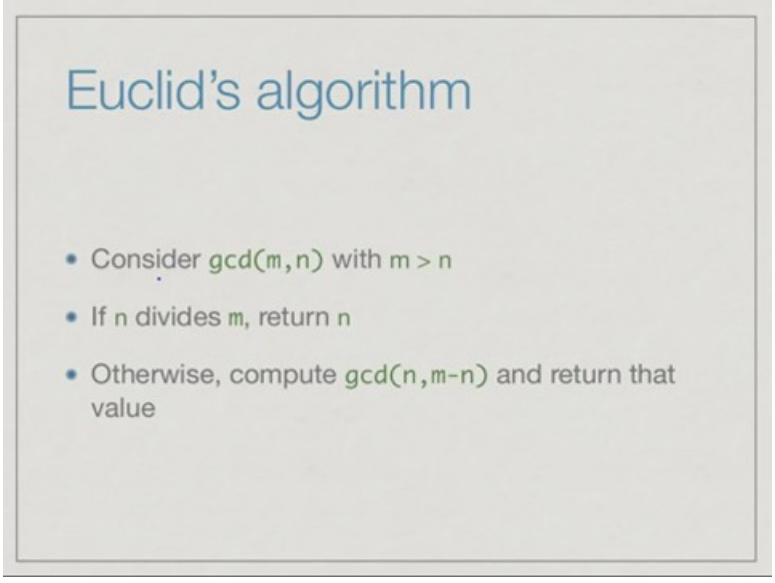
So at the time of the ancients Greeks, what was possibly the first algorithm in modem terminology was discovered by Euclid, and that was for this problem - gcd. So what Euclid said was the following. Suppose we have a divisor d which divides both m and n, so this is a common divisor and we are looking 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.

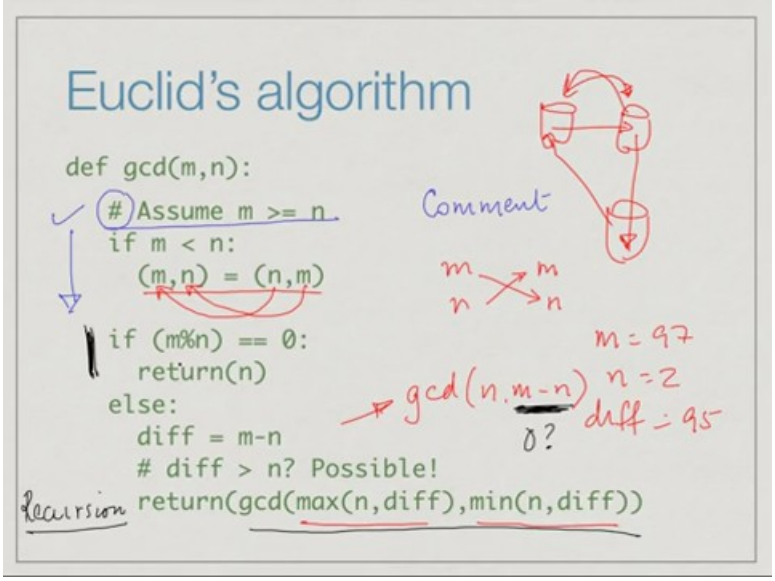
(Refer Slide Time: 04:00)



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.

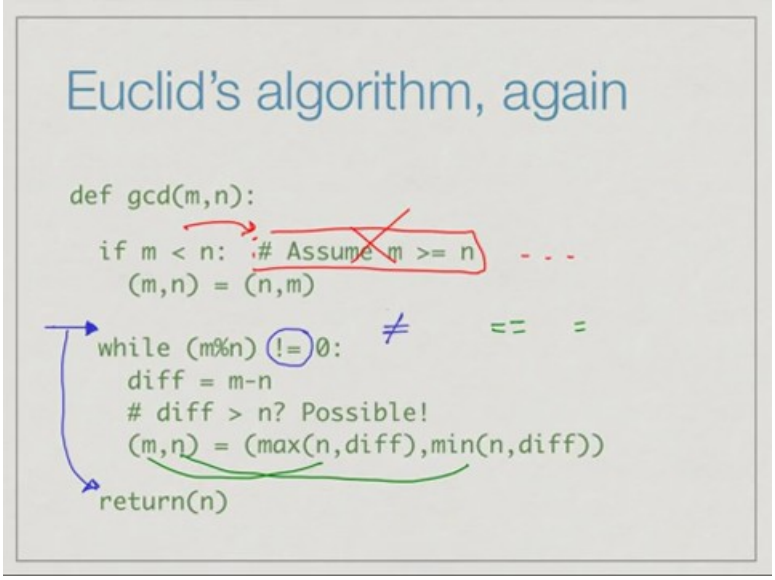
Now whenever you do a recursive call like this, it is like a while loop; it will invoke the function again, that in turn will invoke a smaller function and so on. And you have to make sure that this sequence in which gcd keeps calling gcd with different values does not get to an infinite progression 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.

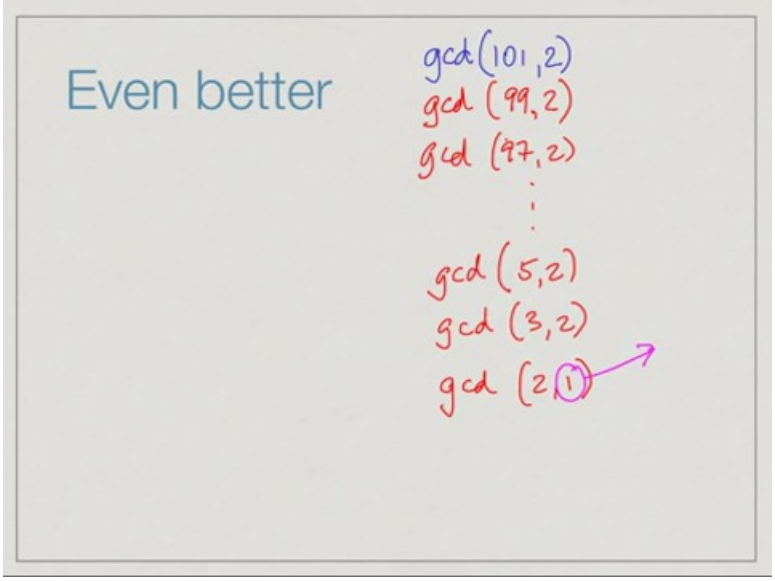
This exactly what we are doing in the recursive call, we are saying pretend we are computing gcd of that. Here in this while loop effectively we are saying replace the gcd of m n by the computation of maximum n diff and minimum n diff. We keep doing this until we hit a condition where n actually divides 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.

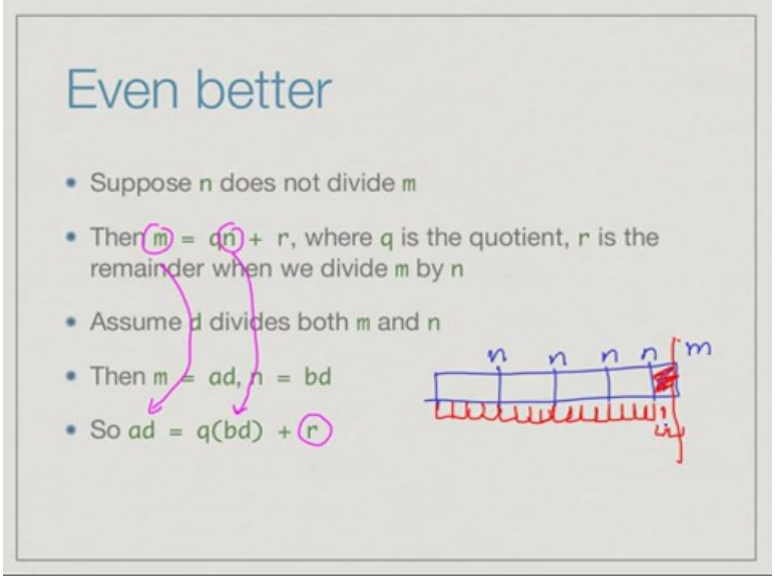
(Refer Slide Time: 15:42)



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)



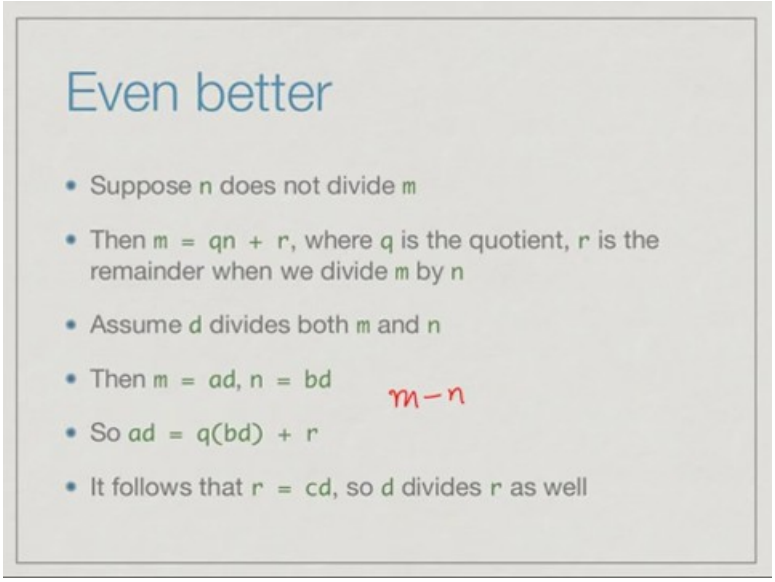
Here is a better observation suppose n does not divide m. In other words if I divide m by n i will get a quotient and a remainder. So, I can write n as q times n plus r where q is quotient and r is the remainder, so you may remember these terms from high school arithmetic. N goes into n q times and leaves 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.

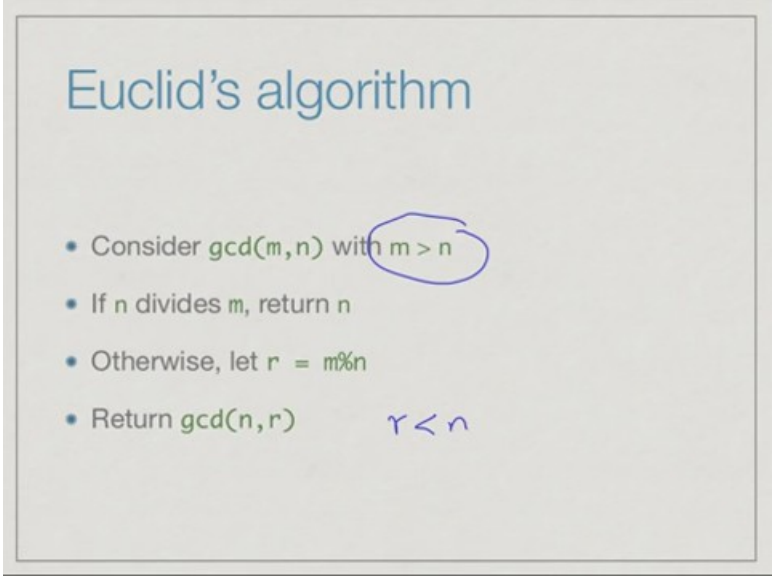
The way to think about it if you want to pictorially is that I have this number m and I can break it up into units of n and then there is a small bit here. On the other hand if I look at d, d evenly divides everything. So it divides each of these blocks it also divides the whole thing. If I continue 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.

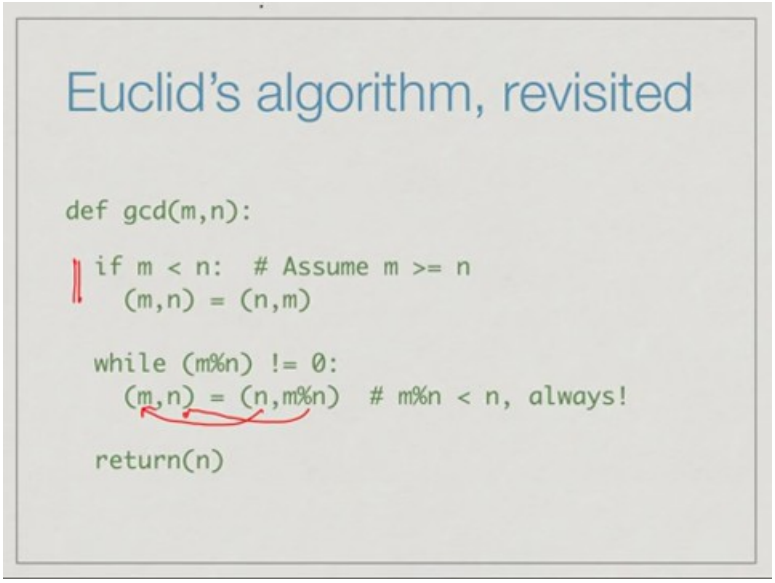
(Refer Slide Time: 19:44)



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.

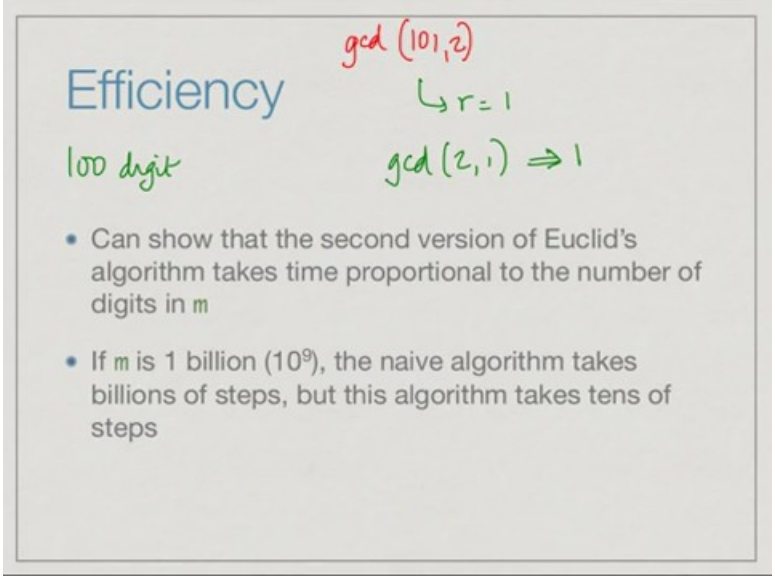
(Refer Slide Time: 20:31)



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.

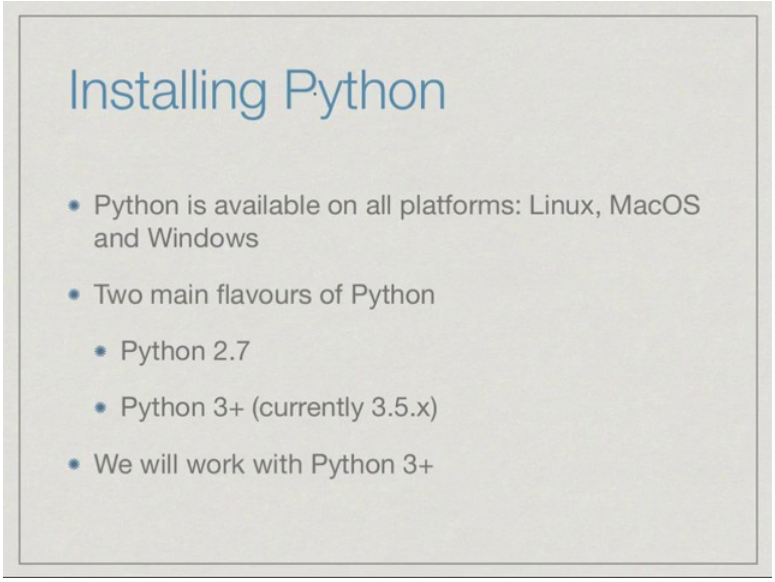
This is like writing in any other language. You may have brilliant ideas to express, but if you cannot convey them to the person you are talking to the ideas lose their impact. So, you need ideas and you need a language to express them. Programming is about expressing these ideas, but the ideas themselves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

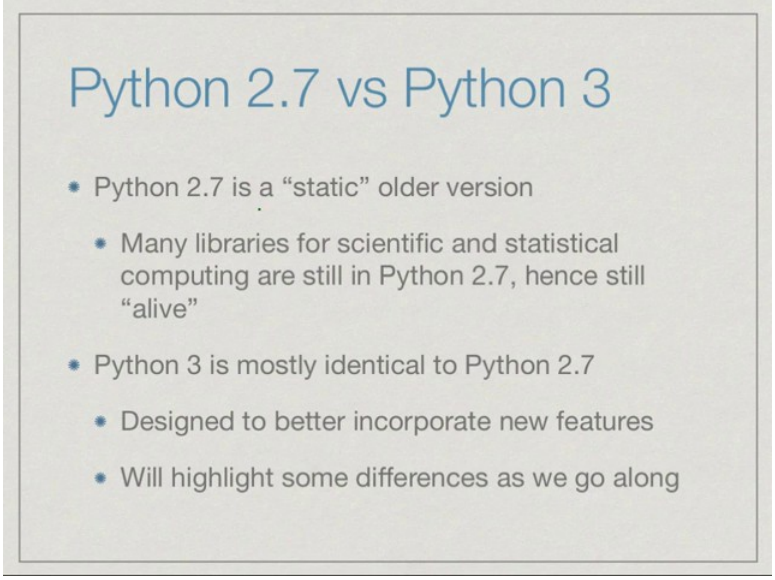
For our final lecture of this first week, we will see how to actually use Python on our system.

(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.

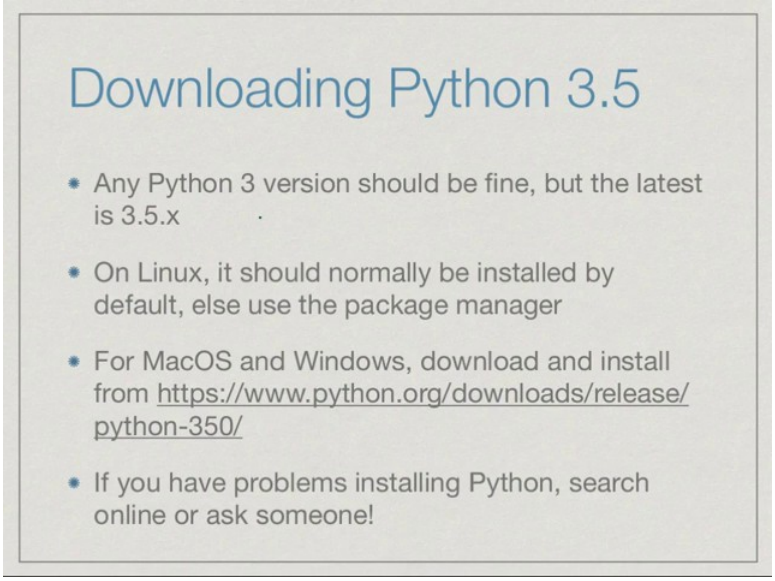
Python 3 essentially is a modern version of python, which incorporates features that where added on to python as it group in a way that makes it more consistent and more easy to use, but as often happens a lot of people had already been using python, and python 2.7 has a lot of software written using 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)

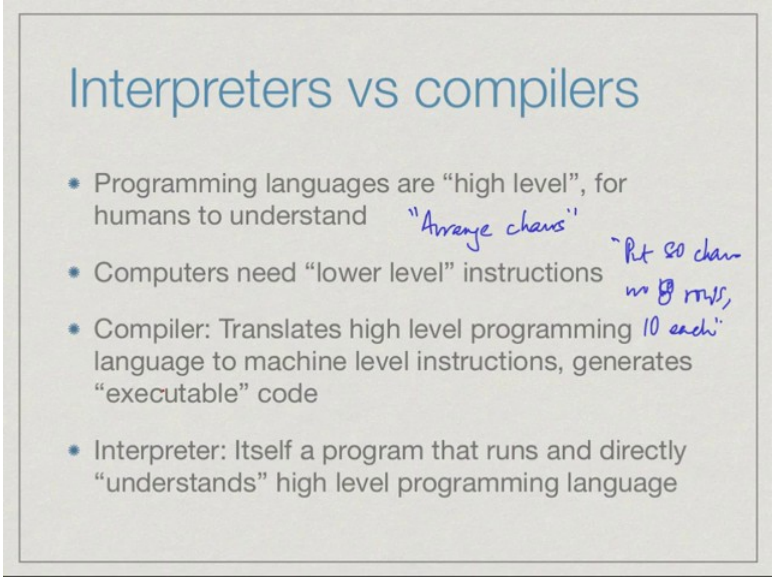


As far as this course is concerned, any version of python 3 should be fine. The latest version as I said is some 3.5.x, where I thing x is 2, but if you do not have 3.5, but you have 3.4 or 3.3 do not bother everything should work fine. But if you are interested, you can install the latest, latest version. 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

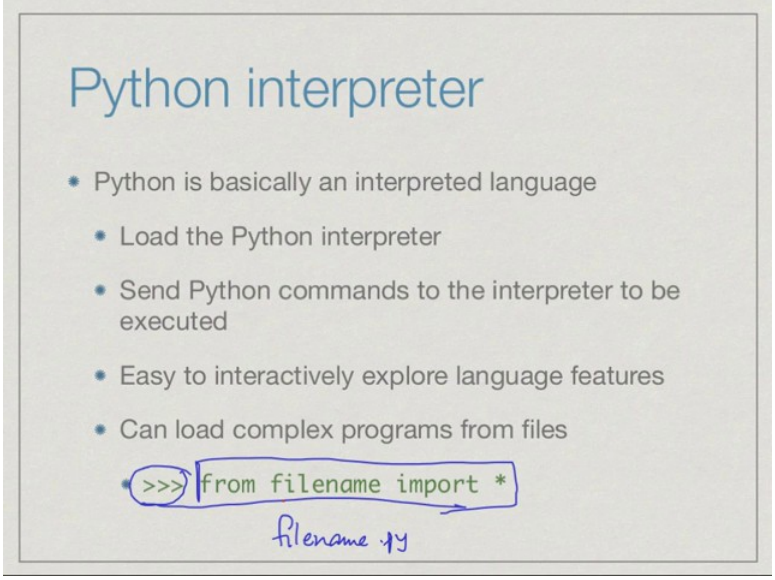
these things, so we need a translation. If you remember the very first lecture, we talked about arranging chairs. So, we said arrange the chairs as a high level thing, and we said put 80 chars in 10 in 8 rows, 10 each right.

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.

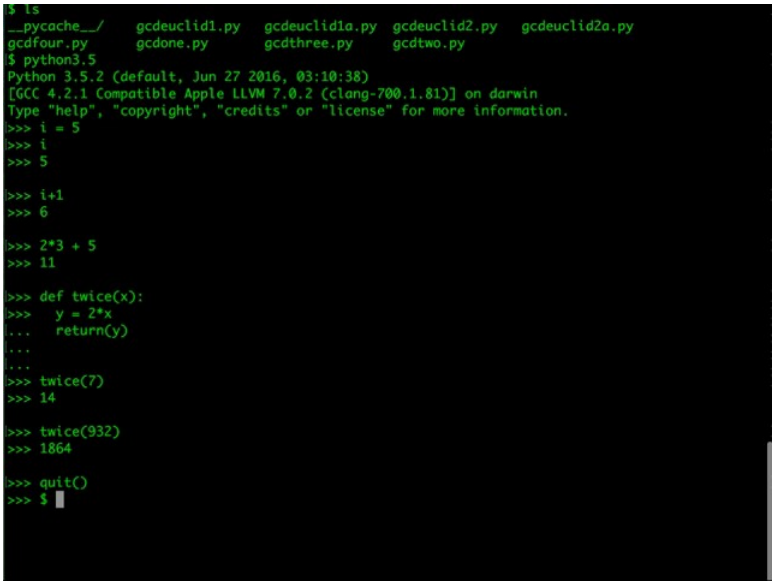
(Refer Slide Time: 07:31)



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.

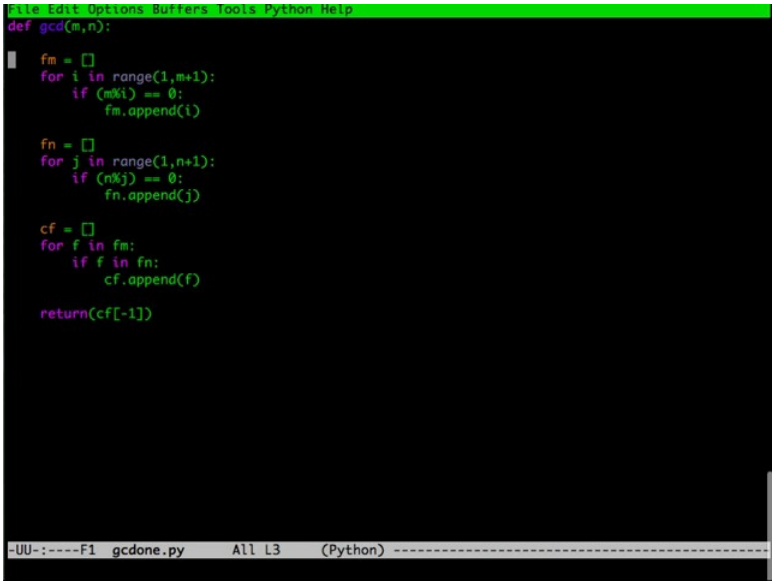
Let us to the sake of illustration create a new name y, and say y is two times x. Now it is still continuing to ask me for the definitions, so the prompt has change to dot dot dot. Now I must induct it a same way and say return y. So, what I have done is I say this function takes in value x, computes 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)

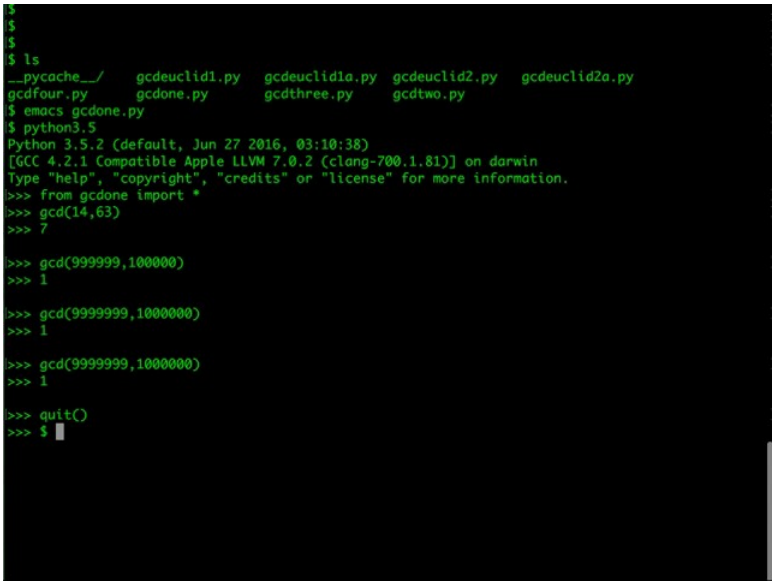
And then you get back to this prompt which is dollar which is the outside terminal or the command prompt. So, I have actually already created something. Let us start with, so I use an editor called emacs, you can use any takes editor if you are using Windows, you can use notepad, if you are using and 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.

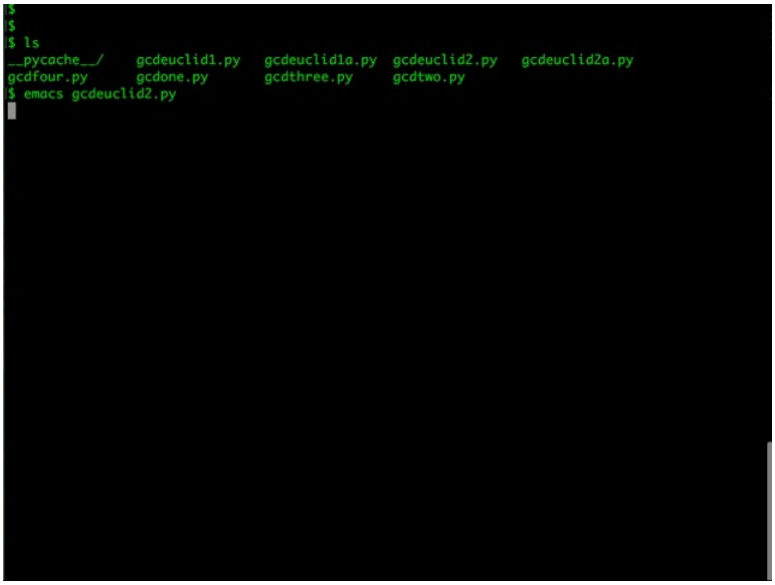
(Refer Slide Time: 14:03)



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.

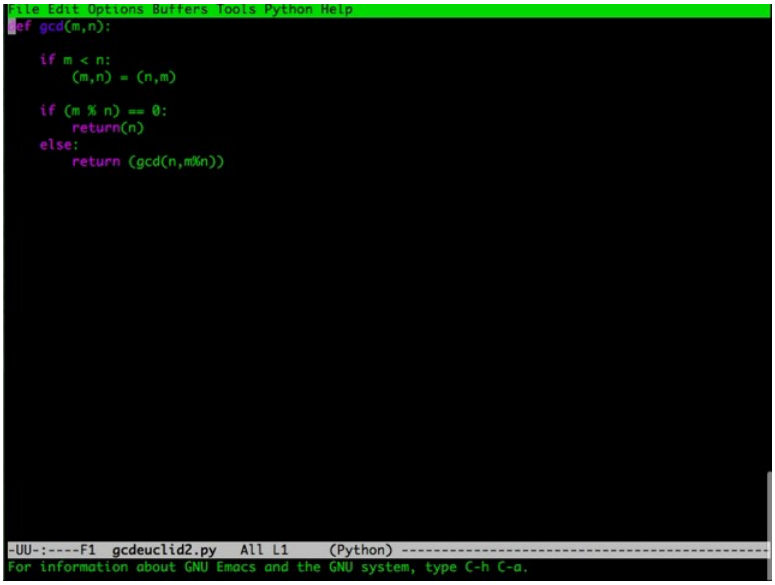
It has the visible gap of a few seconds before it produces the answer. And this is the illustration that this is not a very efficiency gcd. So, one of the problems with this python interpreter which I will see if we can solve is that if I have already loaded one file then it is safer to exit and then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

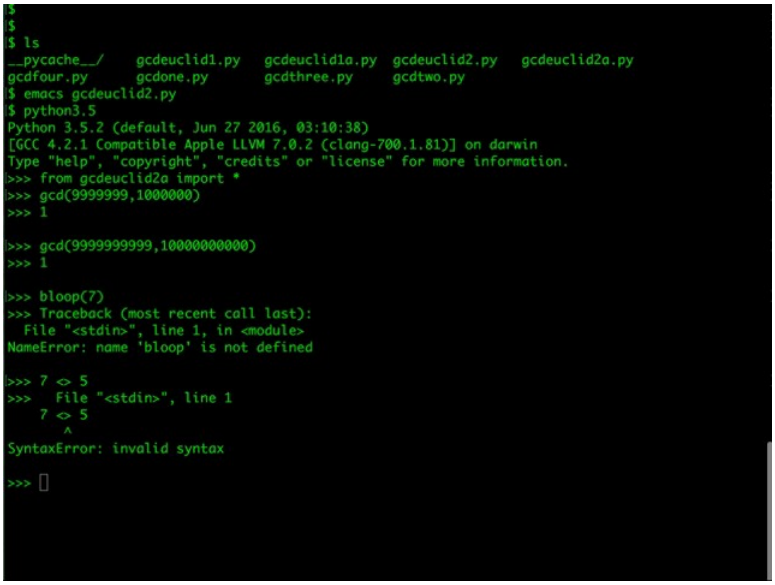
(Refer Slide Time: 15:32)



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)



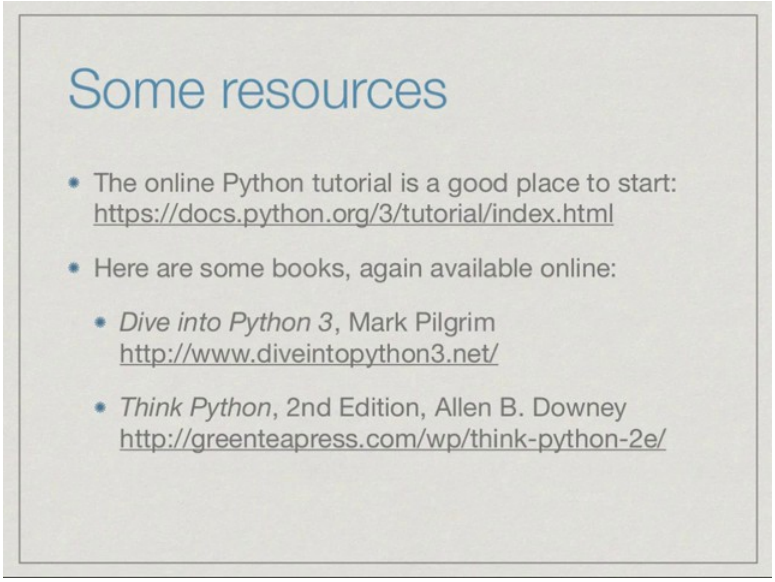
I am going to take this particular thing and load it into python. So, again I first invoke the interpreter python then I say from gcdeuclid2a import star. Now I am going to give that same large value that we saw before and which I think was say 9999999 and 1000000. And now you see, you get an instant 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)

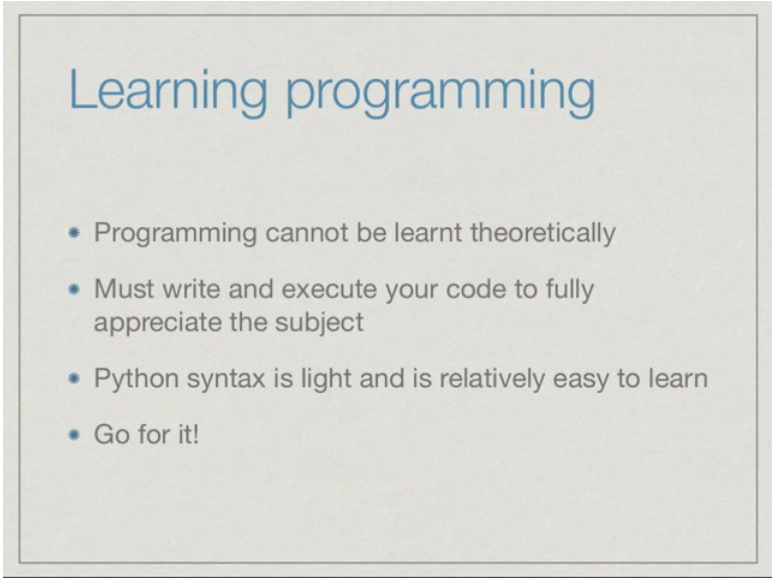


We are going to be looking at some specific features of python in this course, but you may find as we go along that there is something that you do not understand or something new that you would like to try out your own. So, it is always a good idea to have access to other resources. The python online 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.

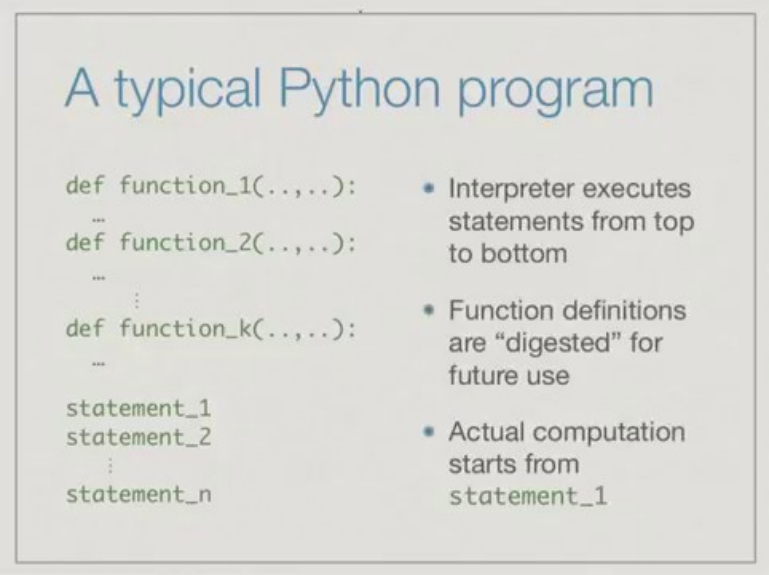
Although the syntax may vary, the ideas do not. The ideas are eventually what write the program, but to be a fluent speaker of a programming language, you must practice it. So, do try.

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)

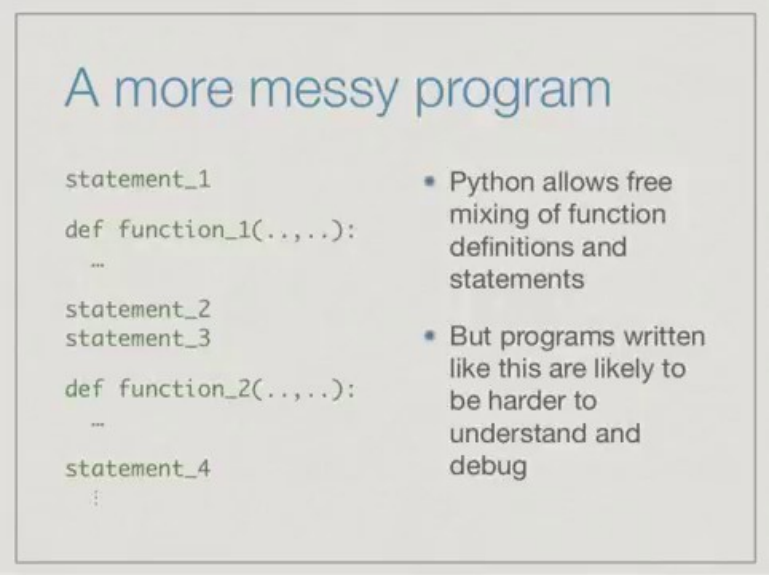


A typical python program would be written like this, we have a bunch of function definitions followed by a bunch of statements to be executed. So remember that we said python is typically interpreted, so an interpreter is a program, which will read python code and execute it from top to bottom. So, the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

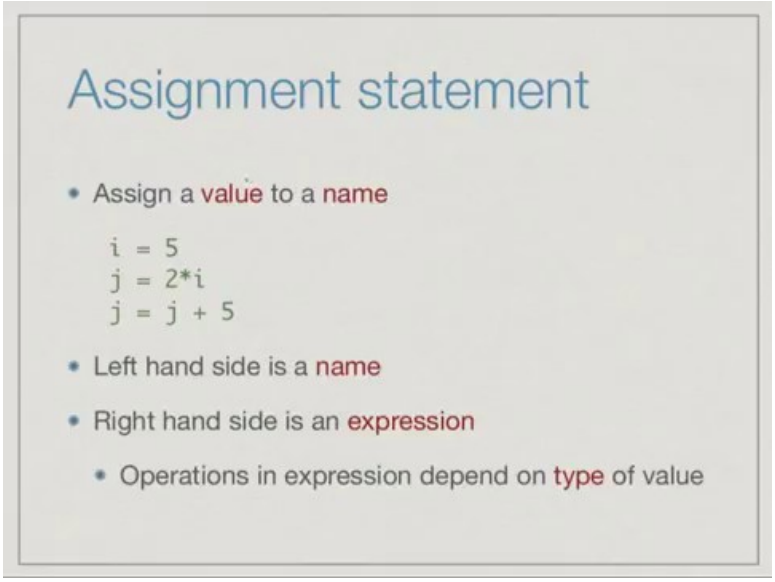
(Refer Slide Time: 01: 09)



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)



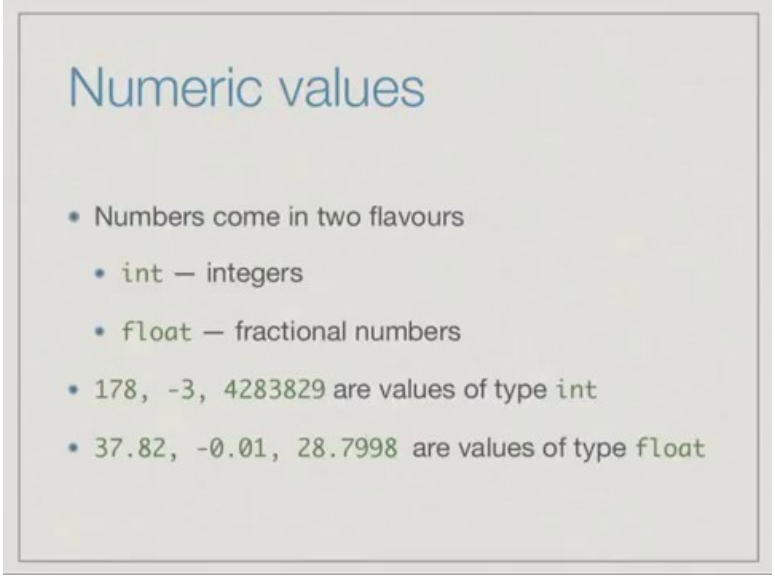
What is a statement ? The most basic statement in python is to assign a value to a name. So, we see examples and we have seen examples and here are some examples. So, in the first statement i is a name and it is assigned a value 5; in the second statement, j is a different name and it is assigned an 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

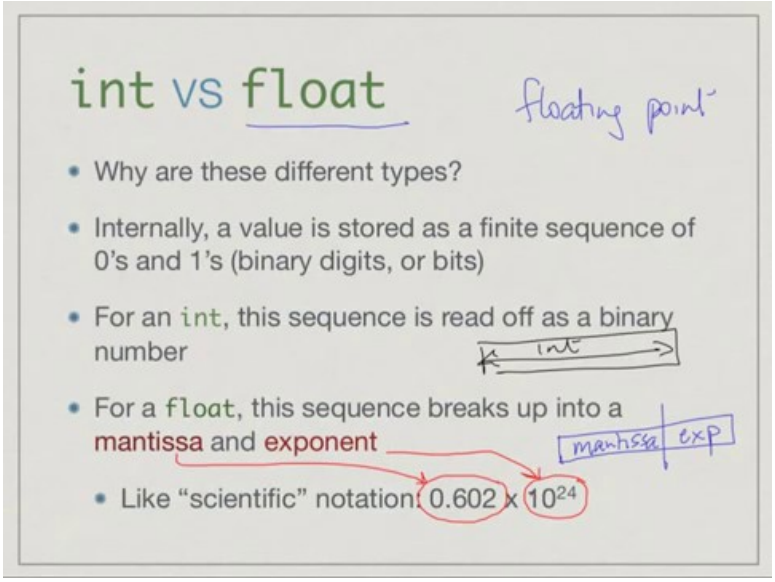
operations are allowed depend on the values and this is given technically the name type. So, when we said type of values it is really specifying what kinds of operations are legally available on that class of values.

(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.

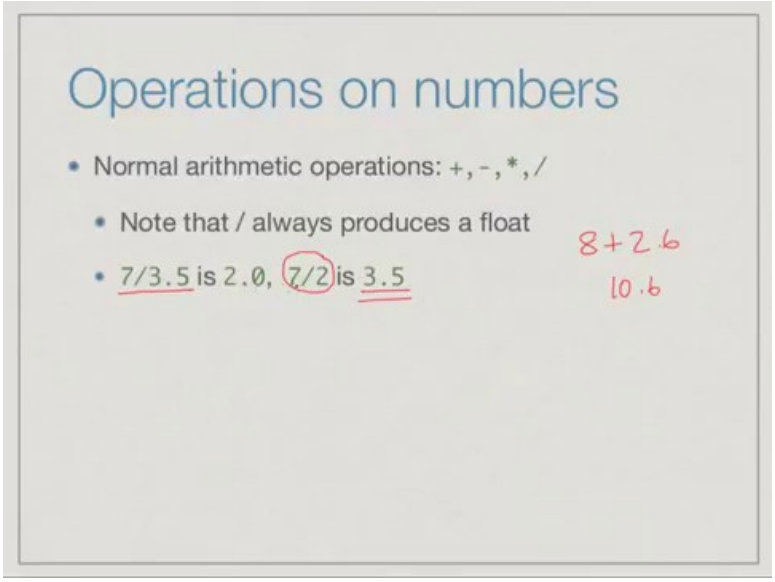
Now, if this happens to be an integer you can just treat that binary sequence as a binary number as you would have learnt in school. So, the digits represent powers of 2, usually there will be one extra binary digit 0 or 1 indicate whether it is plus or minus and they may be other more efficient ways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

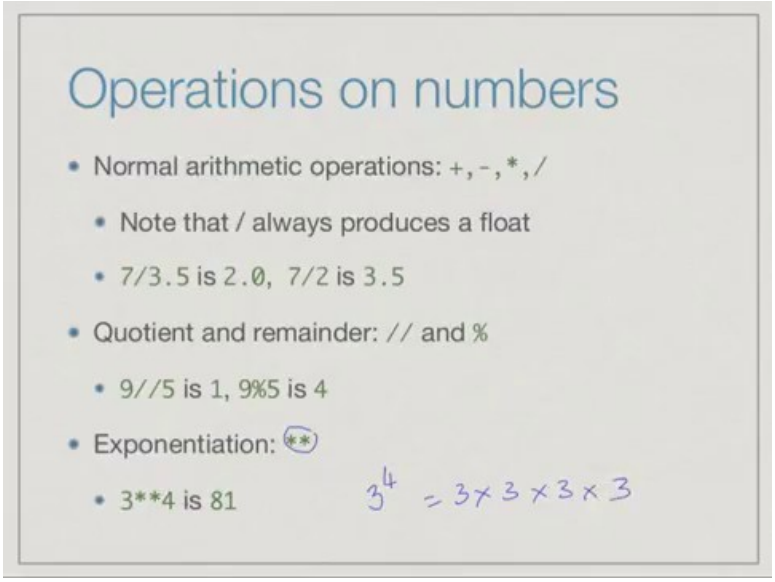
(Refer Slide Time: 07:59)



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.

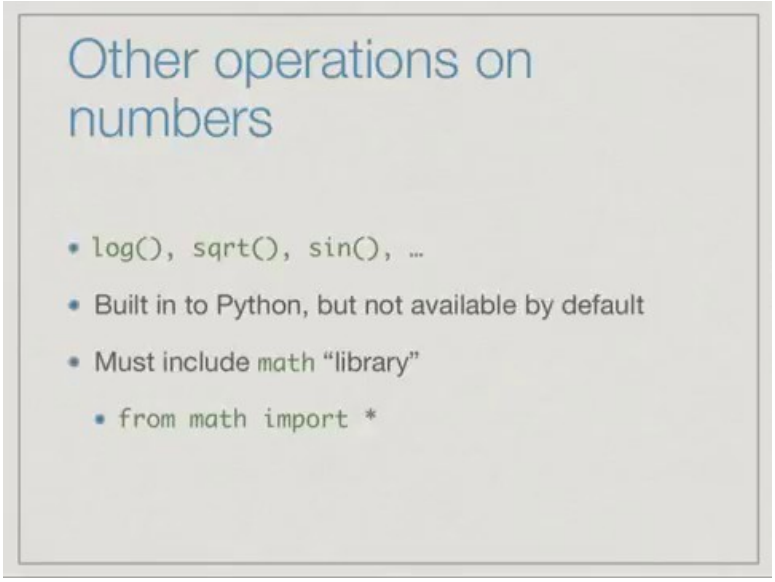
Now in general python will allow you to mix ints and floats, so i can write 8 plus 2 point 6 even though the left is an int and right is a float and it will correctly give me 10 point 6. In that sense python respects the fact that floats are a generalized form of int. So, we can always think of an int 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)



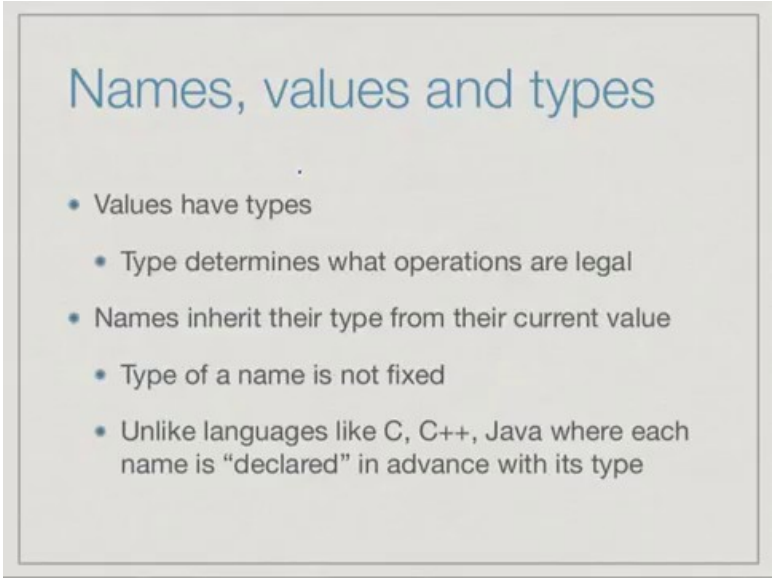
Now there are some operations where we want to preserve the integer nature of the operands. We have seen one repeatedly in gcd which is the modulus operator, the remainder operator. But the req corresponding operator that go through the reminder is the quotient operator. So, if I use a double slash it 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.

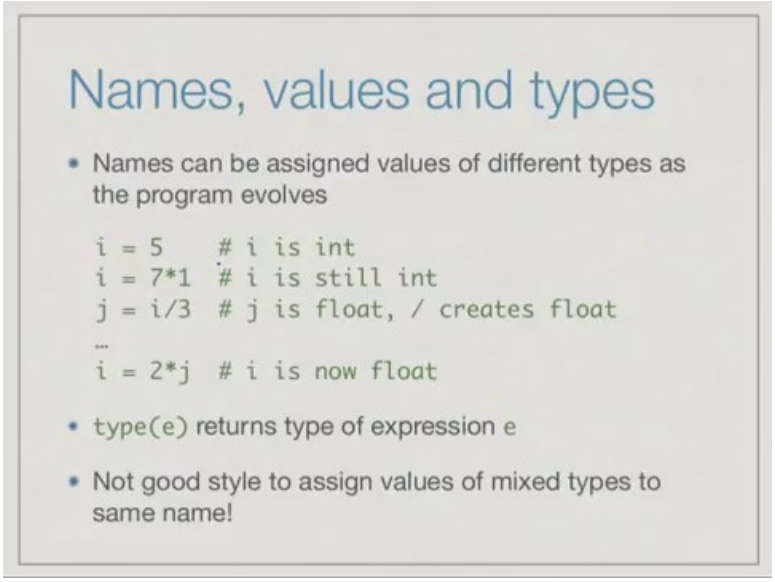
(Refer Slide Time: 10:58)



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.

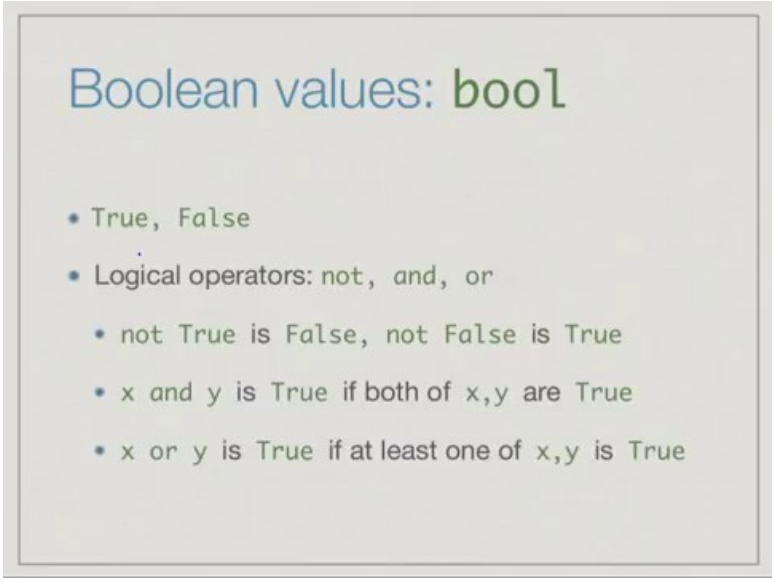
Now although python allows this feature of changing the type of value assigned to a name as the program evolves, this is not something that is recommended. Because if you see an i and sometimes its a float and sometimes its an int it is only confusing for you as a programmer and for the person trying 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.

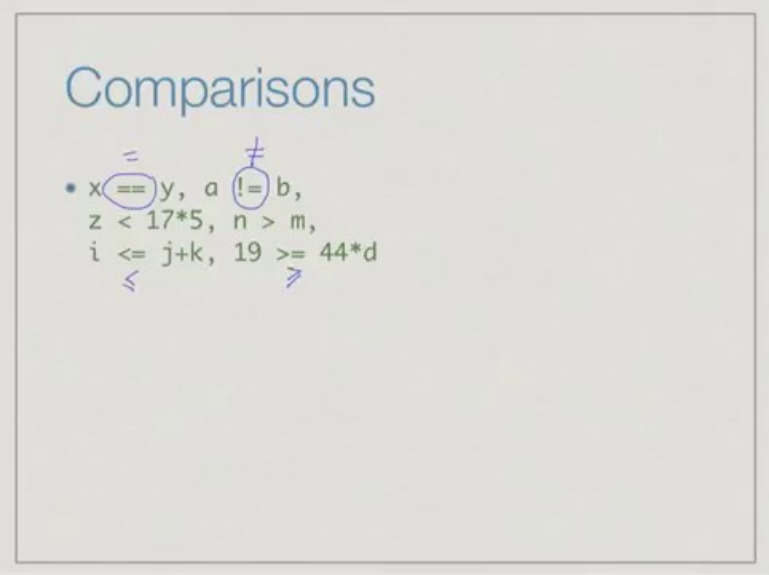
These are implicitly used to control the execution of our program. So, we need to have a way of recording these values and manipulate it. The basic values are true or false and typically there are three functions which operate on these values. So, not negates the value. So, true is the opposite of false. 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)

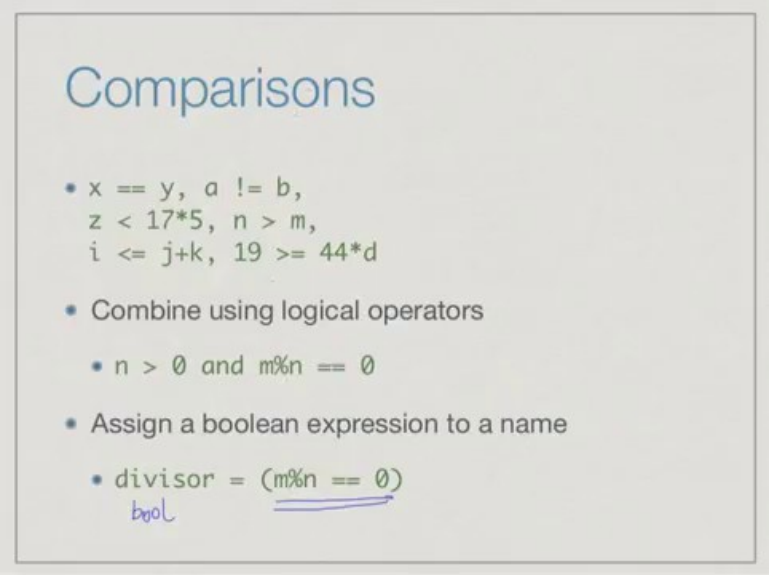


The most frequent way in which we generate Boolean values is through comparisons we have already seen the two of these. So, we have seen equal to - equal to. This is the actual equality of mathematics not the single equal to which is the assignment. So, if x equal to equal to y checks, whether the value 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.

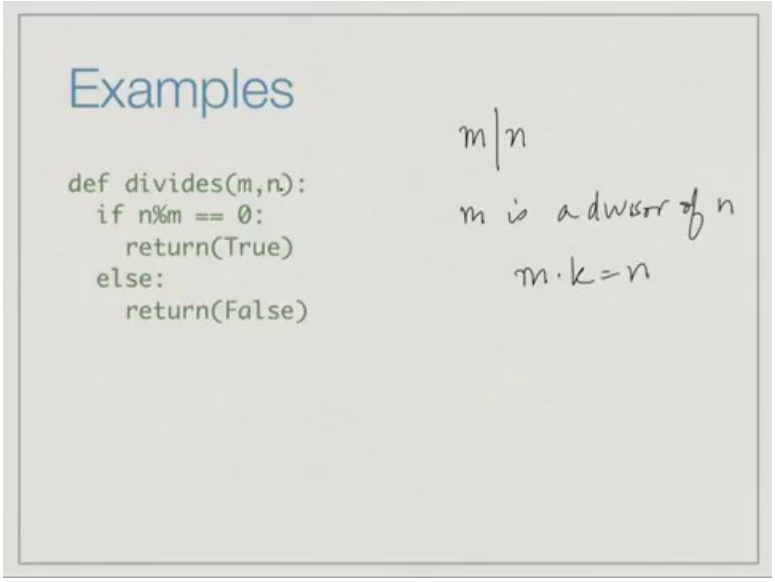
(Refer Slide Time: 18:24)



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.

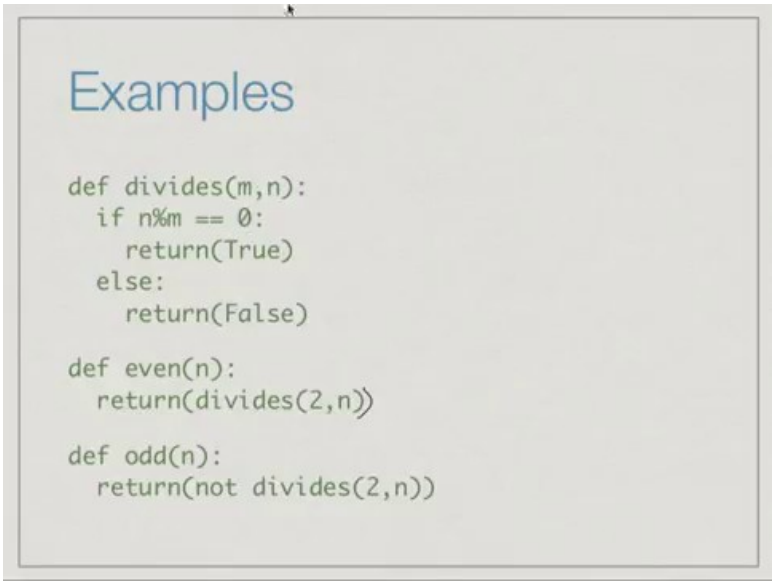
So, we can say that n is a divisor of m if the reminder of m divided by n is equal to 0. And we can say that the fact that it is a divisor its true provided this happens. So, divisor is now of type bool right and it has a value true or false depending or not whether or not n divides m evenly.

(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)

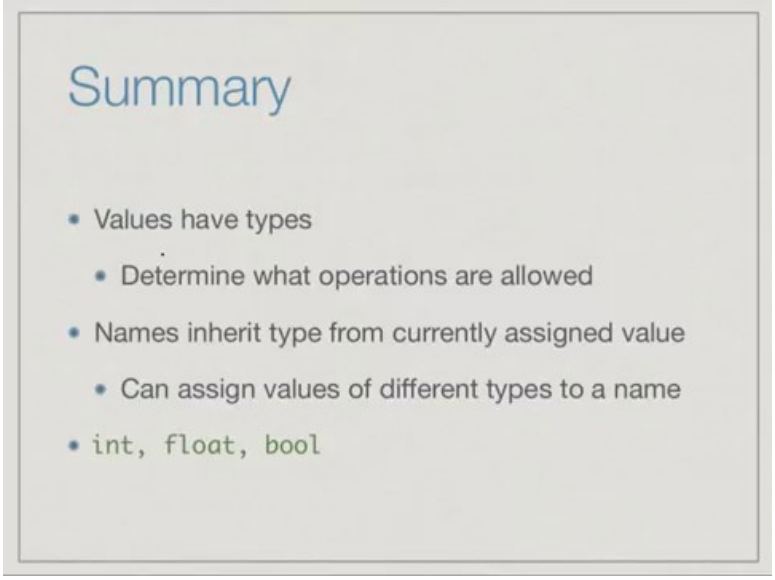


Now what we can do is define an another function called even whose value is derived

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

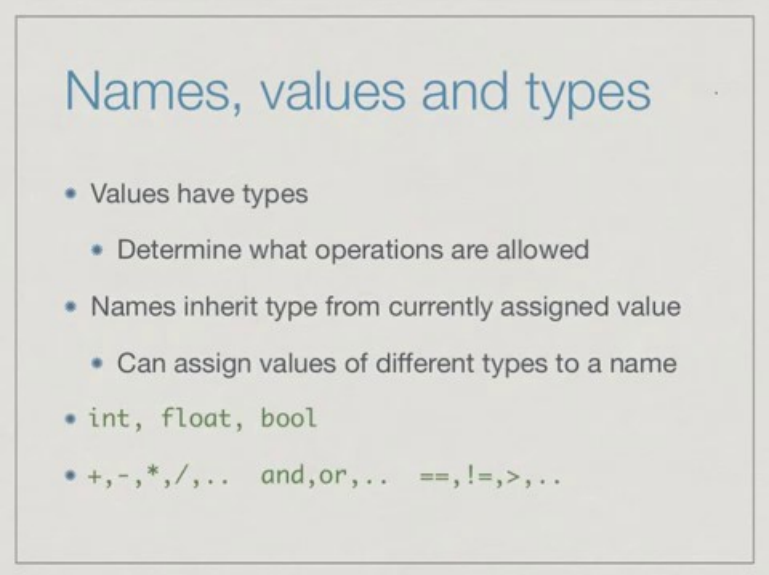
operations defined on them.

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)

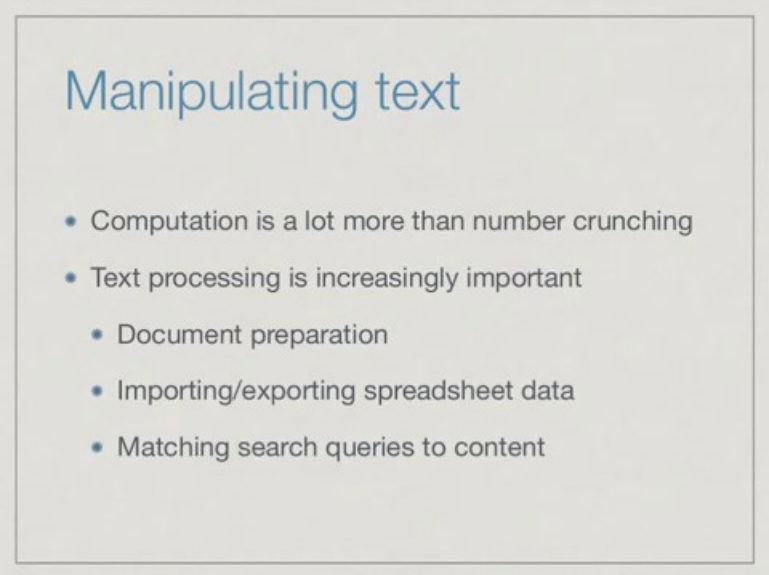


The types we have seen are the basic numeric types - int and float, and the logical type bool which takes values true or false. So, for the numeric types, we have arithmetic operations, we also have other operations which are more complicated. For the Boolean types we have and, or, not, which allows 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)

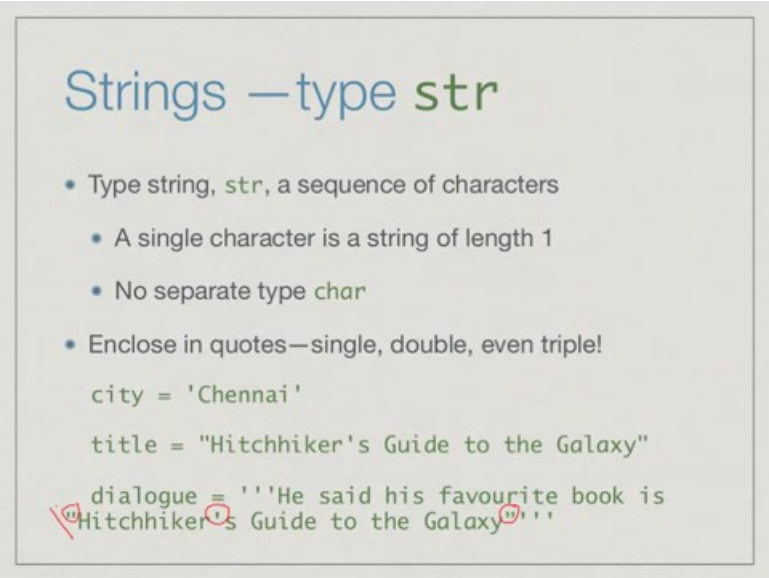


Numeric types by no means the only things that are of interest these days in computation. A lot of the computation we do is actually dealing with text. So, whenever we prepare a document, for example, using a word processor or some other things for presentation, then we are actually manipulating text; 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)



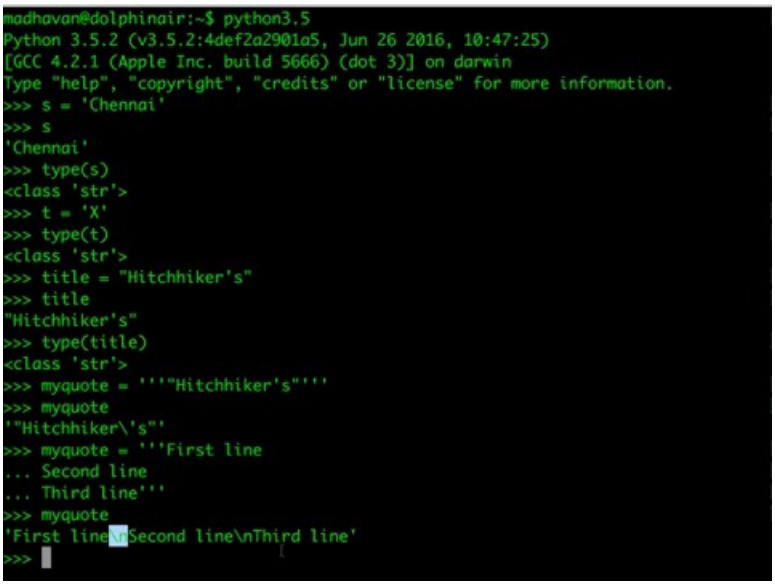
Python uses the type string for text, which internally is called str. So, we will use the word string instead of str, because it is easier to say. So, a string is basically sequence of characters. Unlike other programming languages, python does not have a specific character type to distinguish a single 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)

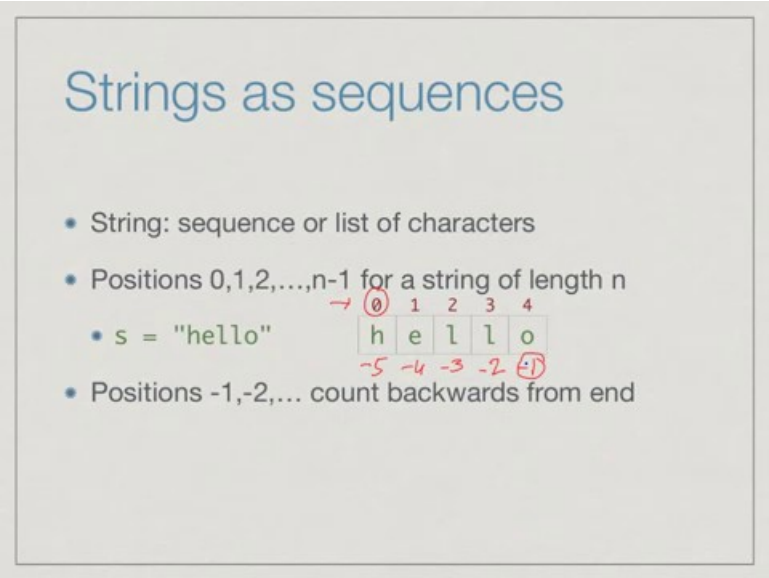


Let see how this works in python interpreter. So, we can say s equal to ‘Chennai’ and now been asked the value of this and we see that it is reported with single quote. If we ask for the type of s, it says that s is class of str. So, this tells us that internally python realizes that s is a string. If 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)



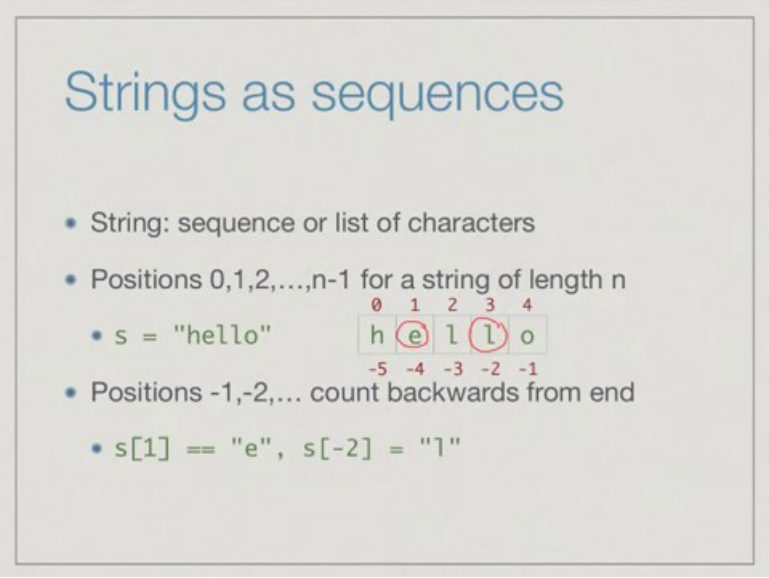
As we said the string is a sequence or a list of characters. So, how do we get to individual characters in this list? Well, these characters have positions and in python positions in a string start with 0. So, if I have n characters in a string, the positions are named 0 to n minus 1. So, supposing we 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.

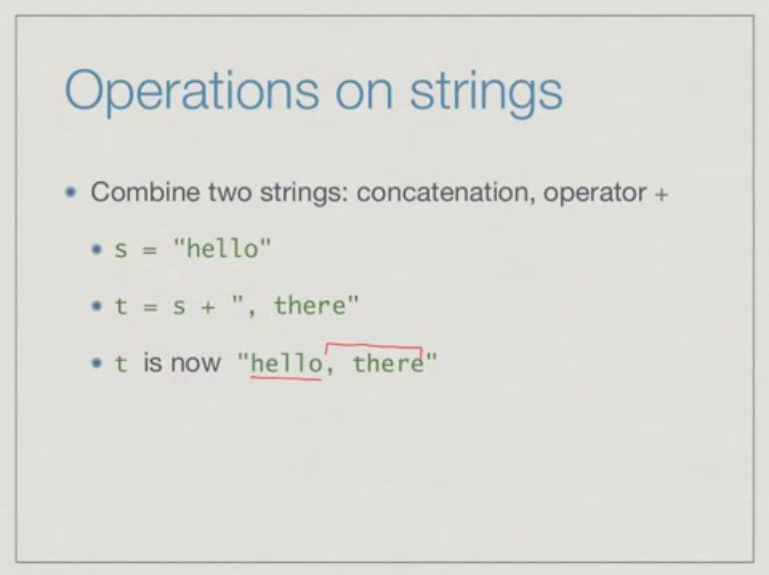
This numbering scheme that we use for list informally in the gcd example without formally explaining, it is actually the same numbering scheme that is used for positions in the string. We have minus 1, minus 2, minus 3, minus 4, minus 5, so the important thing to remember is that going forward, you start 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.

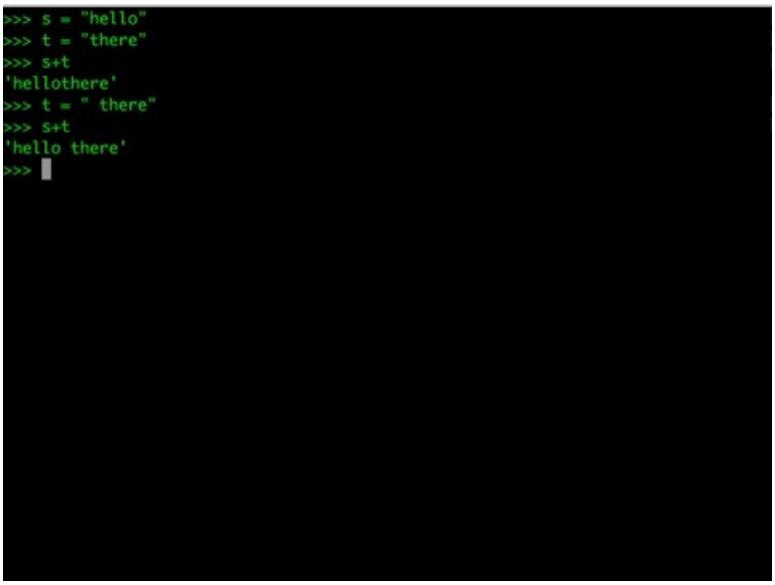
(Refer Slide Time: 10:59)



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.

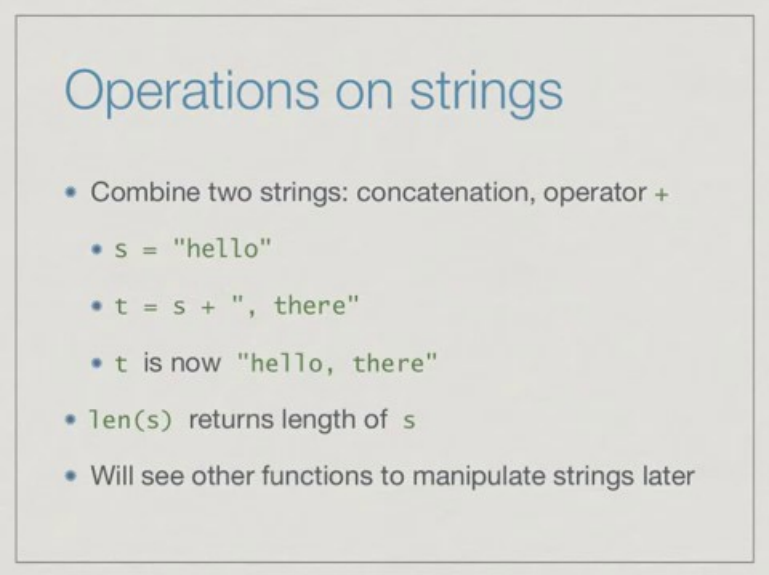
(Refer Slide Time: 11:52)



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.

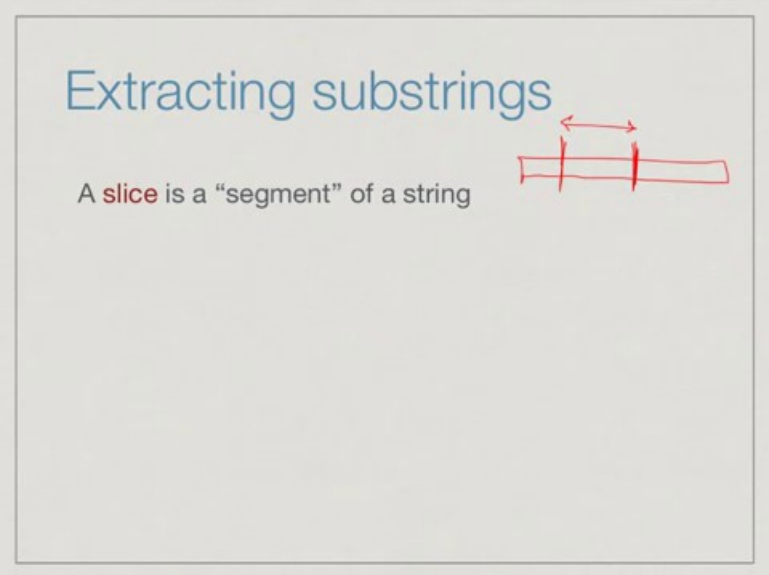
This is important to note that plus directly puts things together it does not add any punctuation or any separation between the two values. So, it is as though you have one new string which is composed of many old strings whose boundaries disappear completely.

(Refer Slide Time: 12:47)



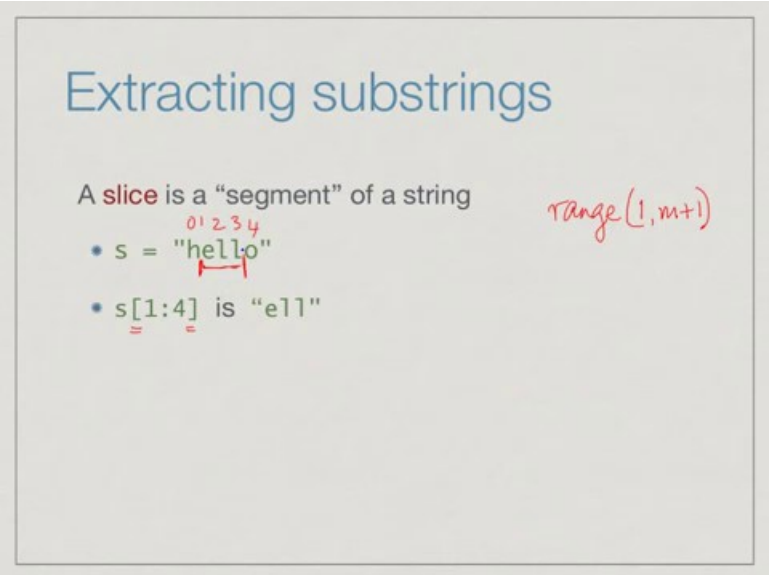
We can get length of the string using the function len. So, len(s) returns the length of s. So, this is the number of characters. So, remember that if the number of characters is n then the positions are 0 to n minus 1. So, the length of the string s here would be 5, the length of the string t here would 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.

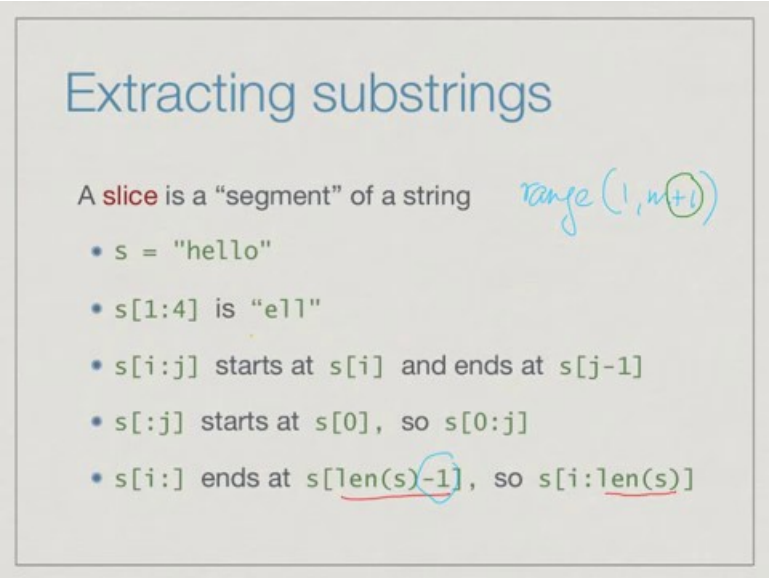
(Refer Slide Time: 13:55)



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)



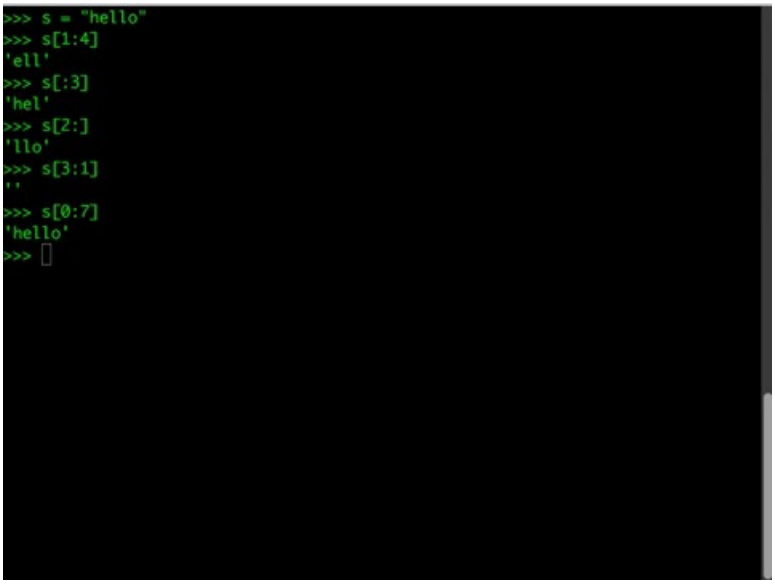
In general, if I write s i colon j then it starts at s i and ends at s j minus 1. There are some shortcuts which are easy to remember and use; very often you want to take the first n characters in the string, then you could omit the 0, and just say start implicitly from 0, so just leave it out, so just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

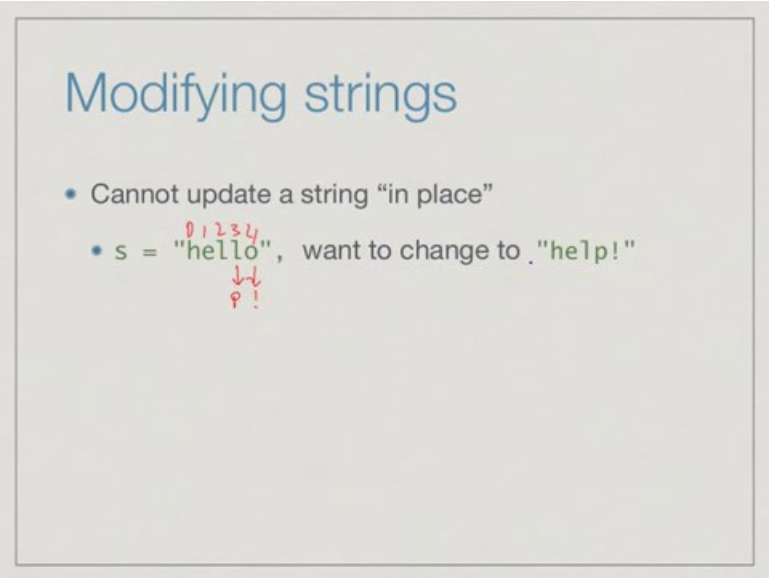


Let us play with the second in the python interpreter. So, if I say s is equal to hello then we saw that if I do 1 to 4, I get 'ell'. If I say colon 3 then I get 'hel' that is 0 1 2. If I say 2 colon, I get 'llo' that is 2 3 4. What if I say 3 2 1, so this says: start at position 3 and go up to position 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.

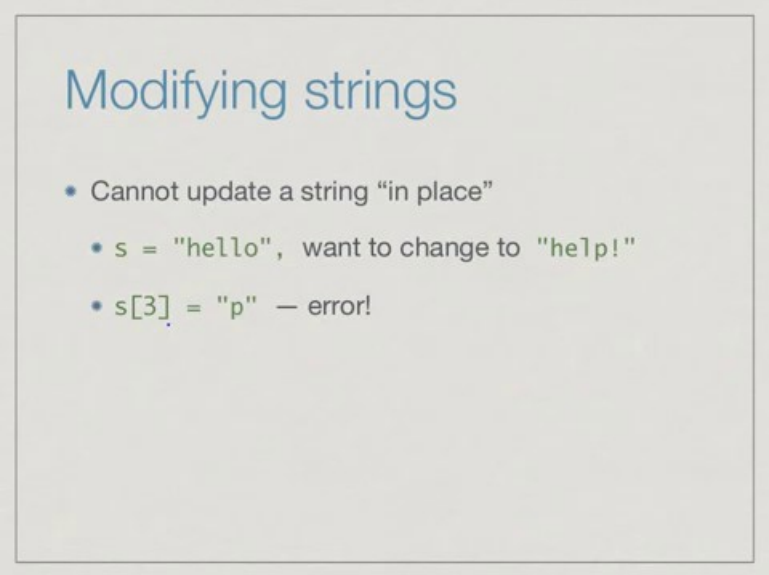
On the other hand, if I say something like go from 0 to 7, where there is no 7th position in the string, here python will not give an error instead, it will just go up to the last position which actually exists in the string below 7. So, in general these range values are treated in a sensible way, if 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)



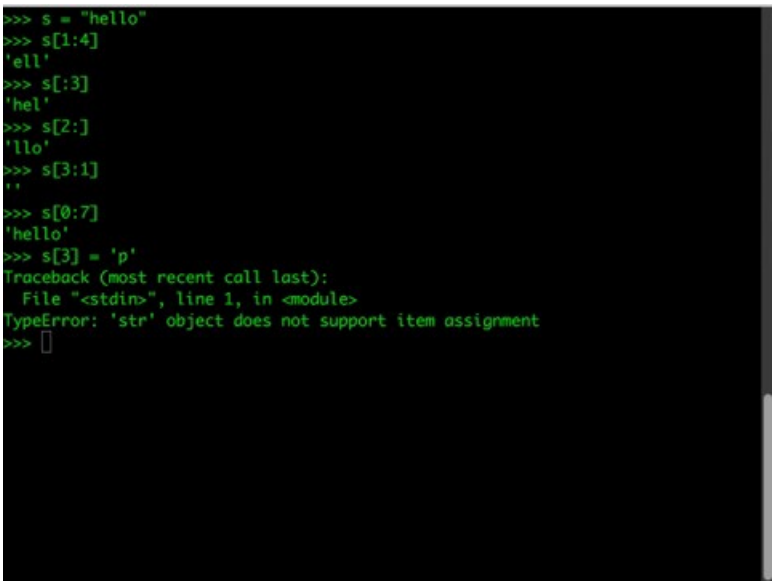
Though we have access to individual positions or individual slices or sequences within a string, we cannot take a part of a string and change it as it stands. So, we cannot update a string in place. Suppose, we want to take our string "hello" and change it to the string "help!" it would be nice if we 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)



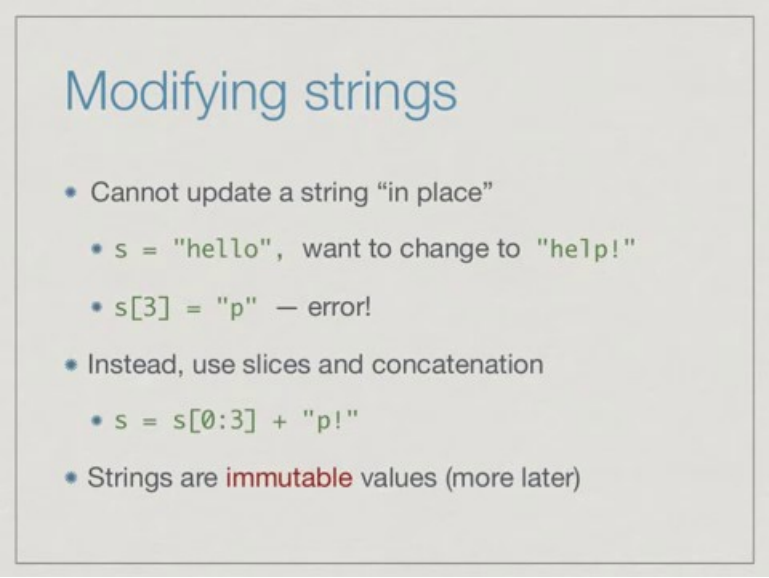
We would want to write something like change s 3, assign the value s 3 to be the string p. Now, unfortunately python does not allow this. So, you cannot update a string in place by changing its part. In fact, if you try this, you will actually get an error message, let us see.

(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)



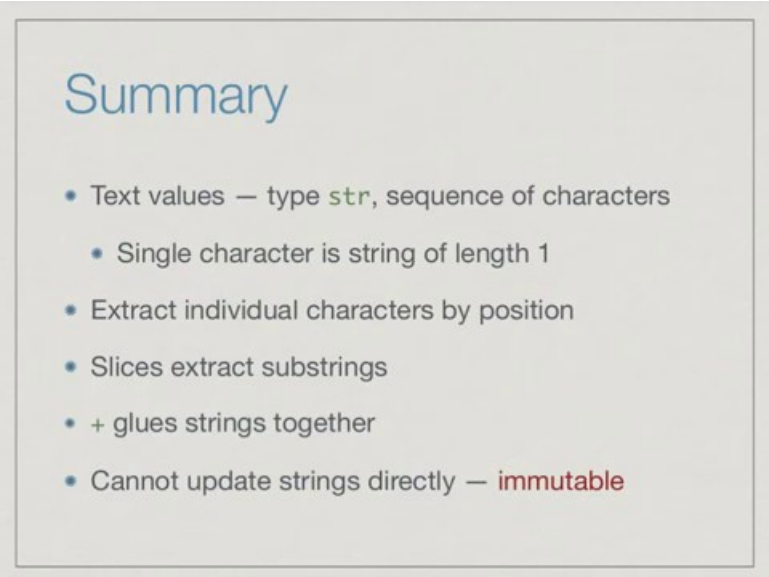
Instead of doing this, instead of trying to take a string and change the part of it as its stands what you need to do is actually construct a new string effectively using the notion of slices and concatenation. Here what we want to do is we want to take the first part of the string as it is. These are 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.

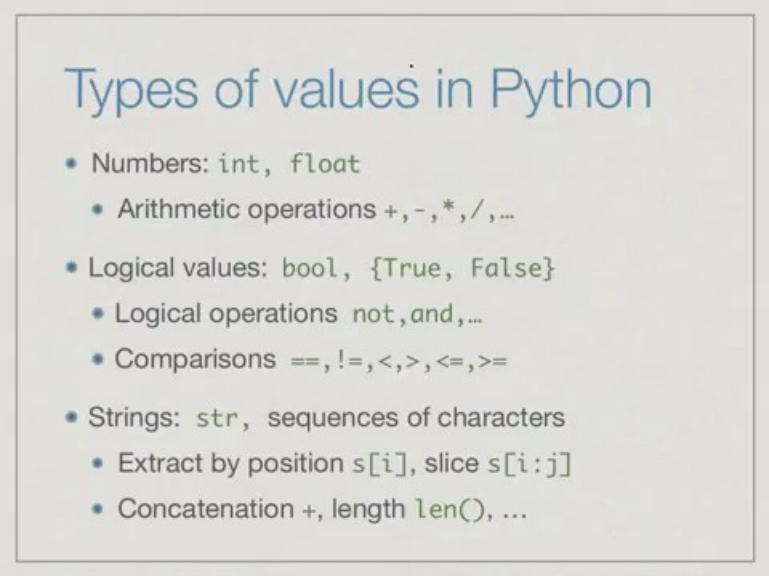
We can extract individual characters by index positions, we can use slices to extract sub strings, and we can glue strings together using the concatenation operator plus, but strings are immutable. We cannot take a value assigned to a string name and update it in place. We can create a new value by manipulating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

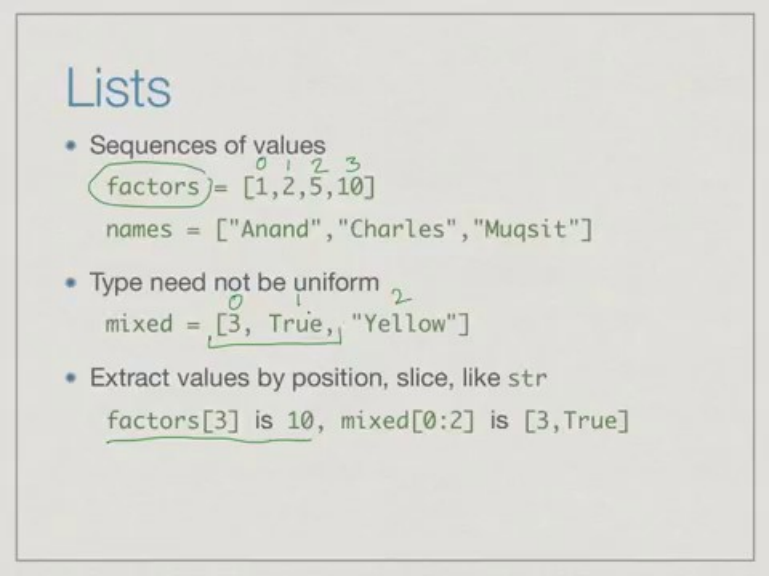


You began with the numeric types, which divided into two categories int and float. So, int represented whole numbers or integers, and float represented values which have a decimal point. And for these, we had arithmetic operations such as plus, minus, times, divide and also other functions which we can 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.

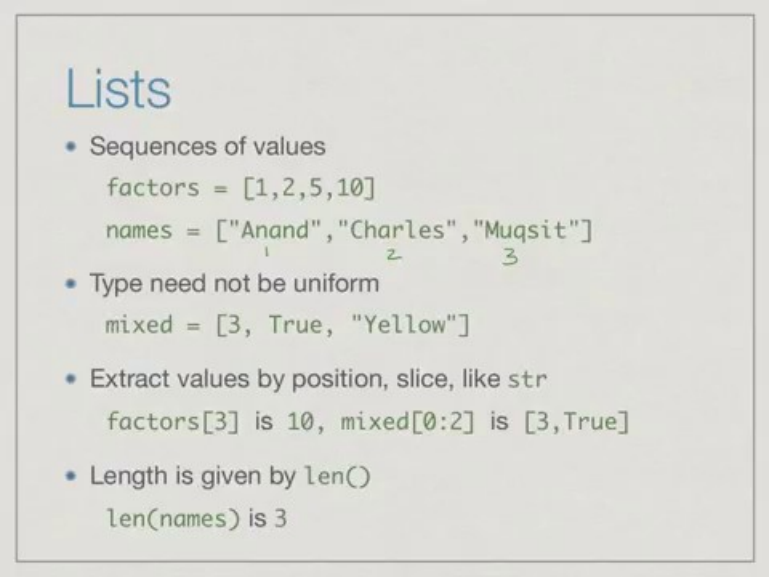
(Refer Slide Time: 01:59)



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.

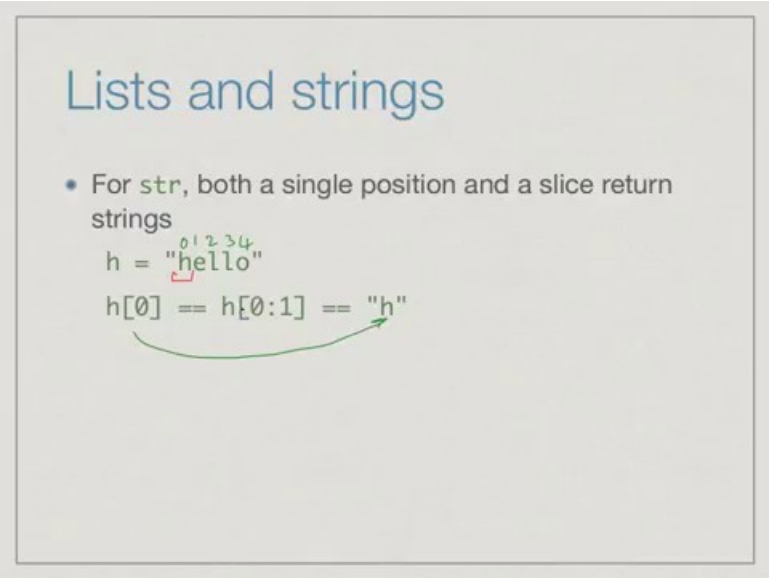
A list is a sequence in the same way as a string is and it has positions 0 to n minus 1 where n is the length of the list. So, we can now extract values at a given position or we can extract slices. In this example if we take the list factors and we look at the third position remember that the positions 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.

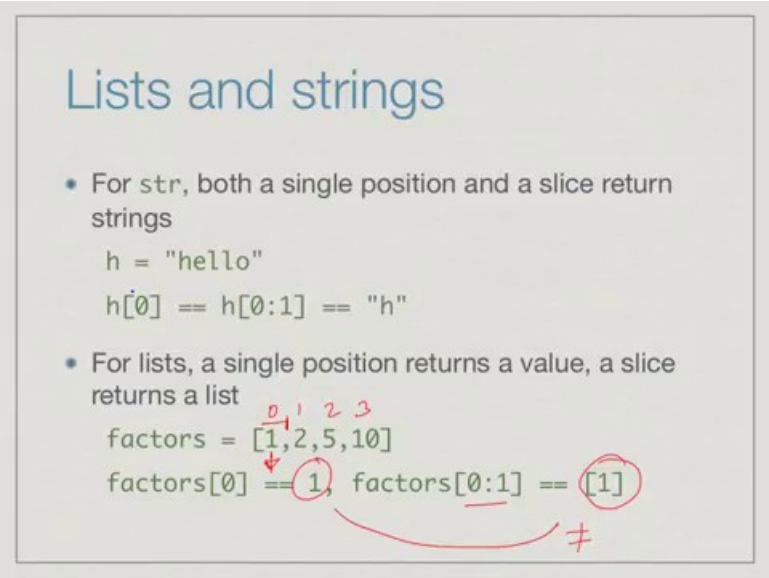
(Refer Slide Time: 03:43)



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.

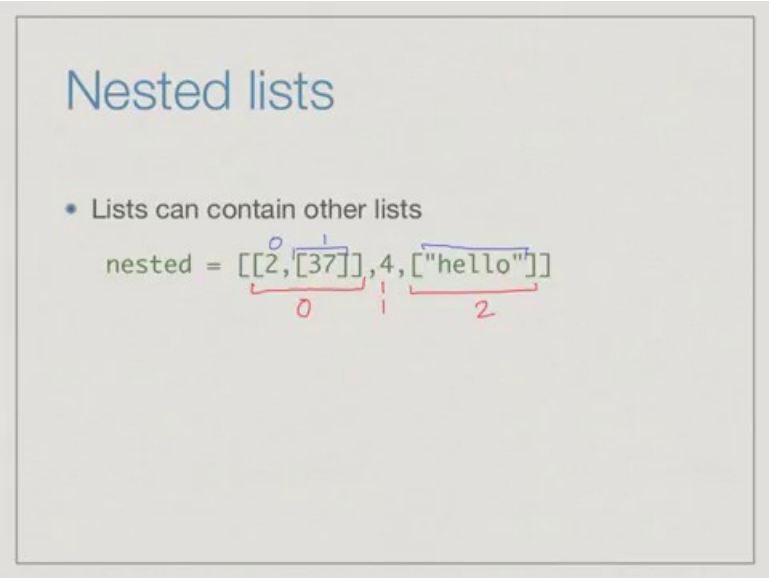
(Refer Slide Time: 04:39)



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.

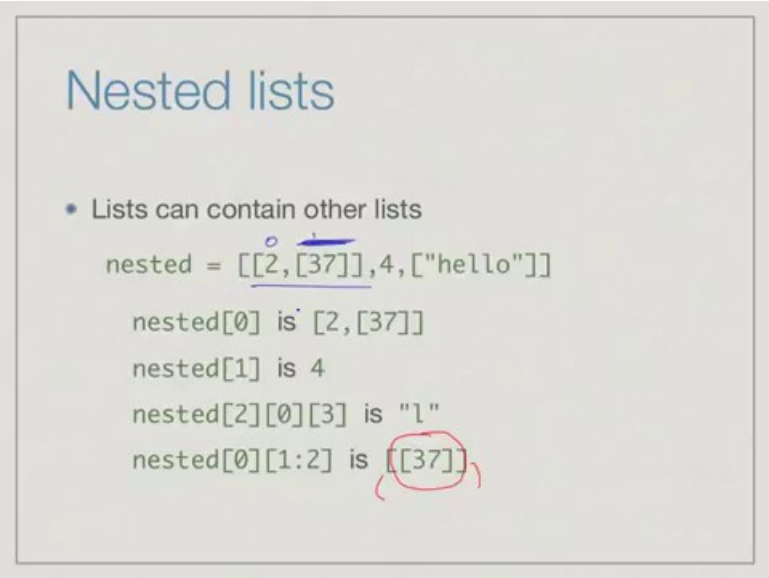
Just remember this that in a string we cannot distinguish between a single value at a position and a slice of length one. They give us exactly the same type of value and the same value itself. Whereas, in a list a slice of length one is a list whereas, a value at a position is a single value at that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

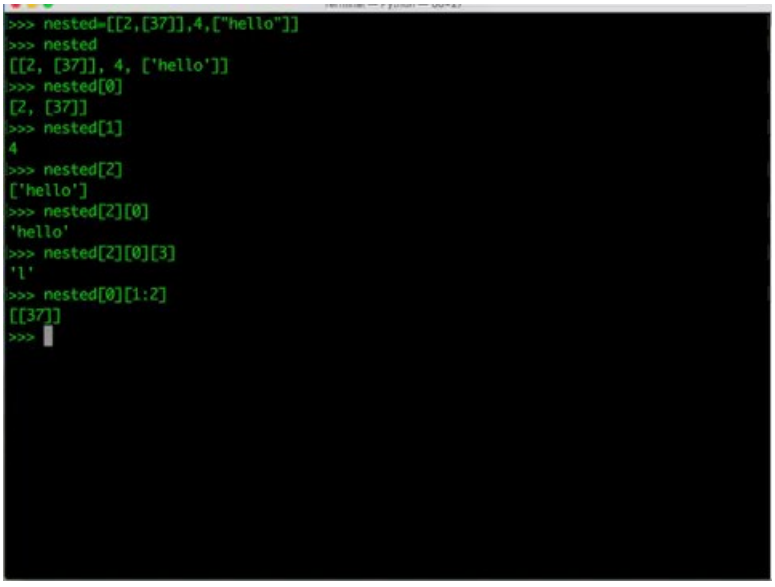


If we look at this example then we can see that if we want to look at the 0th position in nested then as we said we get this value and this value consist of a list itself containing 2 and the list containing 37. On the other hand, if we ask for the first position number 1 then we get the value 4. And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

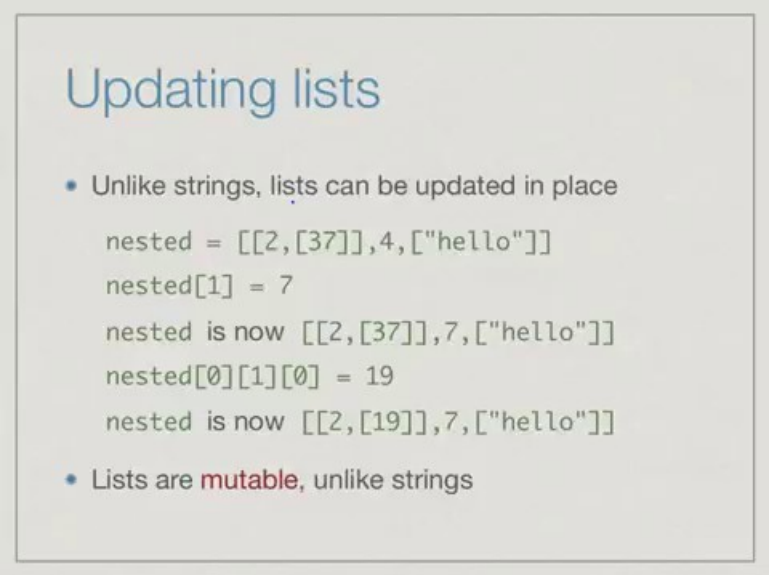
(Refer Slide Time: 08:18)



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)



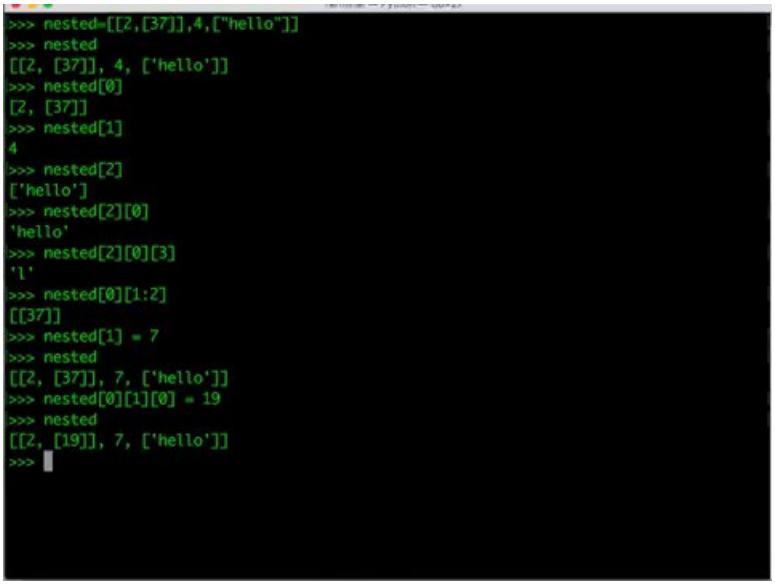
One fundamental difference between list and string or indeed any of the values we have seen before is that a list can be updated in place. So, if we have a list nested as we had before and we say nested of 1 is 7. Remember when we try to change a position in a string we got an error. We cannot change 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.

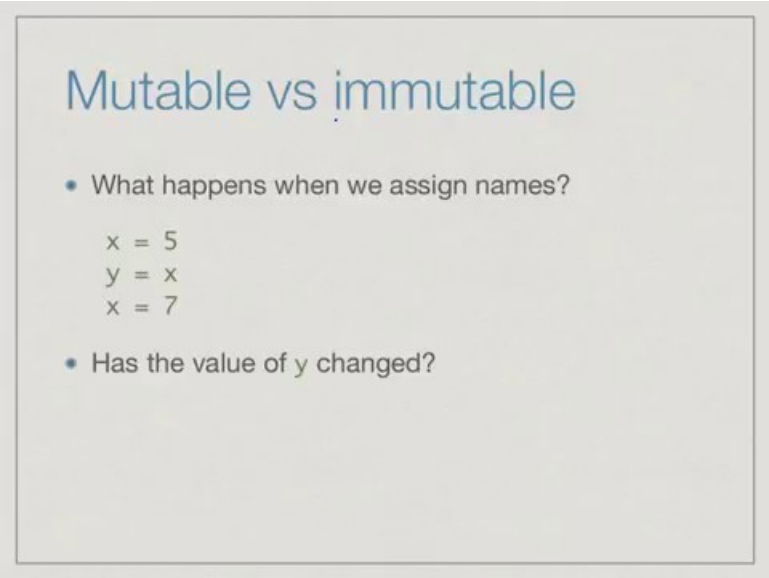
A list can be transformed in place we can take a list and change its structure unlike a string if we try to change a string we have to actually construct a new string and re assign the name, but in a list with the same name we can update parts of it without affecting the other parts.

(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)



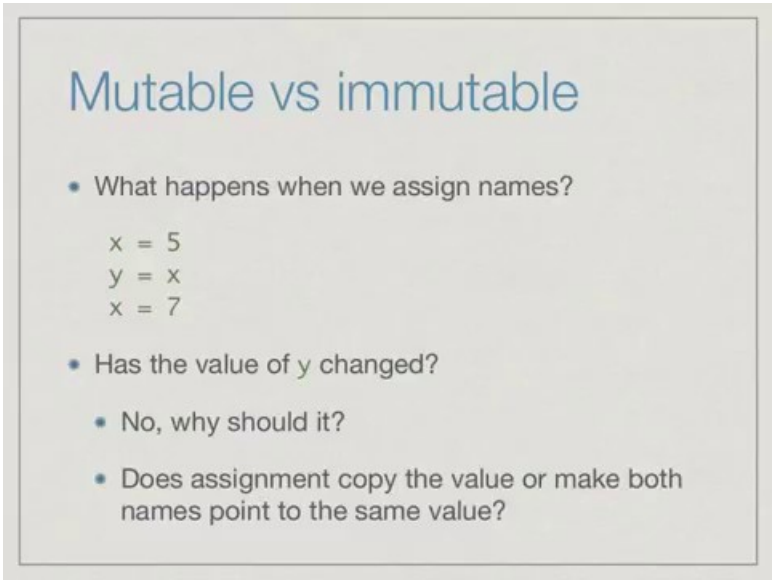
It is important to understand the distinction between mutable and immutable values, because this plays an important role in assignment. And as you will later see, it also plays a major role in what happens when we pass values to functions as arguments. Let us look at what happens when we assign names.

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.

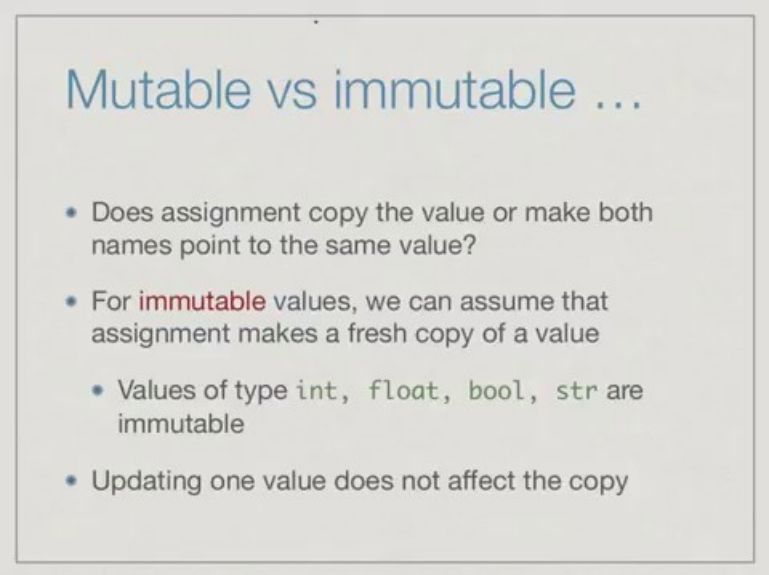
(Refer Slide Time: 13:07)



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.

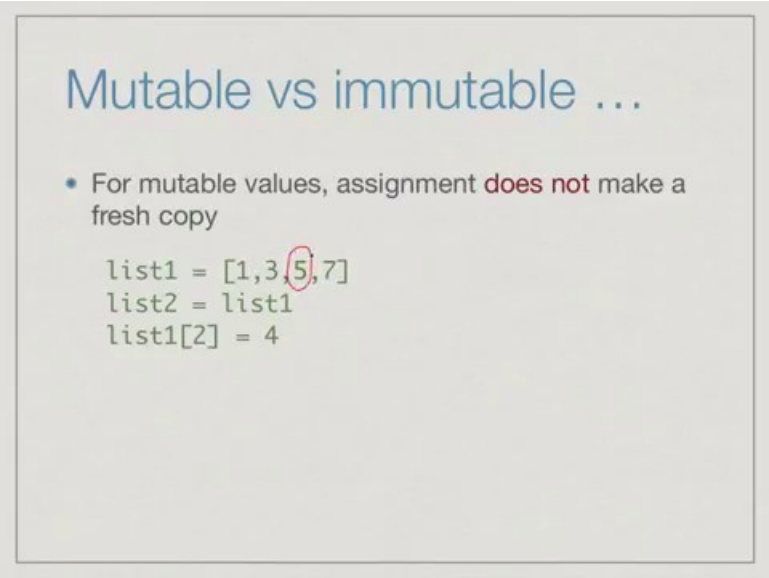
(Refer Slide Time: 13:42)



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.

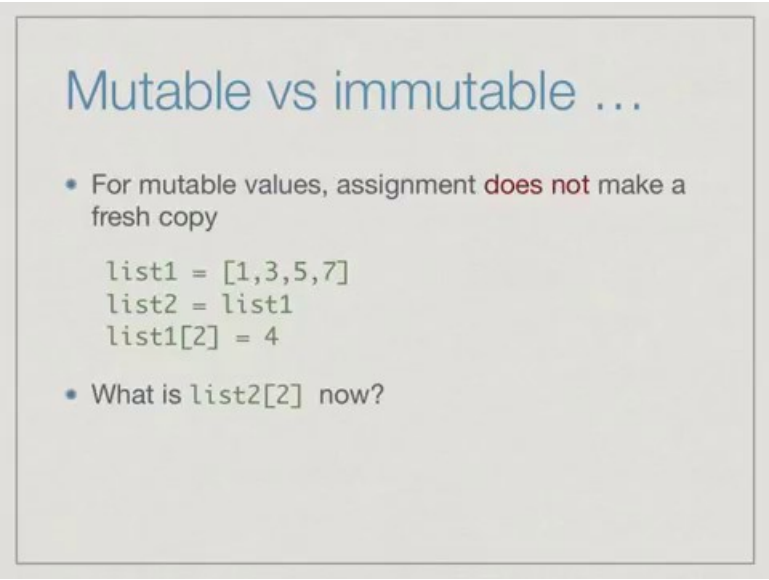
This applies to all the types we have seen before today's lecture namely int float bool and string these are all immutable. If we do the kind of assignment we did before where we assign something to x then make y is the same value was x and then update x, y will not change. Updating one value does not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

(Refer Slide Time: 14:30)



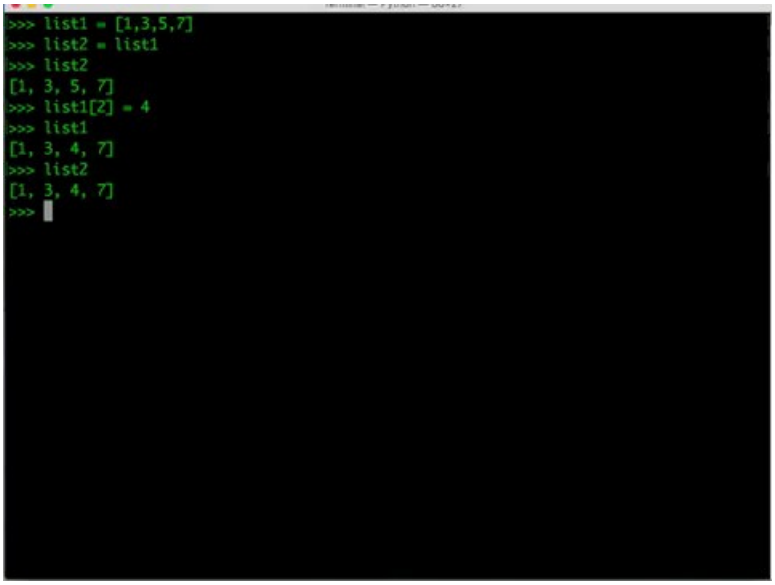
However as we are pointed out lists are difference beast from strings and list are mutable. It turns out that for mutable values assignment does not make a fresh copy. Let us look at the following example we first assign say the list 1, 3, 5,7 to the name list1, then we say that list2 is the same as 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)



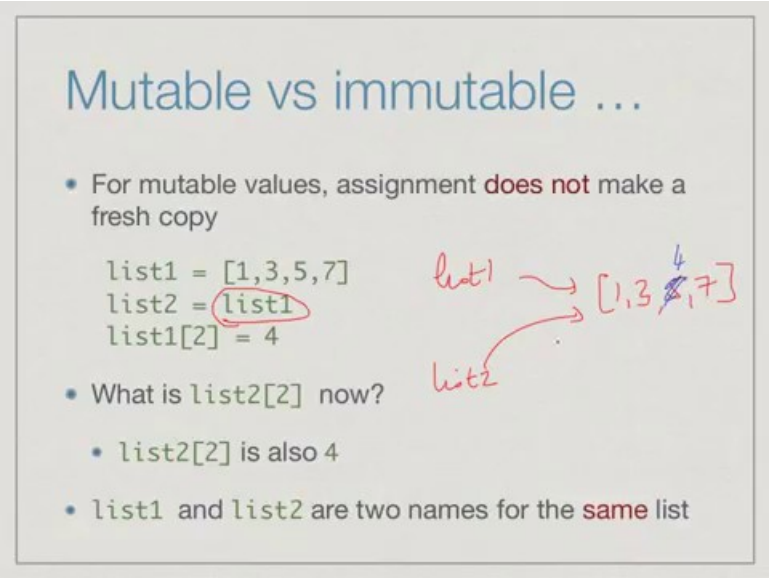
The question is what has happened to list2, is list2 the same as before namely 1, 3, 5, 7 or as list2 also become 1, 3, 4, 7 like list1. So, lets us see what happens in the interpreter.

(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.

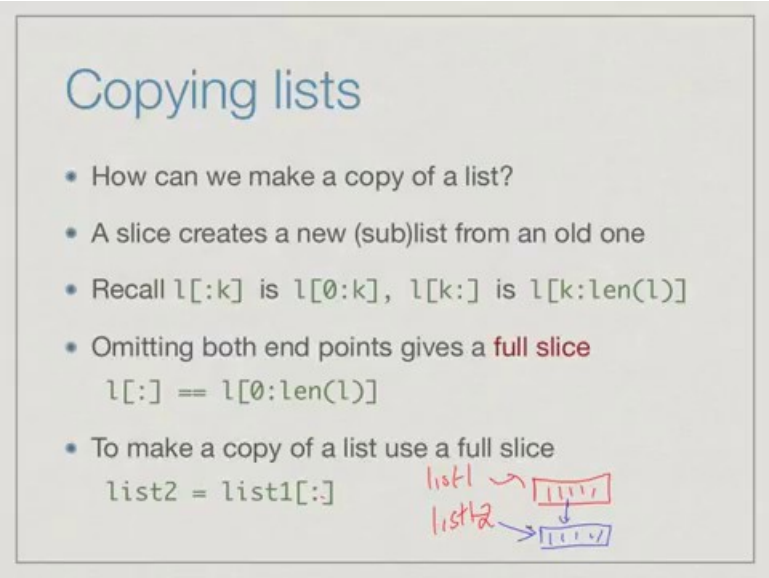
(Refer Slide Time: 16:05)



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.

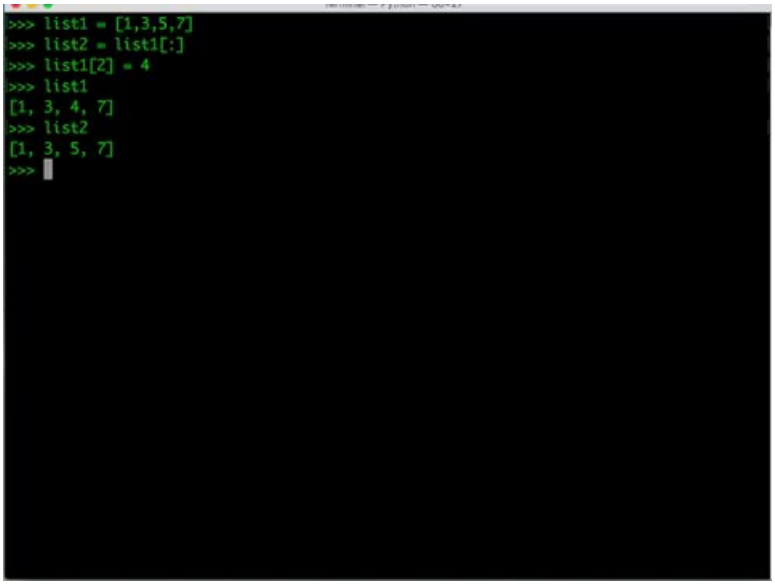
If we leave out the first position, we get a 0; if we leave out the last position, we get the length. If we leave out both position, we just put colon with nothing before nothing after logically this becomes 0 and this becomes the length. We have both characteristics in the same thing and we call this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

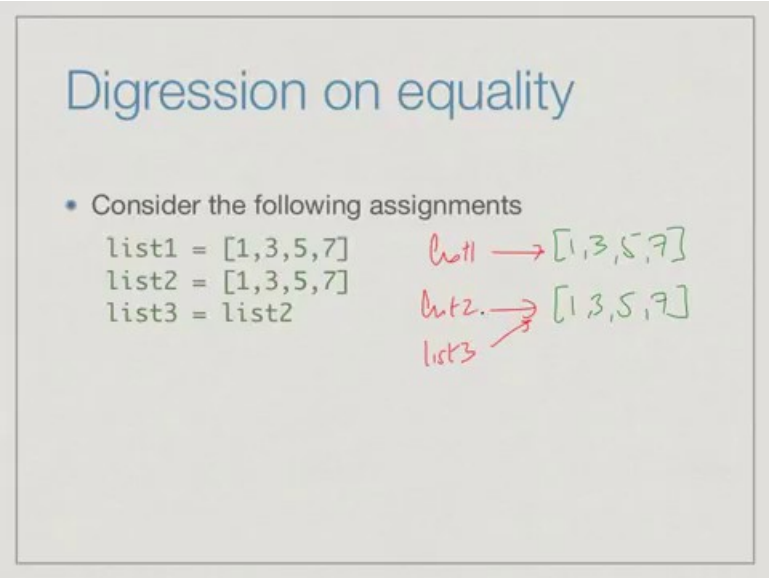
Remember if I do not have this then I will get list1 and list2 pointing to the same actual list. There will be only 1 list of values and will point to the same. But if I do there have this then the picture changes then what happens is that the slice operation produces a new list which has exactly the 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.

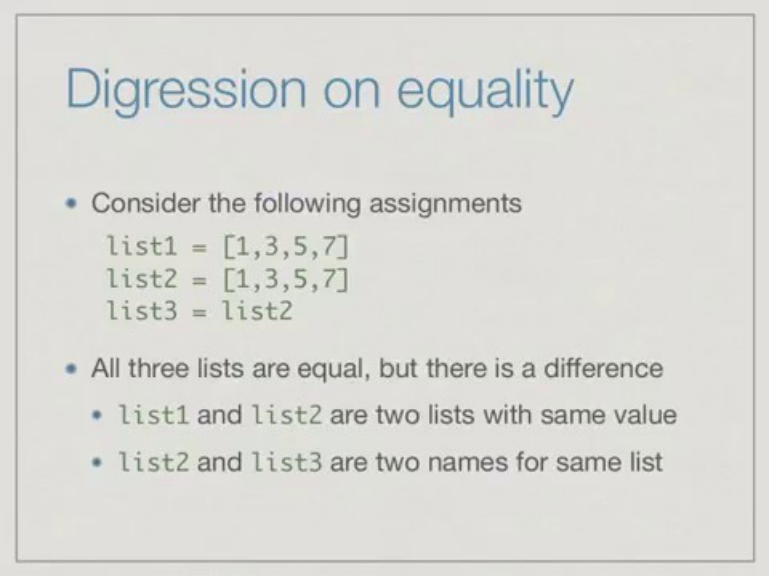
(Refer Slide Time: 20:19)



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.

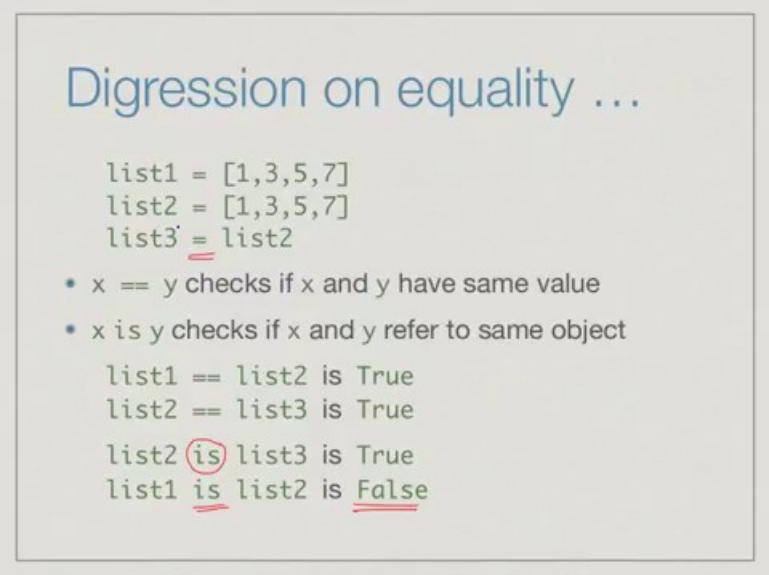
(Refer Slide Time: 21:10)



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)



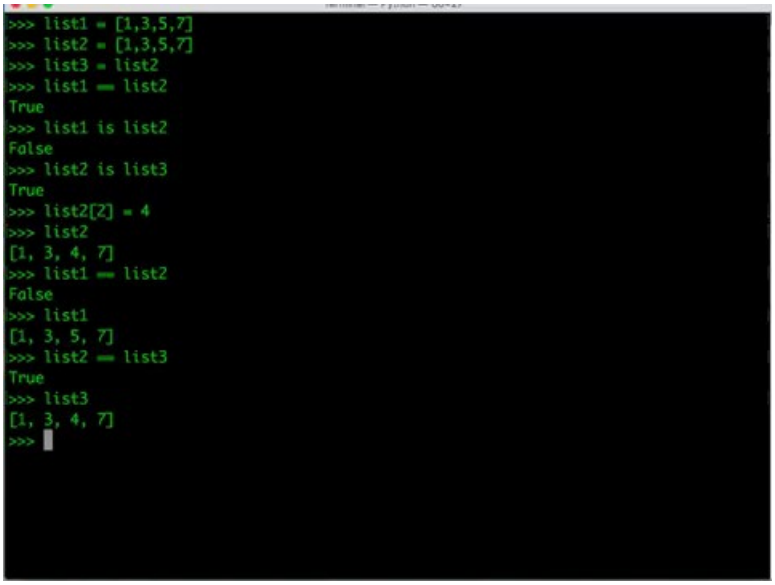
Python has we saw this operation equal to equal to, which is equivalent or the mathematically equality which checks if x and y as names have the same value. So, this will capture the fact that list1 is equal to list2 even though they are two different lists they happen to have the same value.

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)

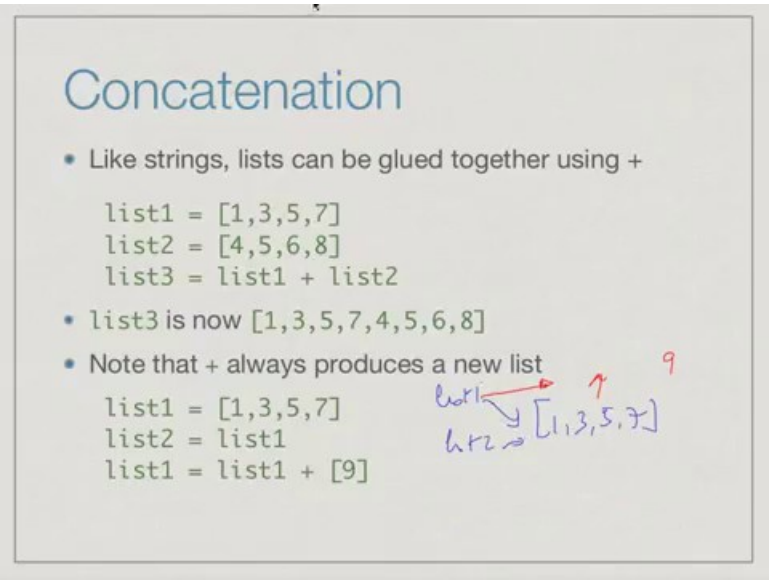


Let us type out those three lines of python in the interpreter. So, we say list1 is 1, 3, 5, 7 list2 is also 1, 3, 5, 7 and list3 is list2. Now, we ask whether list1 is equal to list2 and it indeed is true, but if we ask whether list1 is list2 then it says false. So, this means that list1 and list2 are 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.

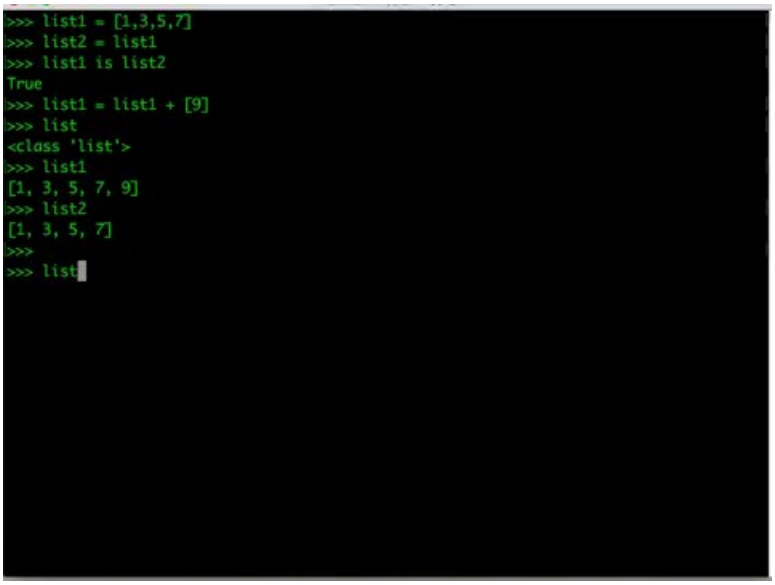
(Refer Slide Time: 24:54)



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.

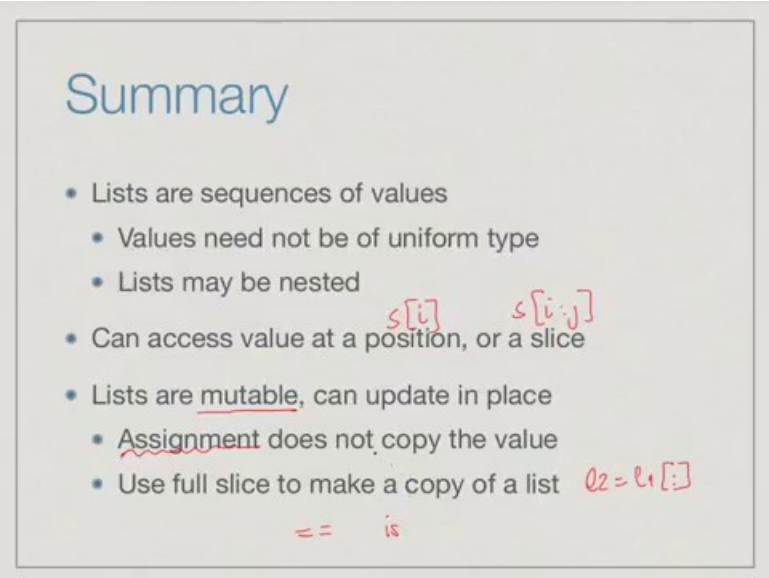
(Refer Slide Time: 26:00)



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

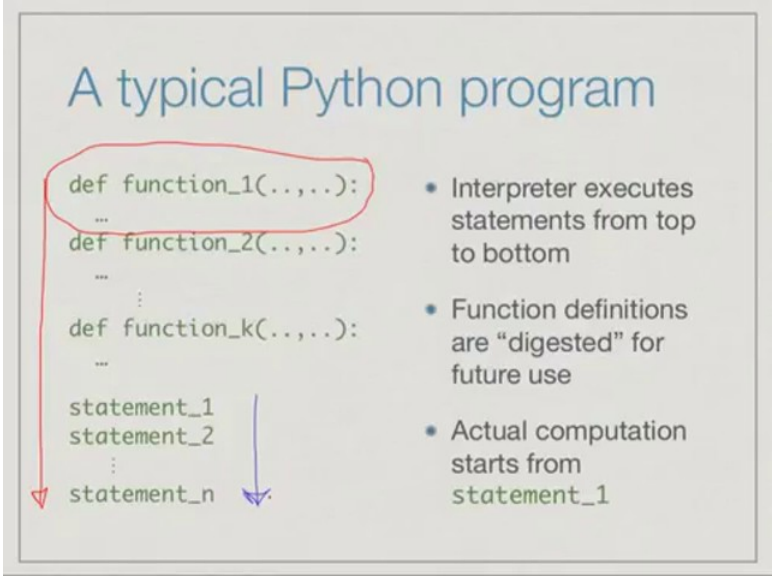
a fresh list with exact same contents as l1. And finally, we saw that we can use equality and is as two different operators to check whether two names are equal to only in value or also are physically pointing to the same type.

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)

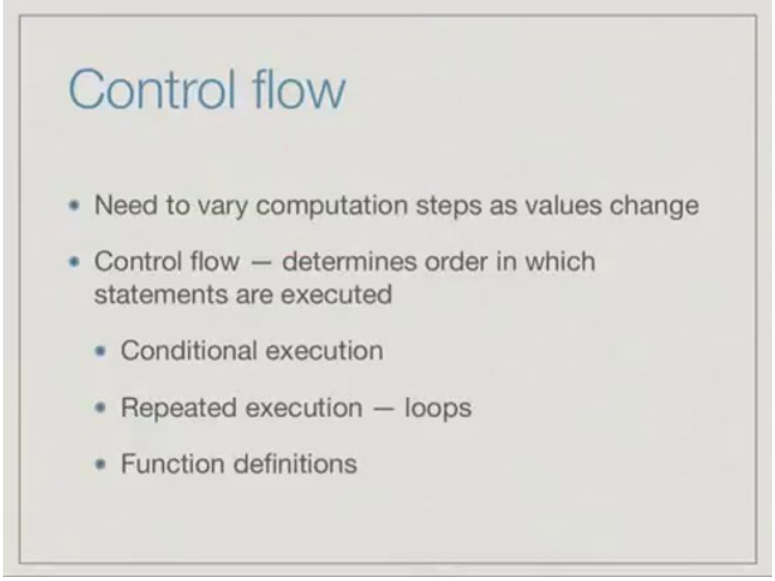


Now let us go back to the order in which statements are executed. Remember we said that a typical python program will have a collection of function definitions followed by a bunch of statements. In general, the python interpreter will read whatever we give it from top to bottom. So, when it sees the 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)



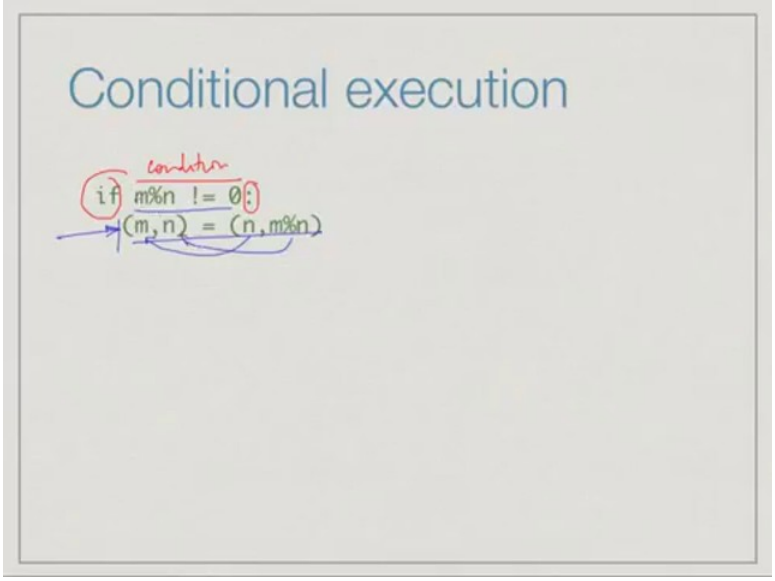
Now in many situations, many realistic situations, we need to vary the next step according to what has happened so far, according to the current values that we see. Let us look at a real life example. Supposing, you are packing your things to leave for the bus stop in the morning, or whether or not you 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.

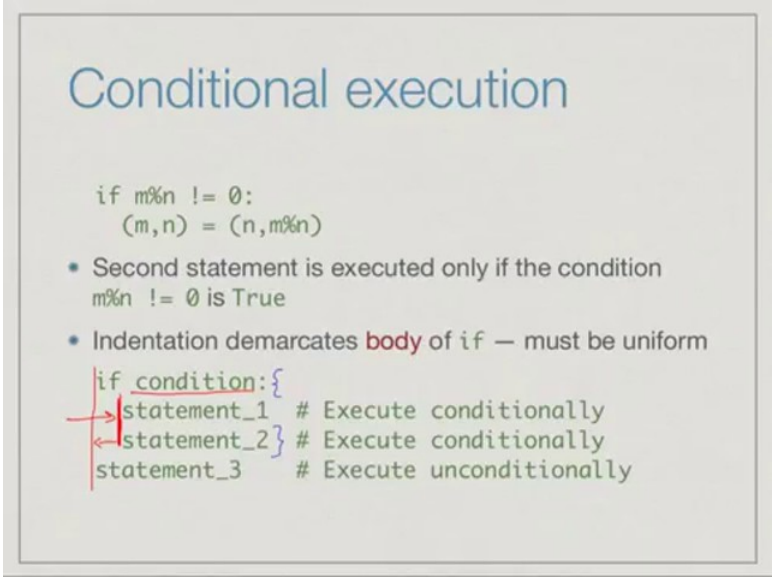
(Refer Slide Time: 03:17)



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.

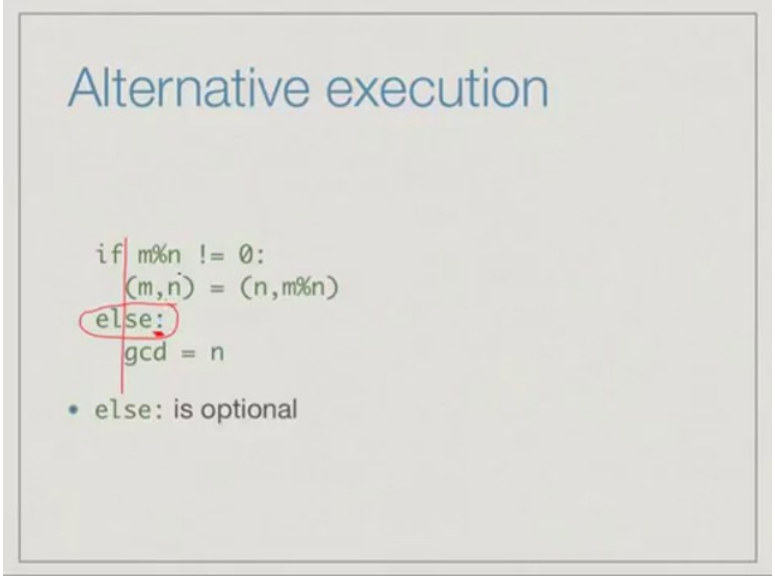
In a conventional programming language, you would have some kind of punctuation typically something like a brace to indicate the beginning and the end of the block, which

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.

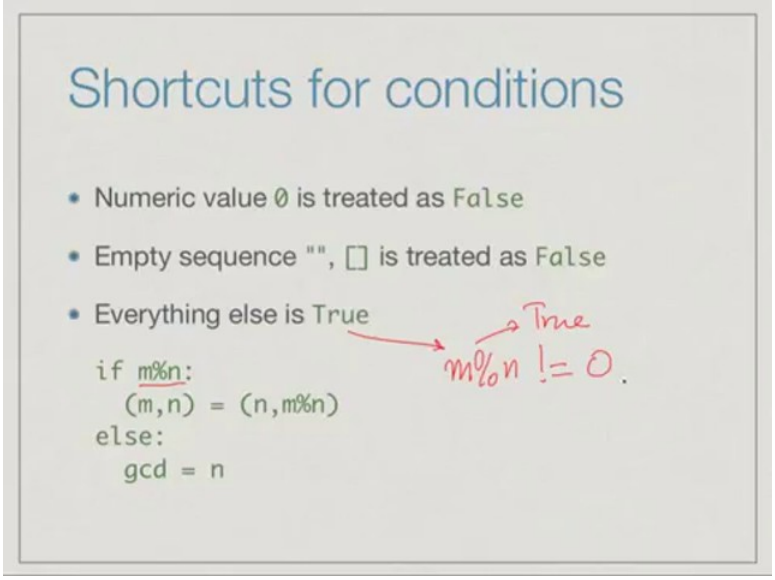
So, one tab is not going to be equal to three spaces or four spaces or whatever you see on the screen; and the more worried thing is the python will give you some kind of error message which is not very easy to understand. So, it is quite useful to not get into the situation by remembering to always 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)



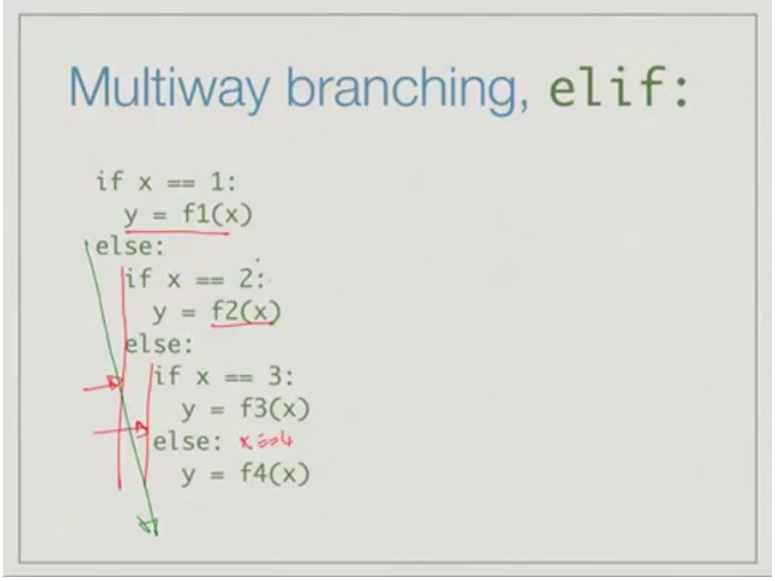
Technically speaking, the condition that we put into an 'if statement' must be an expression that explicitly returns Boolean value true or false. But like many other programming languages, Python relaxes this a little bit and allows different types of expressions which have different values like the 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)

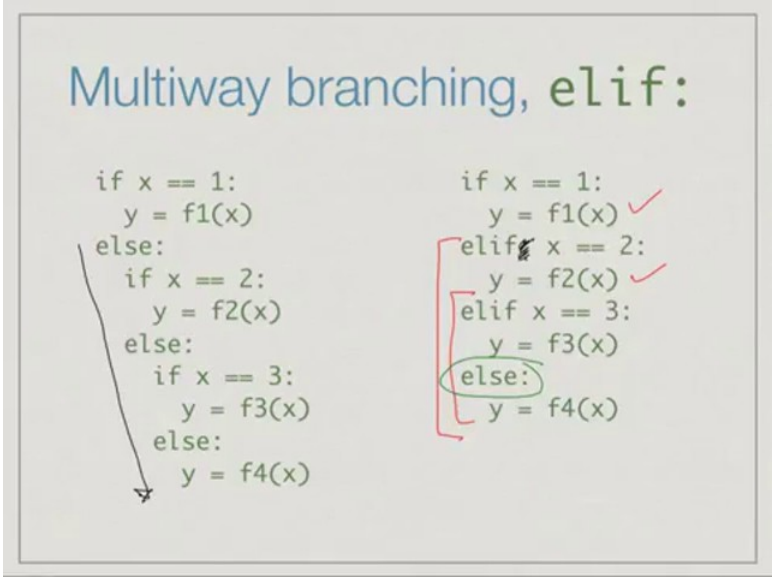


Here is a very common situation that occurs; sometimes we do not want to check between one of two conditions, but one of many conditions. So, supposing we have a value x, name x which can take a value 1, 2, 3 or 4 and depending on whether it is 1, 2, 3 or 4, you want to do four different things with 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.

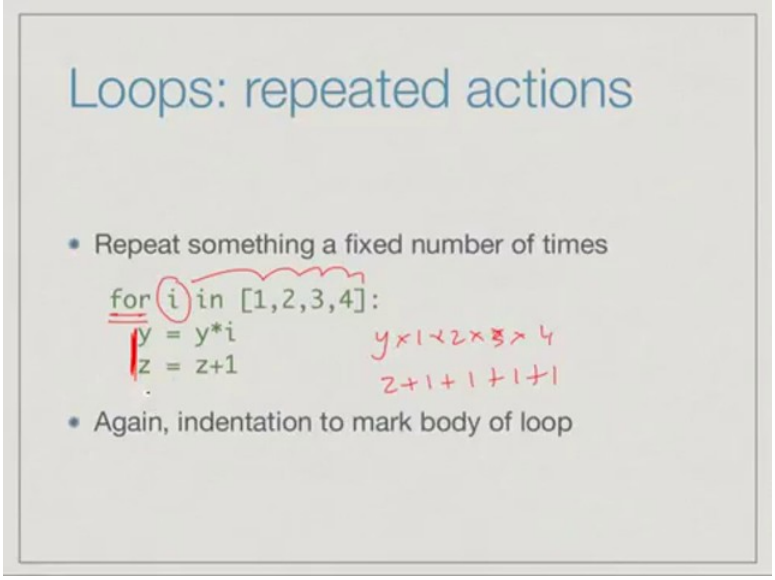
(Refer Slide Time: 10:44)



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.

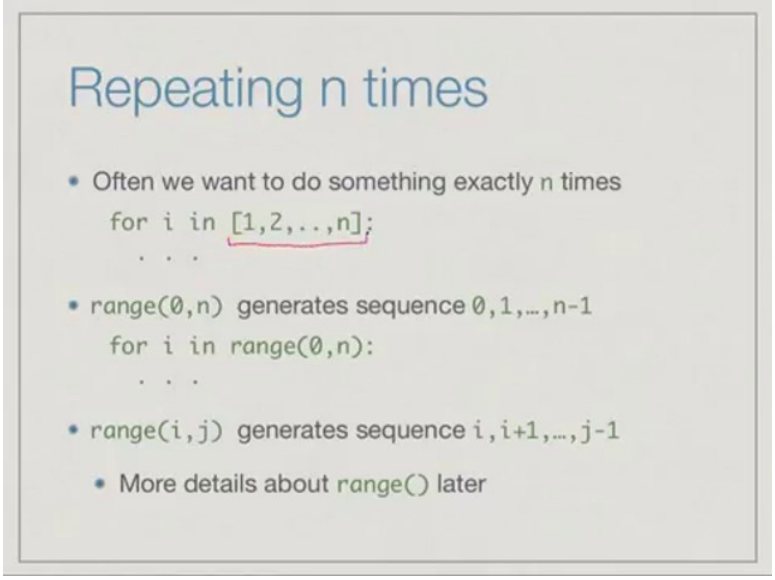
(Refer Slide Time: 12:07)



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)

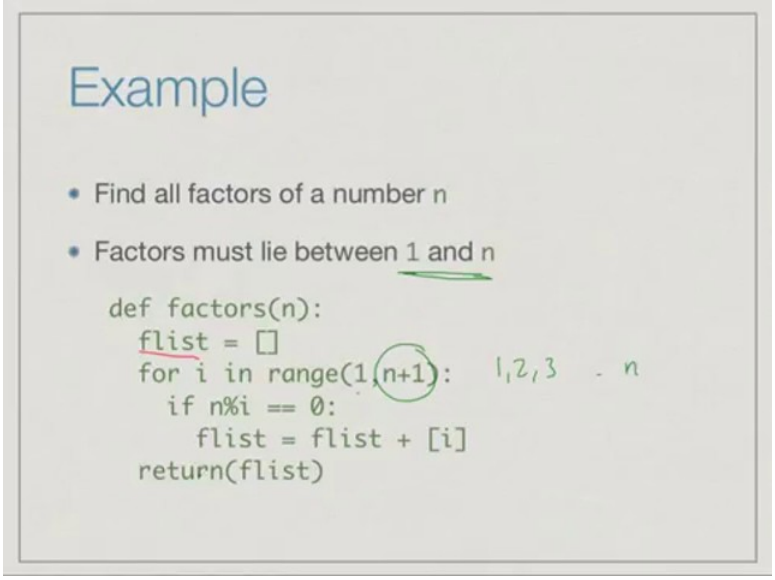


The most common case for repeating things is to do something exactly n times. So, instead of writing out a list 1 to n, what we saw is that we can generate something equivalent to this list by using this built in function range. The range 0, 1, we said it starts at 0 and it generates the sequence of 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)

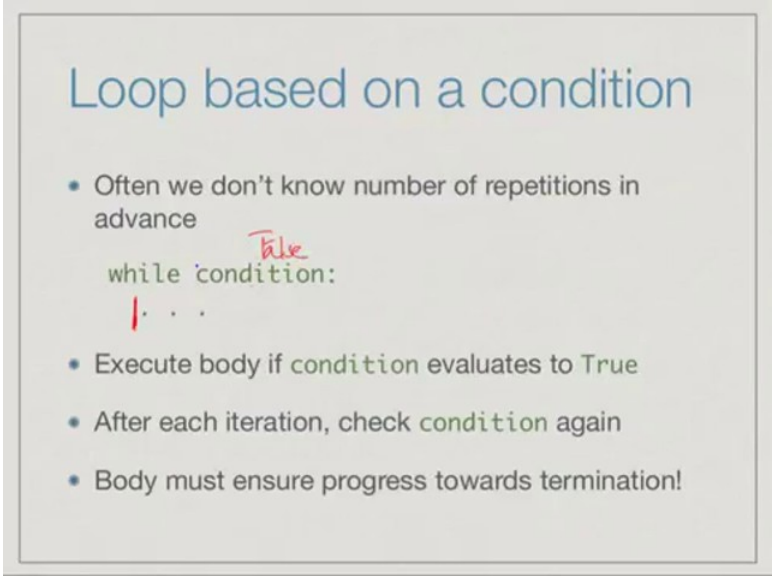


Let us look at a simple example of this. Suppose, we want to find all the factors of a number n, all numbers that divide n without a remainder. As we recalled when we did gcd, all the divisors or factors of a number must lie between 1 and n; we cannot have any number bigger than n, which is a factor 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.

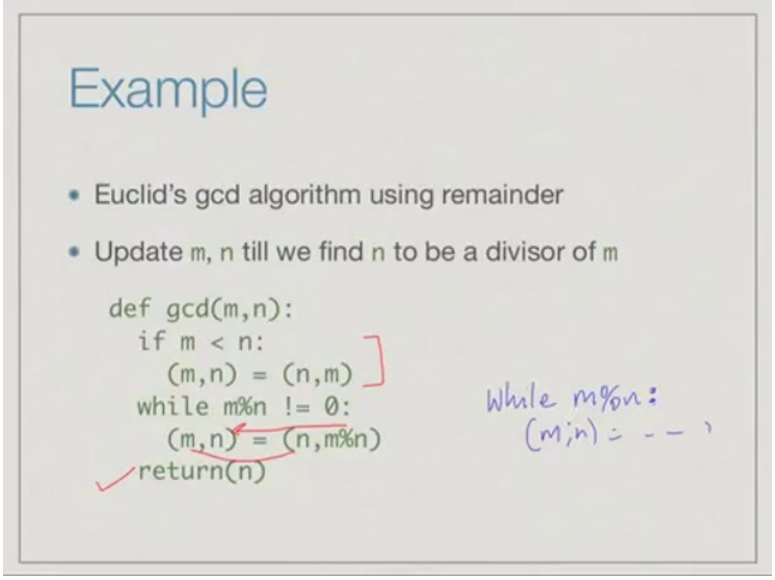
(Refer Slide Time: 16:07)



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.

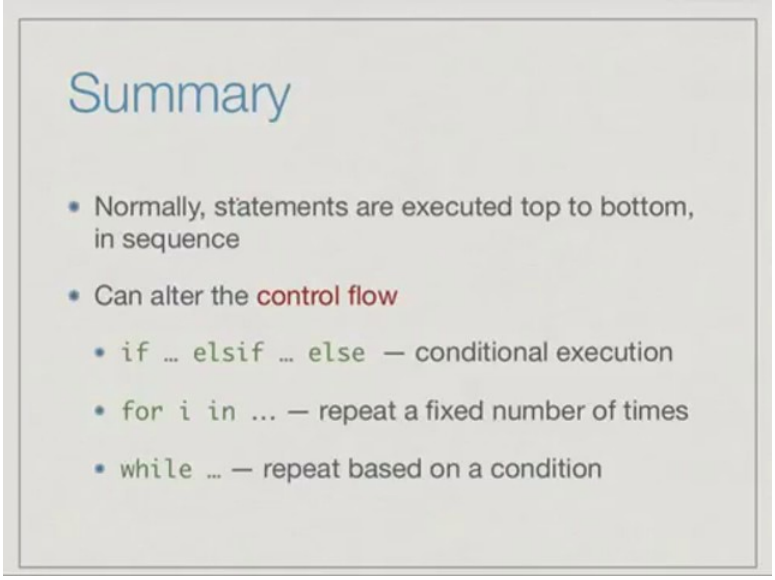
(Refer Slide Time: 17:29)



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)



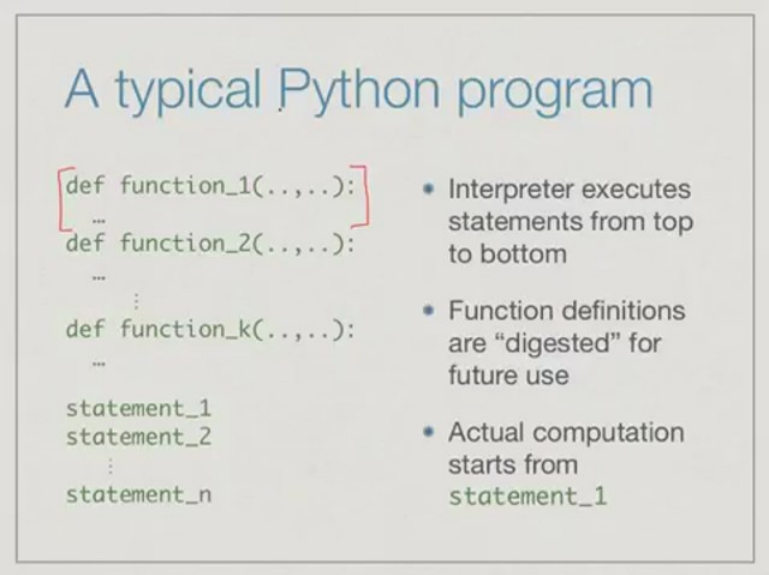
To summarize, a Python interpreter normally executes code from top to bottom in sequence. We can alter the control flow in three ways we have seen. One is using the 'if statement', which conditionally executes and this extends to the elif which allows us to do a multi-way branch. Then there is for statement 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

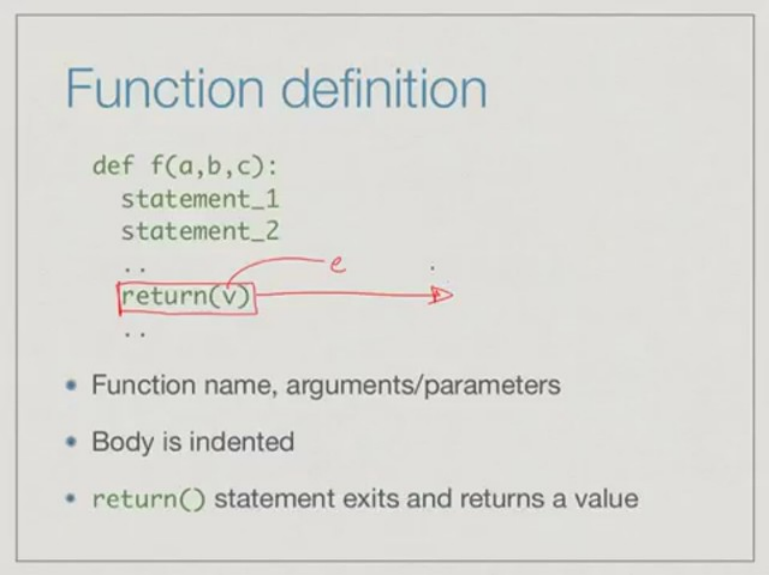
We have seen how to alter the flow of a program by using if, for and while. We can have conditional execution, we can have repeated execution.

(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.

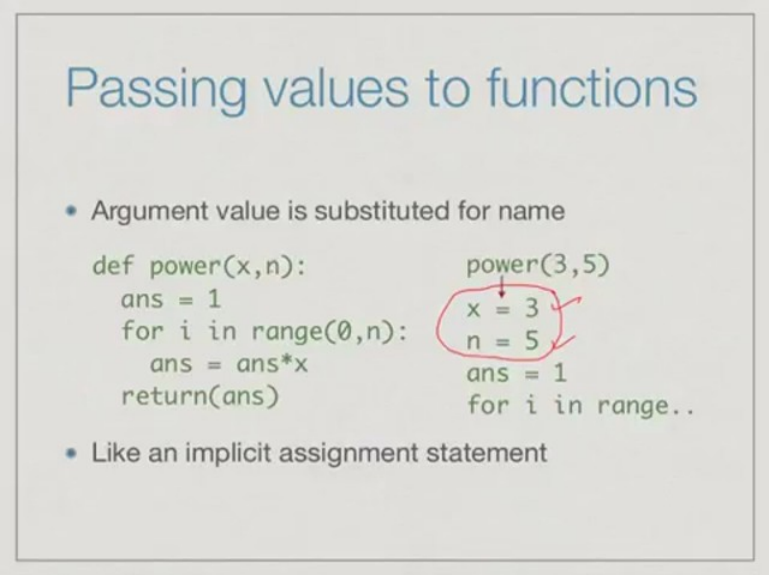
(Refer Slide Time: 00:43)



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)

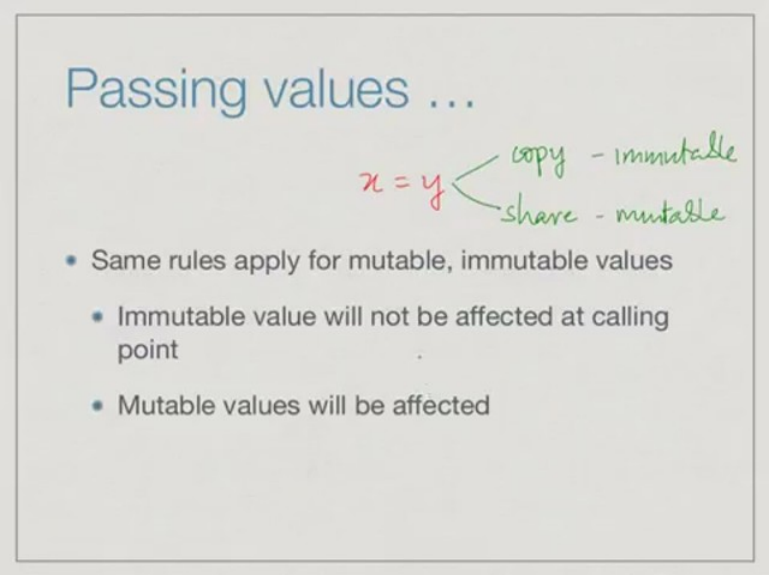


When we call a function we have to pass values for the arguments, and this is actually done exactly the same way as assigning that value to a name. Suppose, we have function like this which takes x and raises it to the power n. Let us just look at the function just to understand what the code is doing. 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.

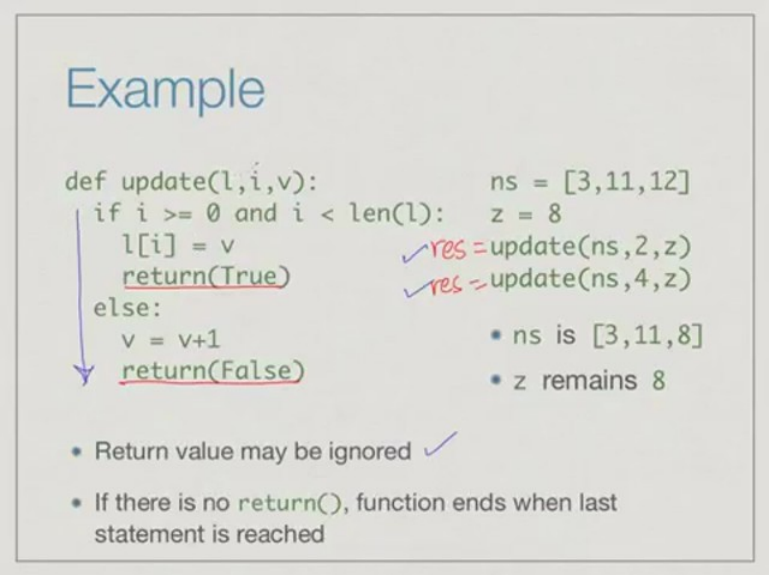
(Refer Slide Time: 03:32)



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.

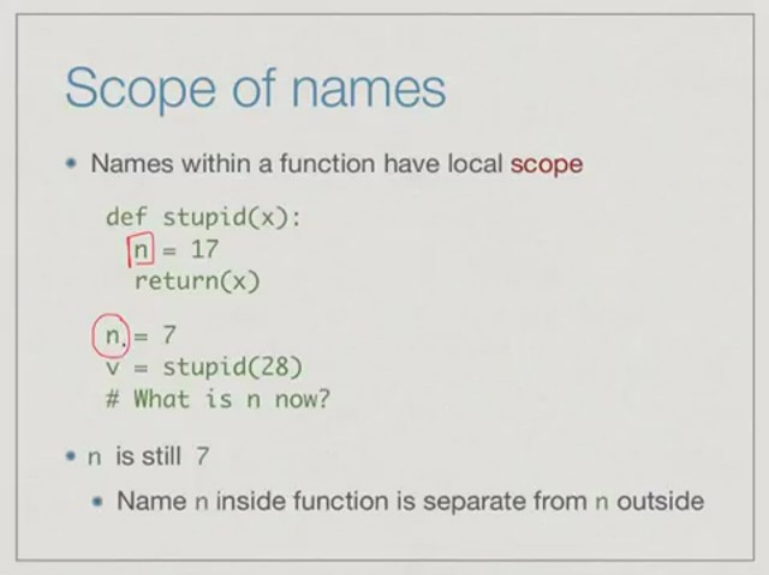
Remember update will update the list or not update the list depending on whether the index is valid and it will return true or false depending on whether they update succeeded. So, by examining the value of whatever is returned we can check whether the update we intended worked or not. This is something 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)



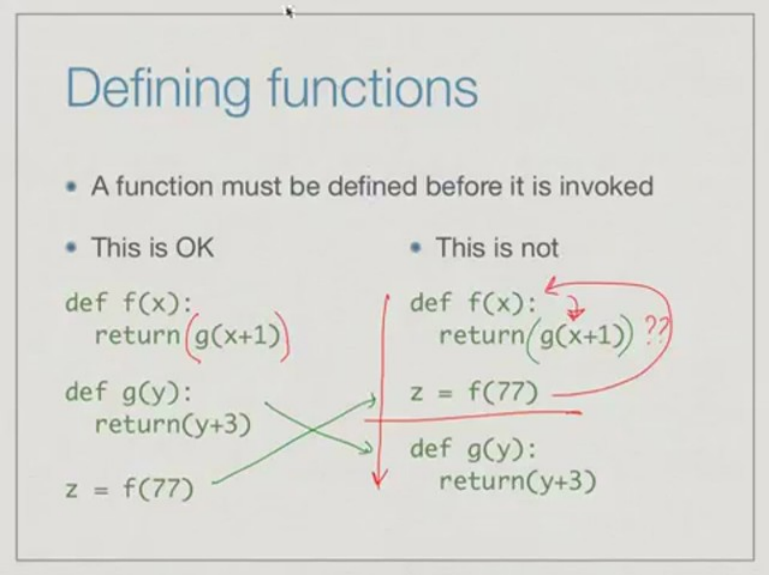
Another point to note about functions in Python is that names within a function are disjoint from names outside a function. So let us look at again at a kind of toy example which does not have anything useful to do. We have a function which we call stupid which takes essentially takes an argument and 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.

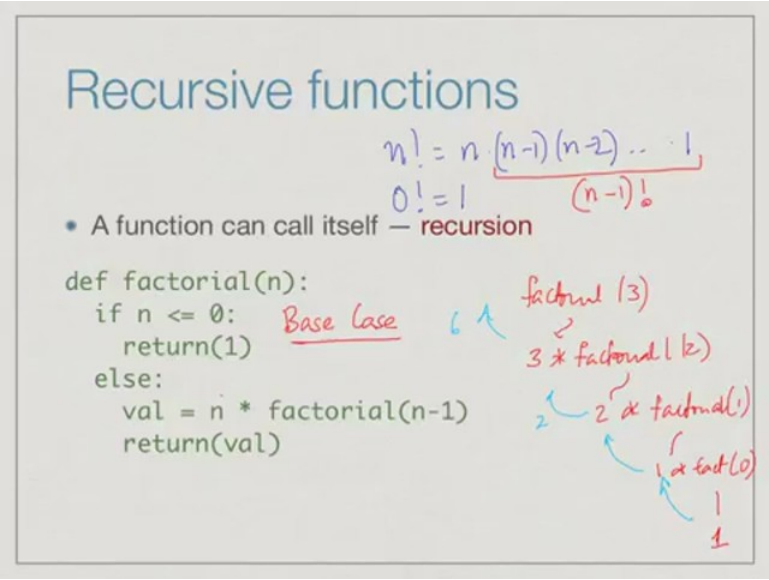
So, it is only when I execute the statement f is executed at that time g has already been

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)



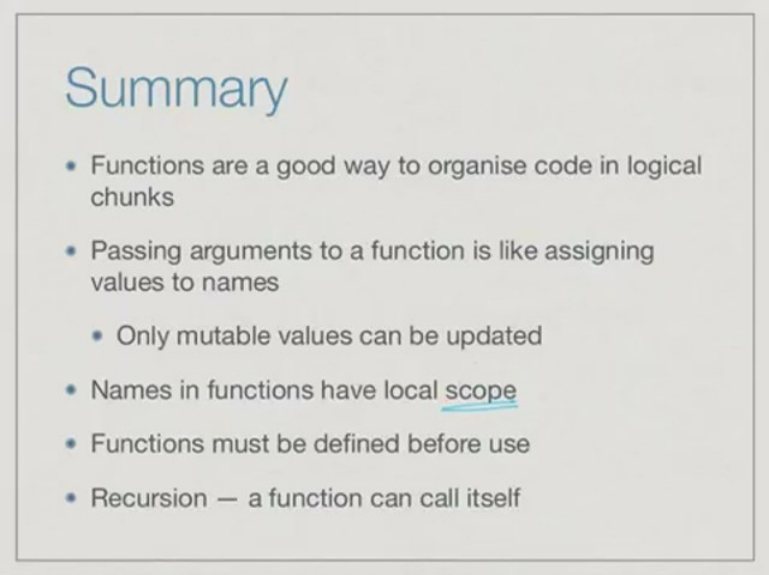
A final point that we will return to later when we go through more interesting examples as we proceed in programming, is that a function can very well call itself. The most canonical function of this kind, these are called Recursive functions. Functions which rely on themselves. Is the factorial function. 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

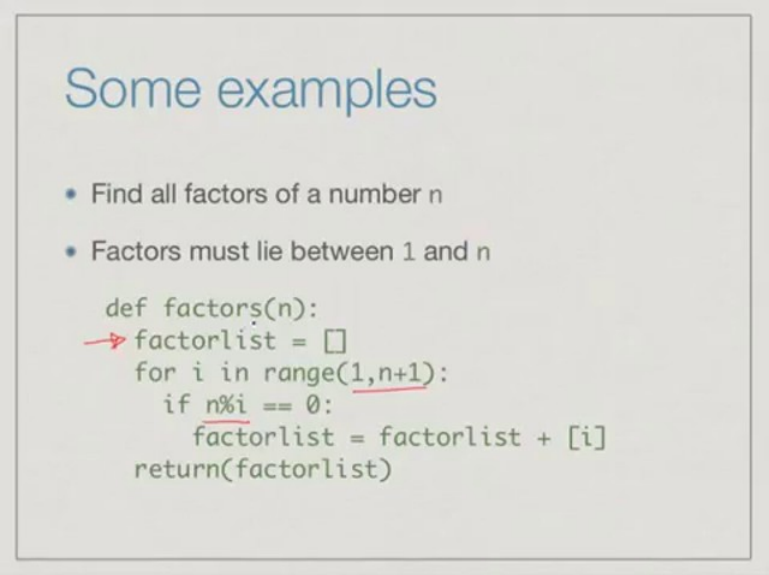
weeks to come.

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

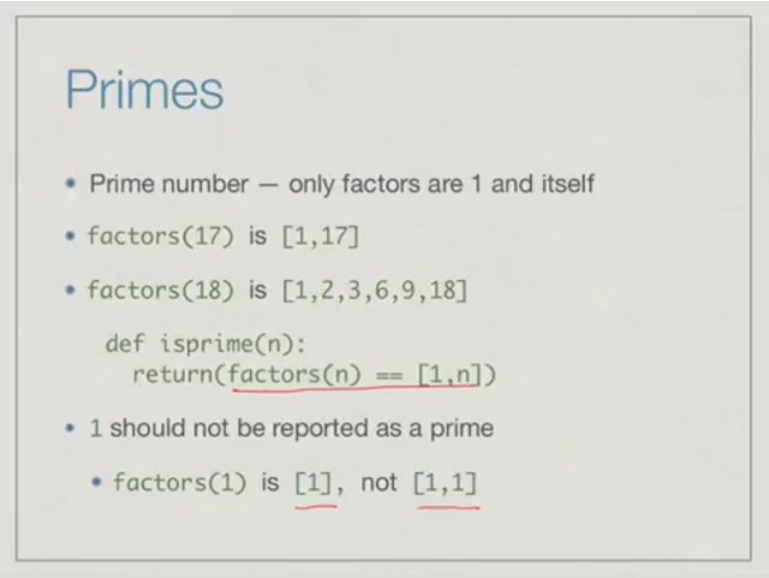
To round of this week, let us look at some examples to illustrate some of the concepts we have seen so far.

(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)



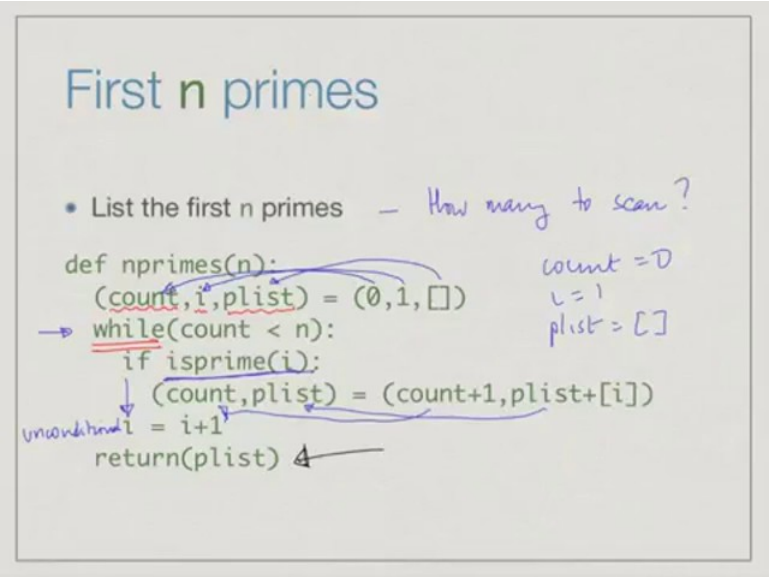
A prime number is one which is divisible by no other number other rather than 1 and itself. In other words, the only factors of n should be 1 and n, if n is a prime. So, 17 for example, which is a prime number has only two factors 1 and 17. Whereas, 18 which is not a prime have many more factors, it 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.

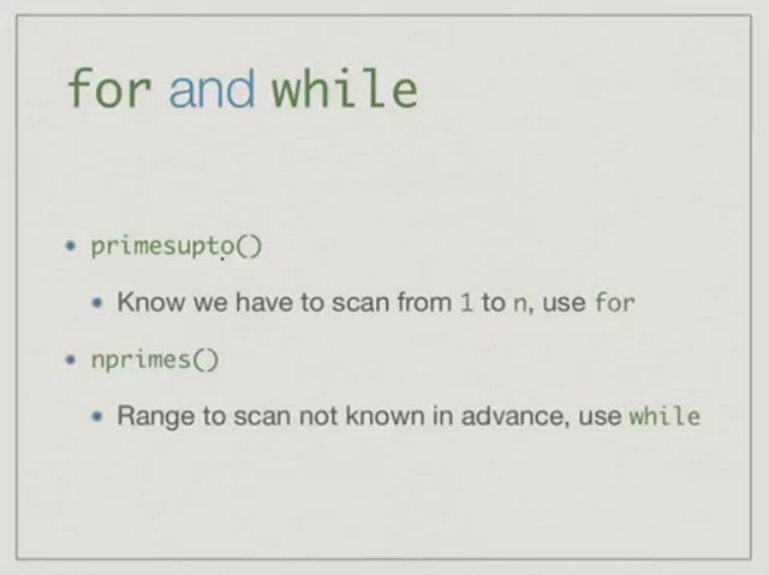
So, what we need to do is we need to keep a count of how many primes we have seen. We need to go through all the numbers one at a time to check if they are primes, and we need a list of all the primes we have seen so far. Just to illustrate another point that we have seen this is a simultaneous assignment 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.

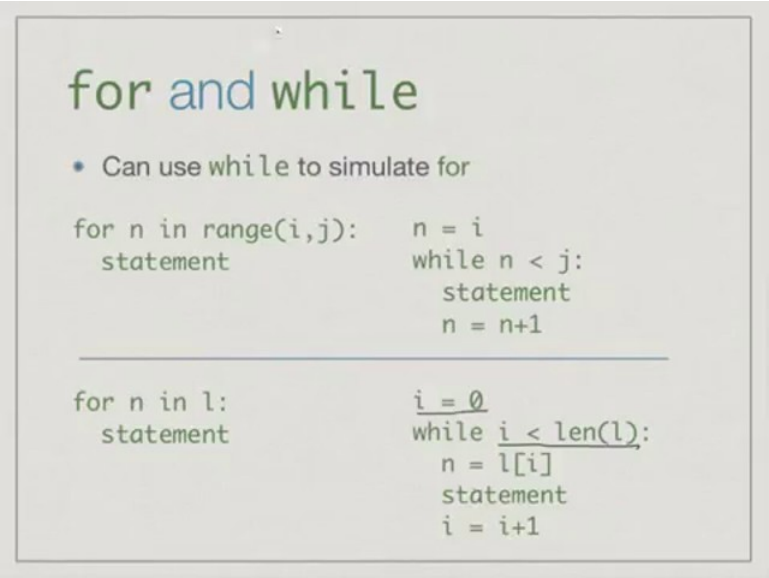
Every time we see a prime count is going to become count plus 1. We start with count equal to 0, so eventually it is going to cross n. Of course, we are using fact the implicit fact we know that there are an infinite number of primes. We were always for any n, we able to find the first n primes. So, 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)



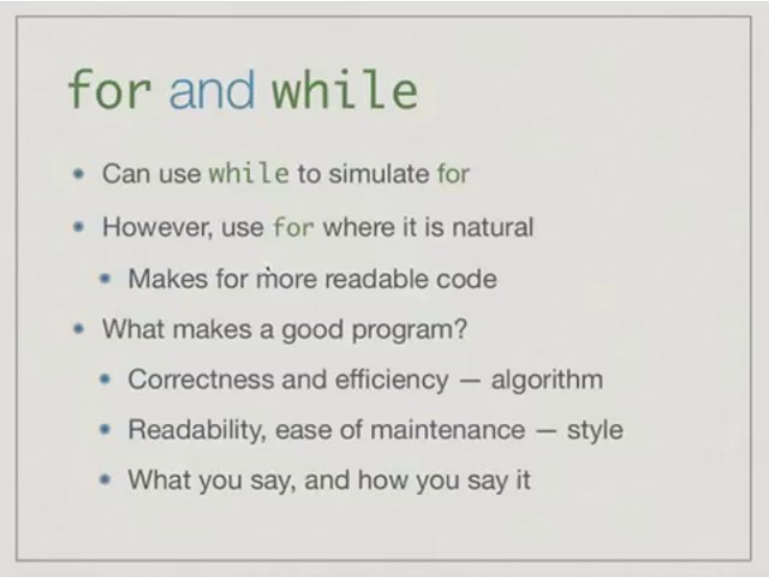
So, it turns out that you do not actually need a 'for', you can always simulate a 'for' by a while. Let us look at the two typical ways in which we write for. The first way is this for in a range, so we say for n in the range i to j. We start at i and we let n go through the sequence i, i plus 1 up to 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.

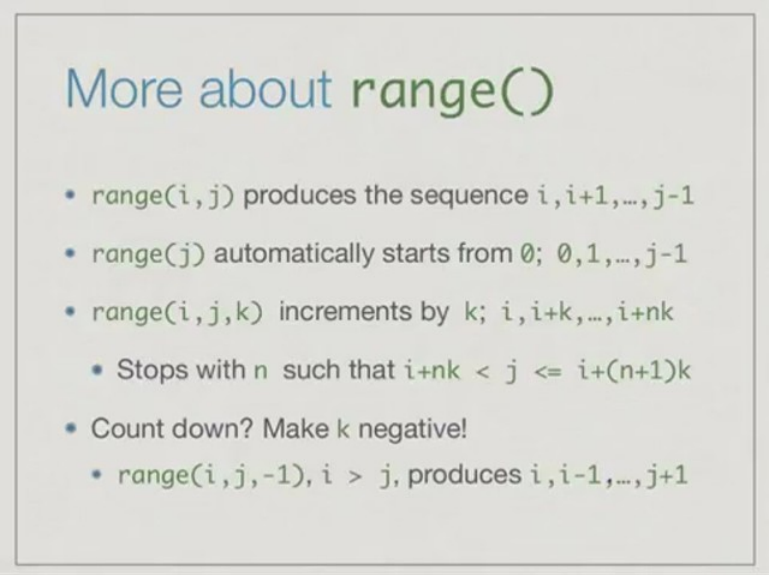
To summarize, there are two parts of programming; first is what you want to say which the algorithm is, in the second part is how you say which is the style. So, every programming language has a style use the style to make for the most effective and readable code you can find; if you have a 'for' - use 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.

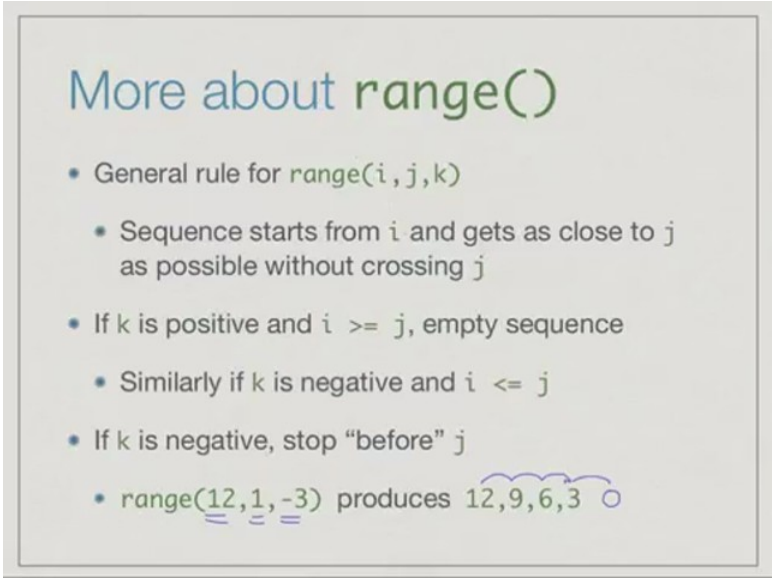
Often we may want to generate a sequence where we skip by a value other than 1. We want a kind of arithmetic progression if you are familiar with that. So, we want i, i plus k, i plus 2 k and so on, we do this by giving a third argument. The third argument if we

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)

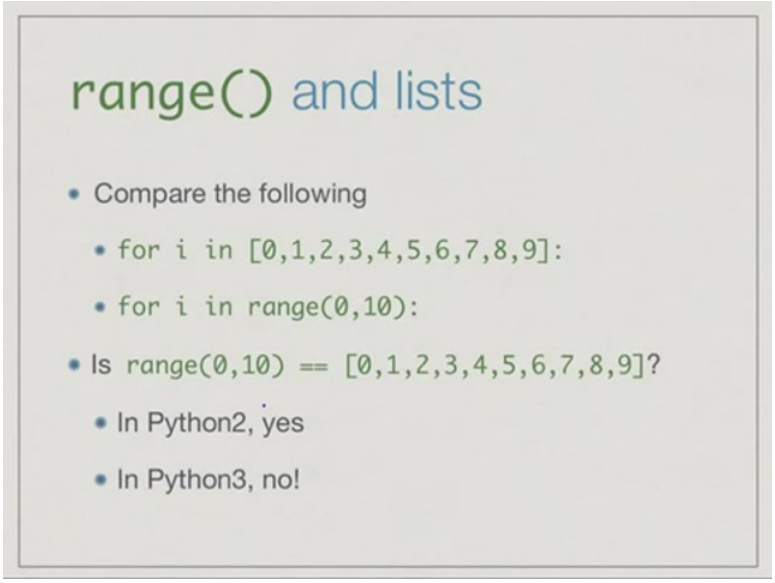
It is often confusing to new programmers that range i comma j is actually from i to j minus 1 and not to j itself. So, there is no real reason why it should be this way, it is just

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.

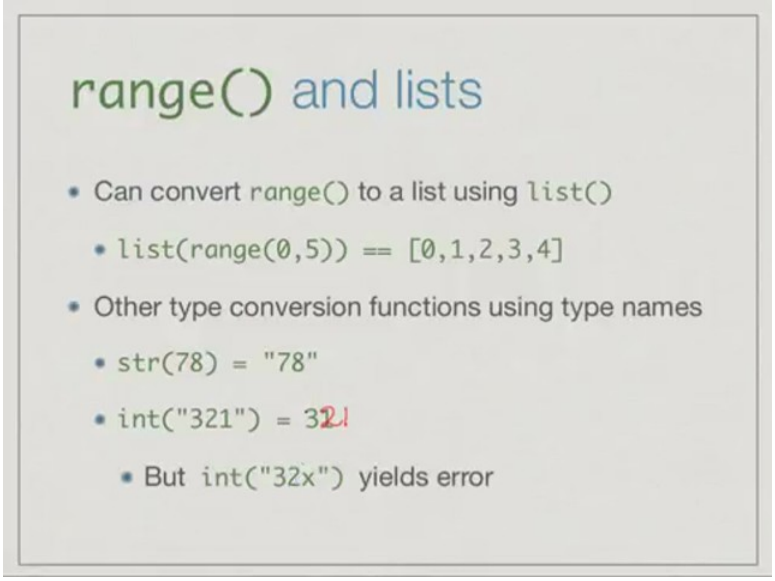
(Refer Slide Time: 05:23)



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

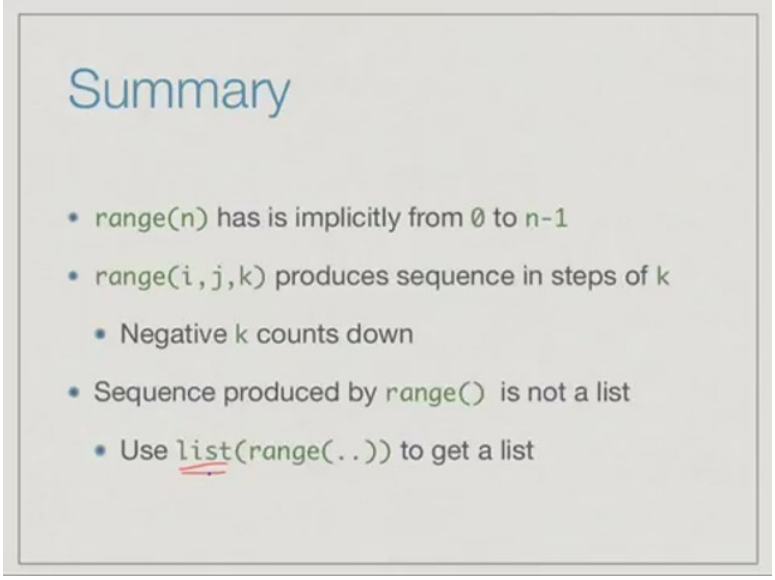
because 5 is the upper bound and this sequence will be converted to a list by the function list.

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

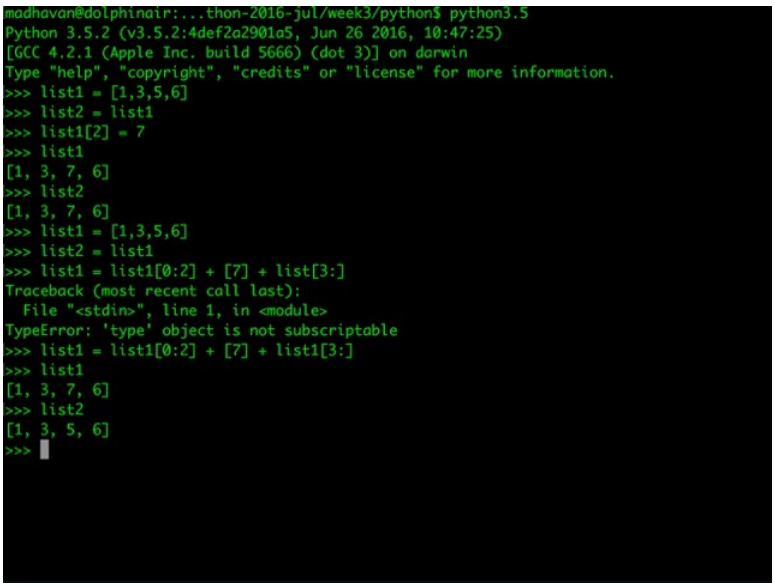
Week - 03
Lecture - 02
Manipulating lists

(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.

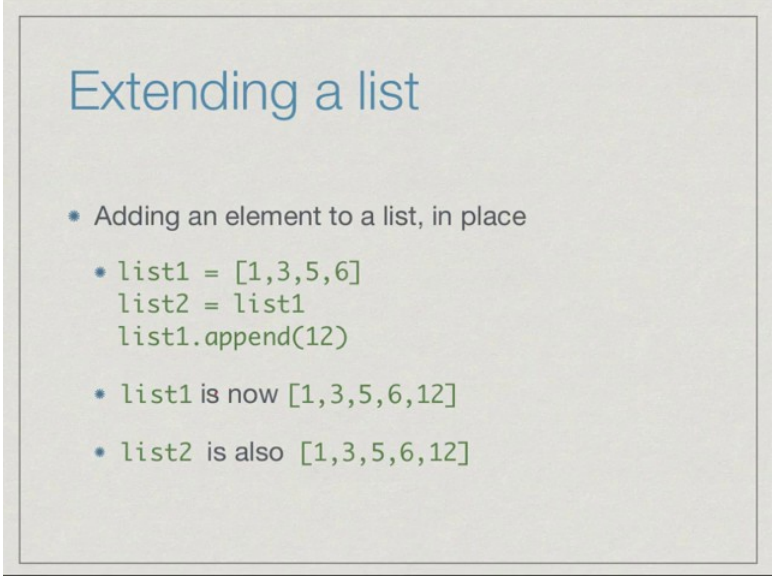
(Refer Slide Time: 01:38)



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)



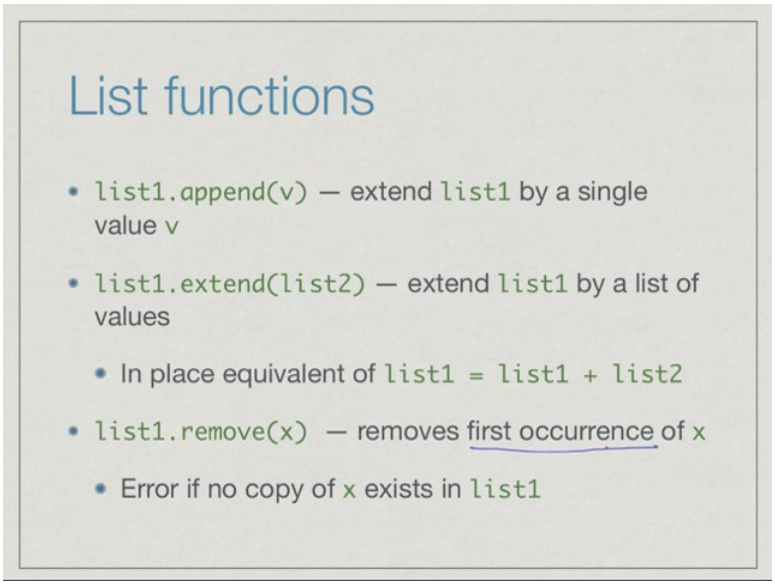
Now, how would be go about extending a list? Suppose, we want to stick a new value 22 at the end of a list; one way to do this is to say l is l plus 22. But as we saw, this plus operator will create a new list, so if you wanted to append a value to a list and maintain the same list so that for instance 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)

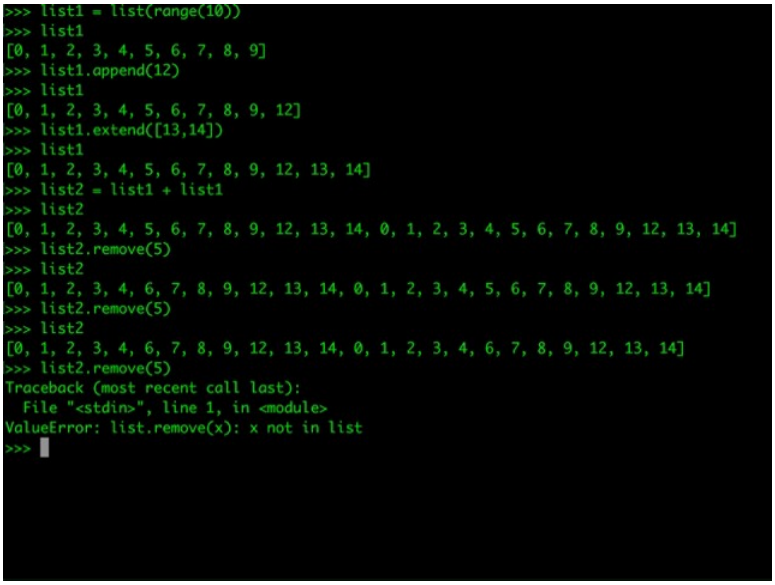


So, append takes a single value. Now, what if we wanted to append not a single value, but a list of values; we wanted to actually take a list and expand it by adding a list at the end, we had say 1, 3, 5 and we wanted to put 6, 8, 10. So, we want to take 1, 3, 5 and we wanted to expand this to have three 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)



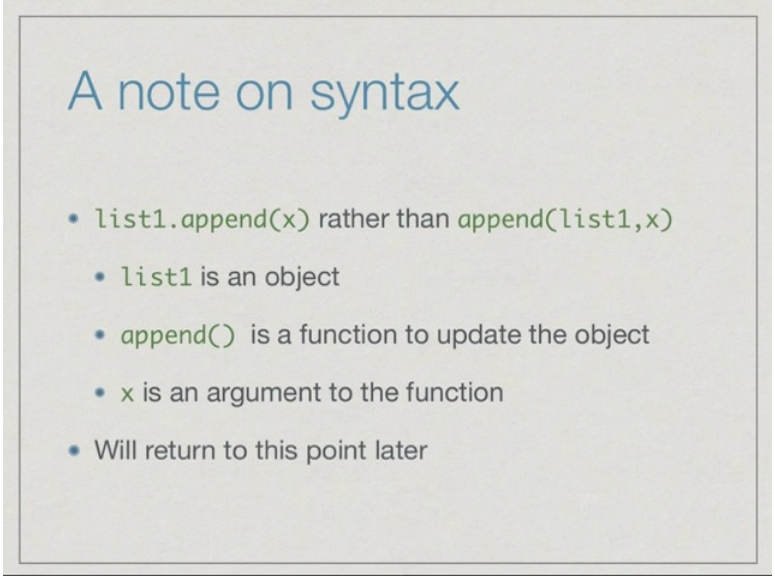
Let us explore these things. Let us start say with list1, so remember from the previous lecture we said we can take range and produce a list. Now if I do this I have list1 is 0, 1, 2 to 9, now if I say list1 dot append 12, then list1 is appended with 12. Now if I say list1 dot extend say 13, 14, then 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.

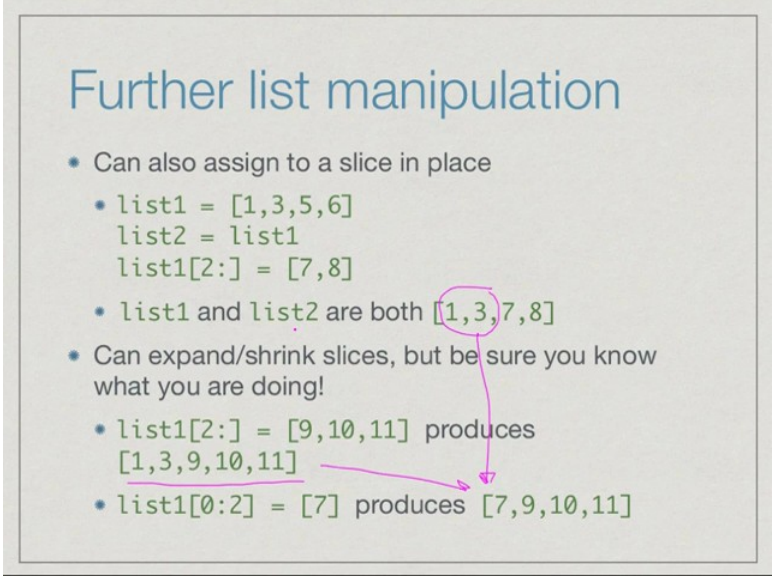
(Refer Slide Time: 08:18)



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.

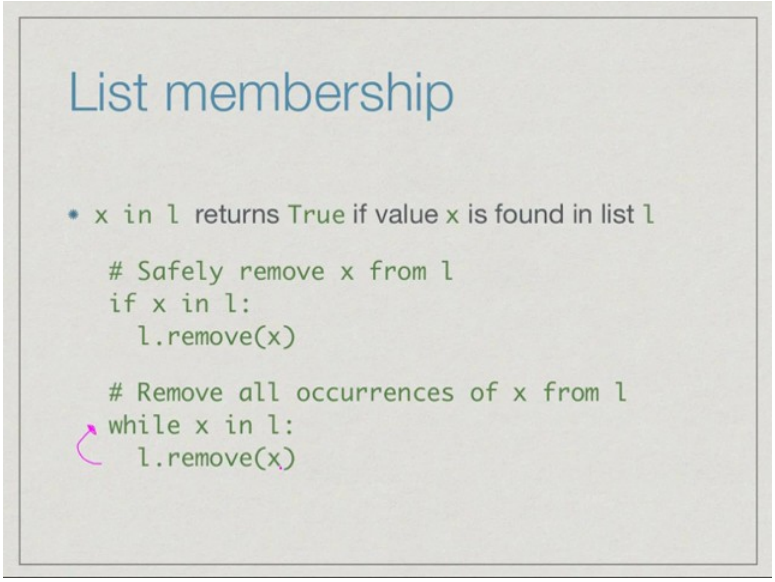
Now, here we had a slice of length two and we replaced it by a new list of length two. So, we preserved the structure of the list in terms of the number of positions. This is not required, Python allows you to both expand and shrink a slice. For instance, you could have taken that list, now let us say 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)

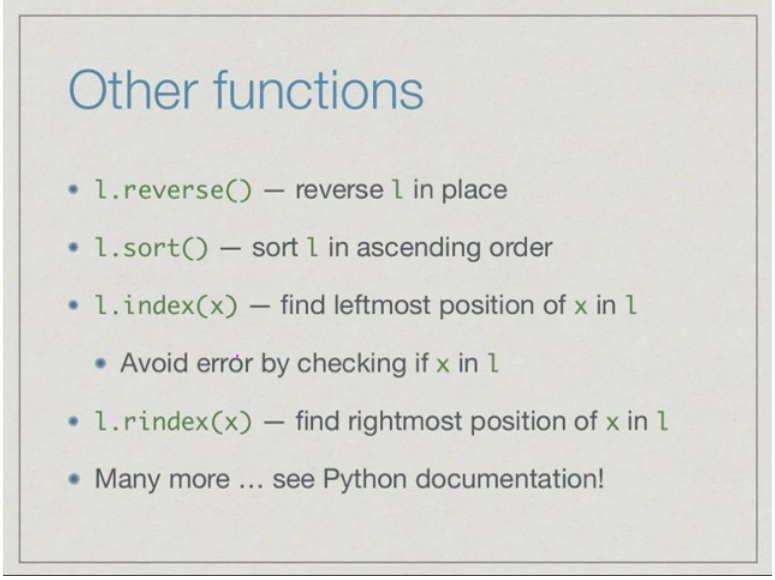


One of the very common things that we want to know about a list is whether a value exists in a list. So, Python has a very simple expression called x in l. So, x in l returns true if the value x is found in the list l. Now we can use this for instance to make our remove a safe operation; before we invoke 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)



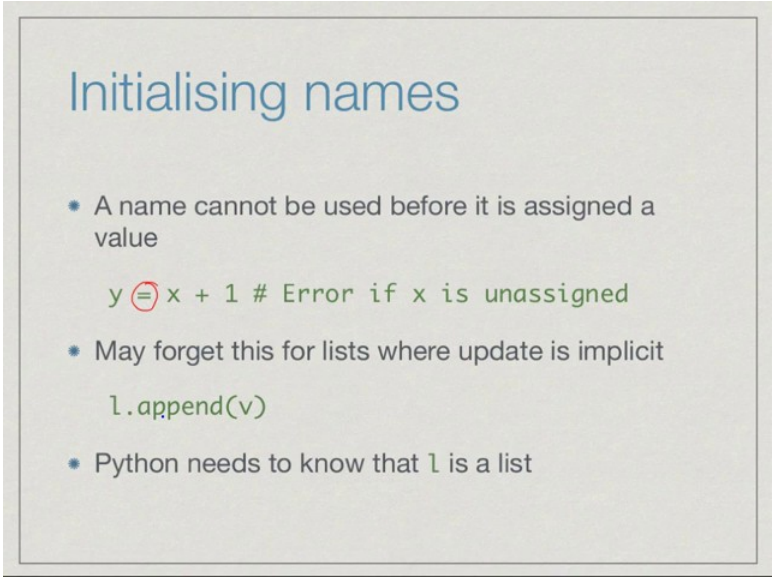
Now there are a host of other functions defined for list, for instance l dot reverse will reverse a list in place, l dot sort will sort a list in ascending order. You can also sort it in other orders you can look up and see how to do that. If we only want to know whether an element is in l we say x in 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)



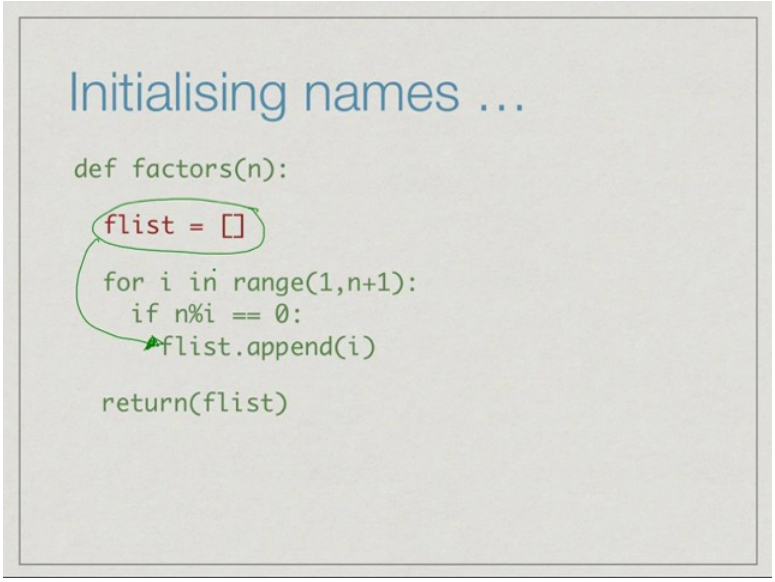
Final point regarding list is something we talked about in passing, which is that since names do not have types in Python, we do not have to announce the name, names just pop up as the code progresses. So every time a name pops us Python needs to know what value it is. Typically the first time we use 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)



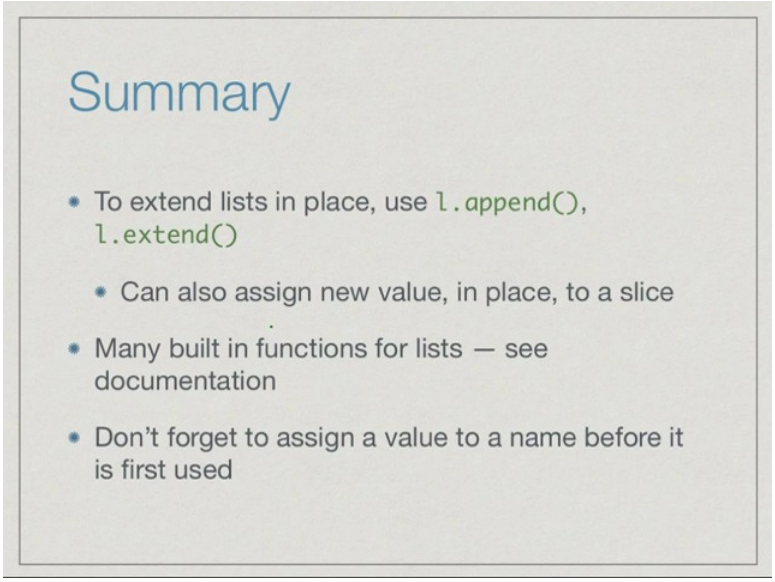
So, we saw this small function earlier which computes factors of n. So essentially what it does is, it takes all numbers in the range 1 to n. I take 1 to n plus 1 so that I run through the sequence 1 to n. And if a number divides n evenly if there is no remainder I have used the append function now to 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

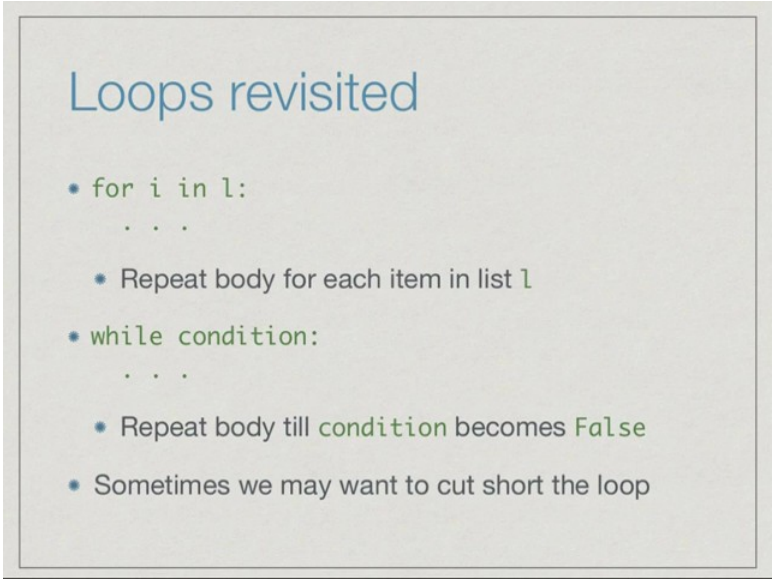
list; we will see some of them as we go along and use them and describe them as we see them, but it is impossible to document all of them and to go through all of them and it is also a very boring to just list out of a bunch of functions.

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

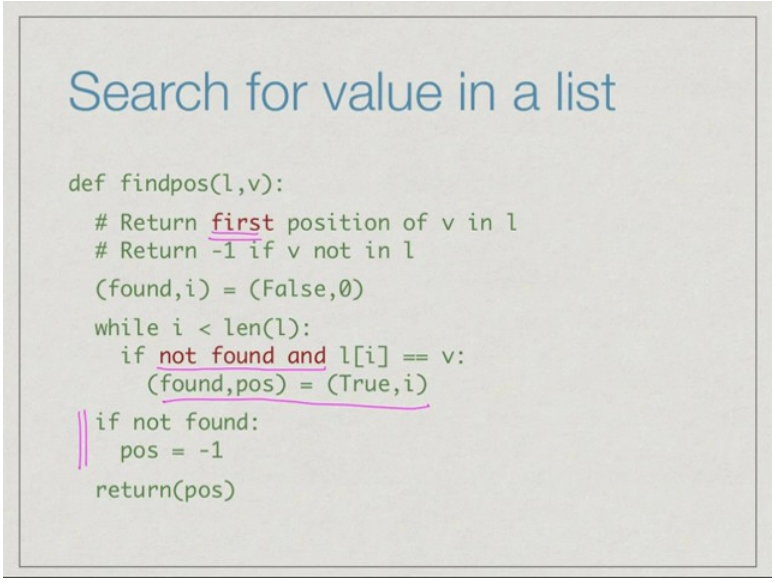
(Refer Slide Time: 00:01)



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.

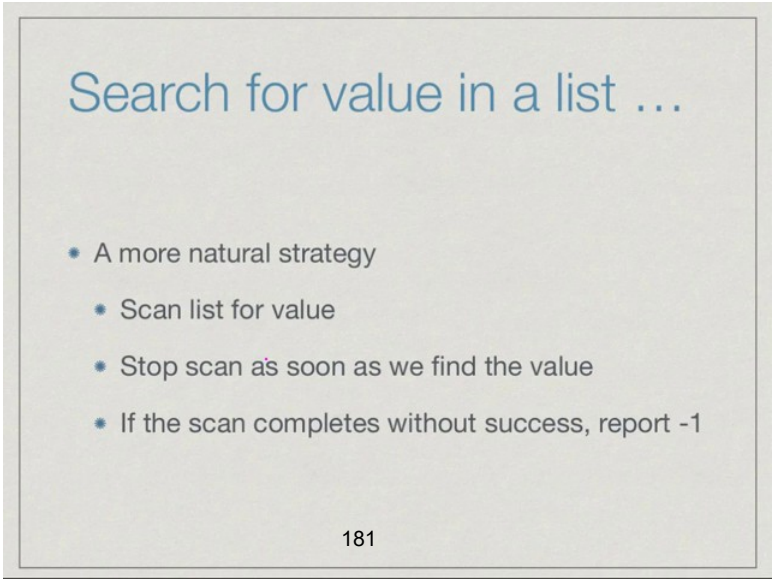
There are two points to observe about this; the first point which is the main issue at hand is that we are going to necessarily go through every position i from 0 to the length of l minus 1 regardless of where we find it. Supposing, we had several hundreds of thousands of elements in our list and we 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.

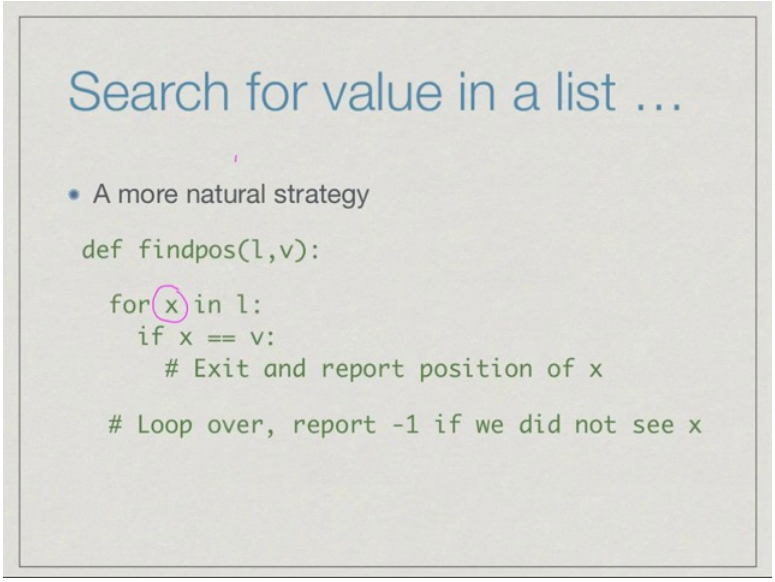
This ensures that pos is updated to i only the very first time we see v after that the value found prevents us from doing this update again. So, at least we are correctly finding the first value of the position. And finally as before if we never find it then the value found is never set to true and so 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)



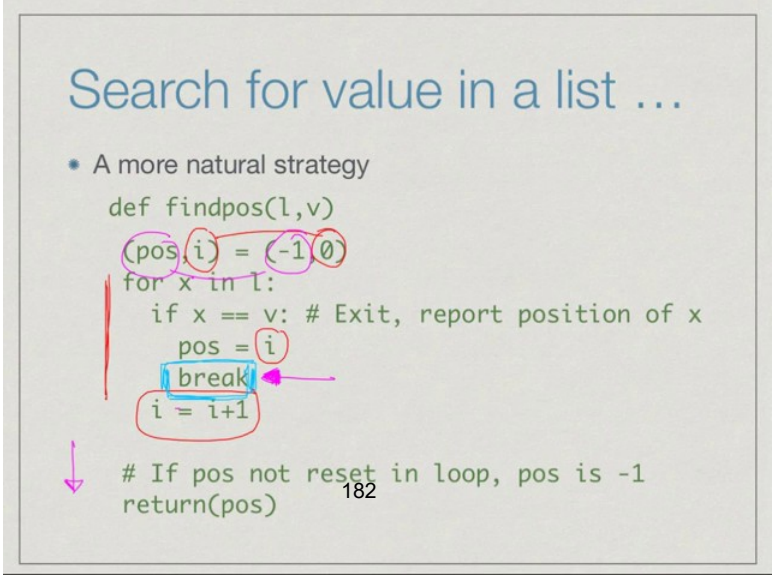
So, a more natural strategy suggests the following; we scan the list and we stop the scan as soon as we find the value. So, whenever we find the value for the first time we report that position and we get out. Of course, the scan may not find the value, so if the scan completes without succeeding then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



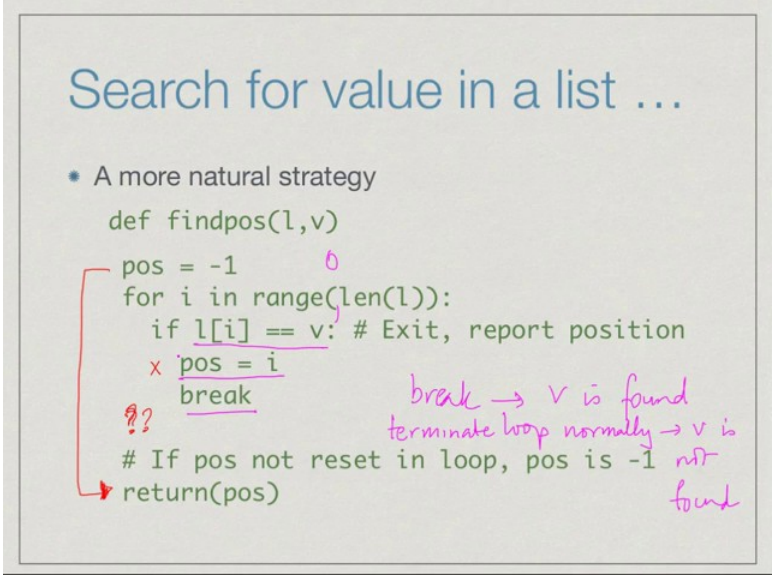
Here is a first attempt at doing it, so let us go through it systematically. First of all we have this loop, so for each x in the list if x is v then we report the position. Now we have to find the position because we are going through the values so this has forced us to use a name i to record the position 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)

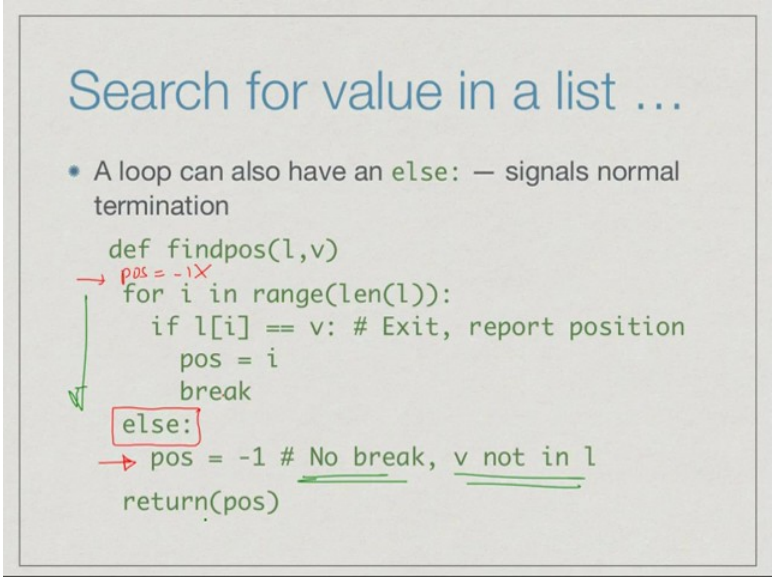


We can simplify this a little bit by first removing this i instead of scanning x actually it is better to scan the positions. So, it is better to say pos is minus 1, but instead of going through x in l it is better to go through i in the range 0. Remember now this is implicitly 0 to the length of l minus 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)



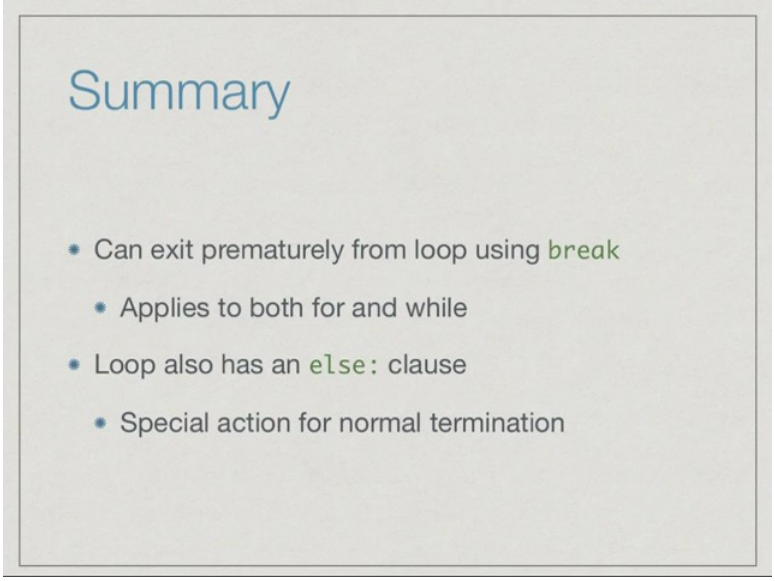
Python is one of the few languages which actually provide way to do this. So it allows something that looks a bit odd because of the name it allows something called else which we saw with if, it allows an else for a loop as well. The idea is that else this part will be executed if there is no break if 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)

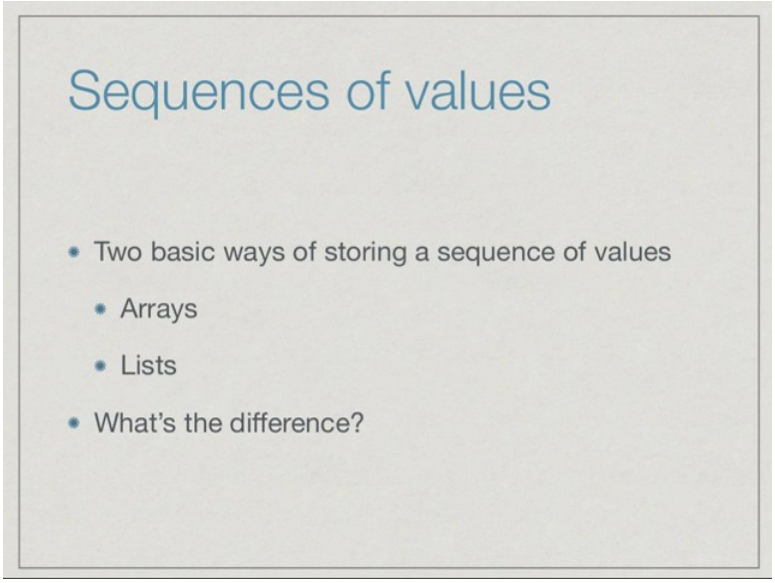


To summarize, it is very often useful to be able to break out of a loop prematurely. We can do this for both for and while we can execute the break statement and get out of a loop before its natural termination. And sometimes it is important to know why we terminate it, we terminate it because the loop 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

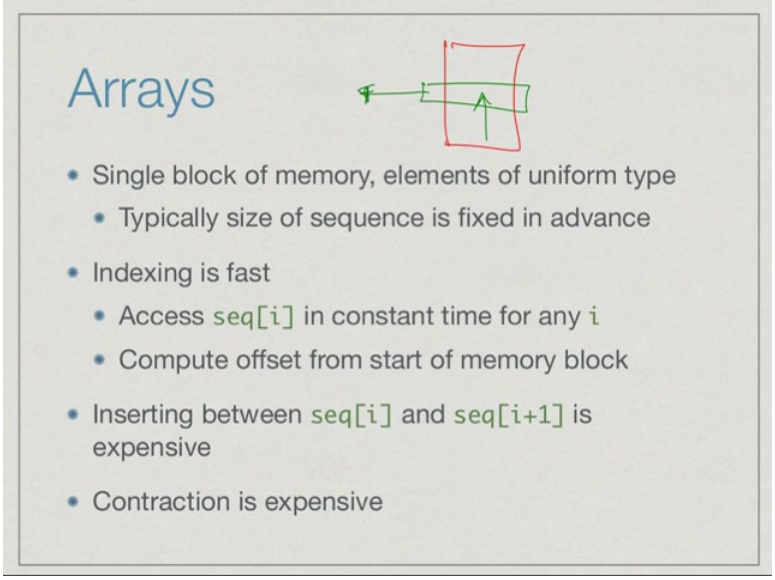
Week – 03
Lecture – 04
Arrays vs. Lists, Binary search

(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.

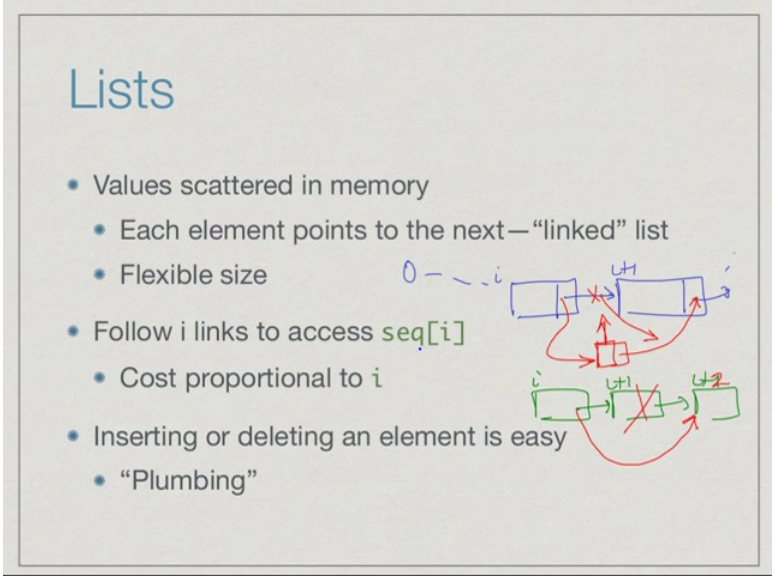
Now when this happens, what happens is that if you want to look at the jth element of a sequence or the ith element of a sequence, then what you want to think of is this block of memory starting with 1, 2, 3, up to i right and you want to get to the ith element quickly. But since everything is of a uniform 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.

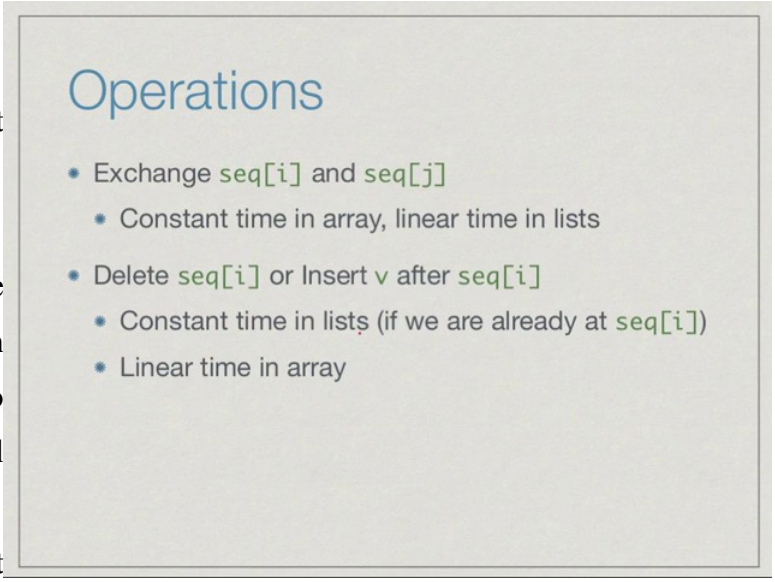
Since we have to follow these links the only way to find out where the ith element is is to start from the 0th element and then go to the first element then go to the second element and so on, because a priori we have no idea where the ith element is. So, after i steps we will reach the ith element. 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)



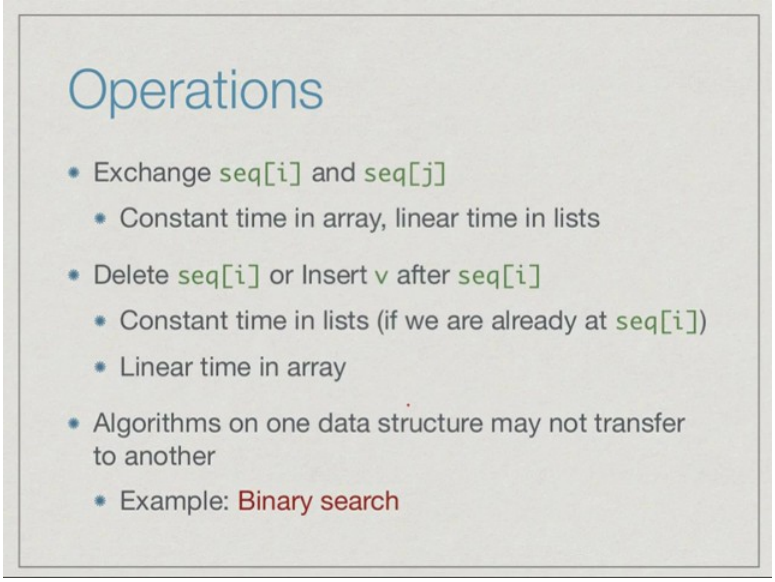
Let us look at typical Operations that we perform on sequences. So one typical operation, now if i just represent a sequence more abstractly as sequences we have been drawing it. Supposing, I want to exchange the values at i and j. This would take constant time in an array because we know that we can 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.

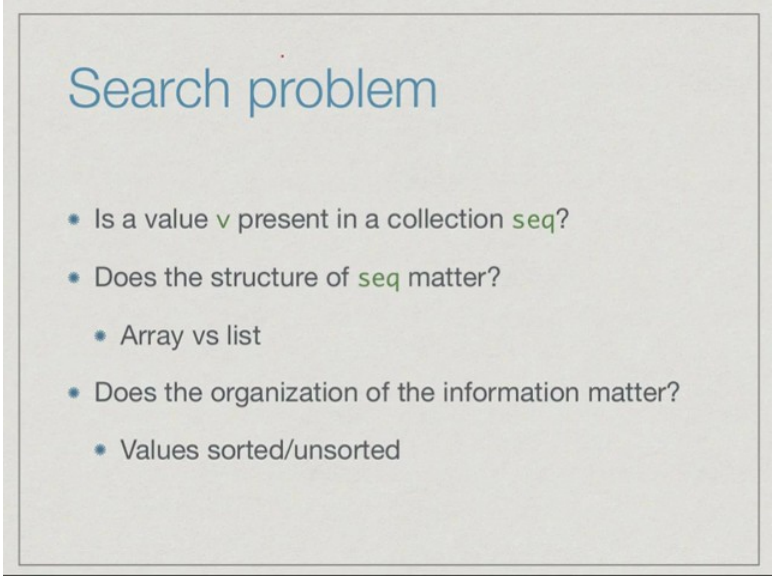
On the other hand as we have already seen, if you want to delete the value at position i or insert the value after position i this we can do efficiently in a list because we just have to shift some links around, whereas in an array we have to do some shifting of a large bunch of values before or after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

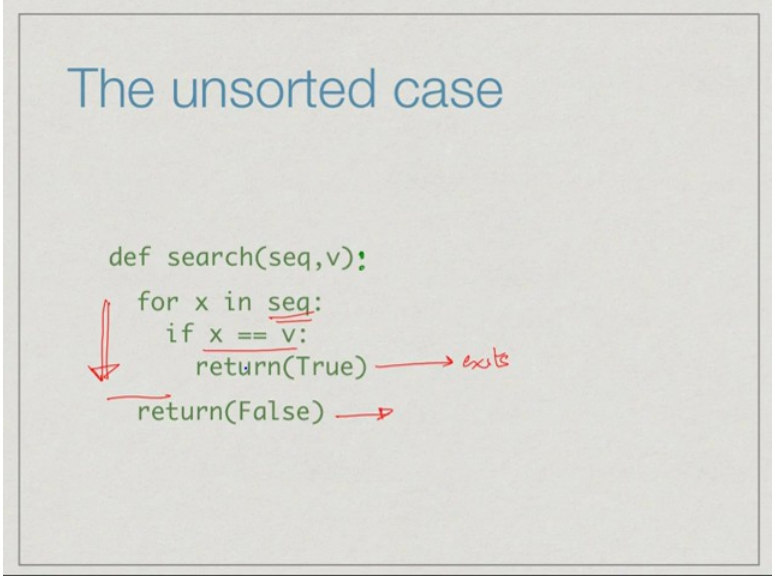
(Refer Slide Time: 09:42)



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)



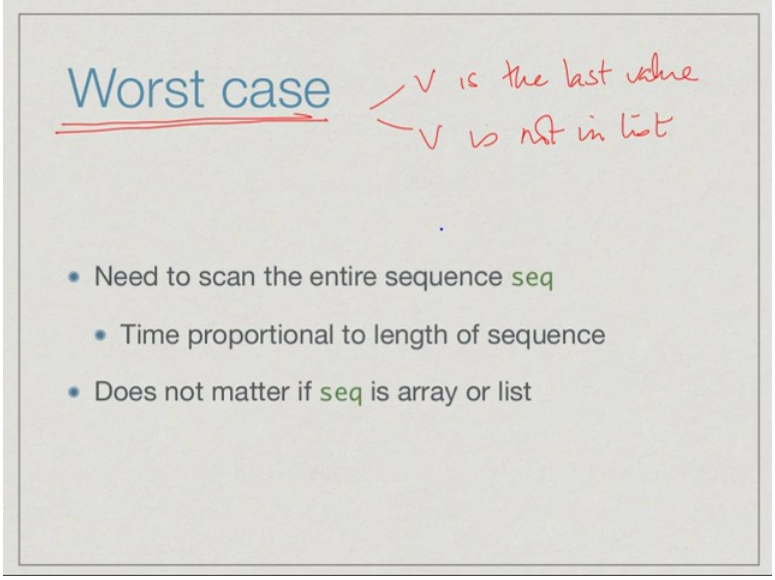
Here is a very simple Python program to search for a value in a unsorted sequence. This is a similar to what we saw before where we are looking for the position of the first position of a value in a sequence, which is we not we do not even need the position we only need true or false, is it there or 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)



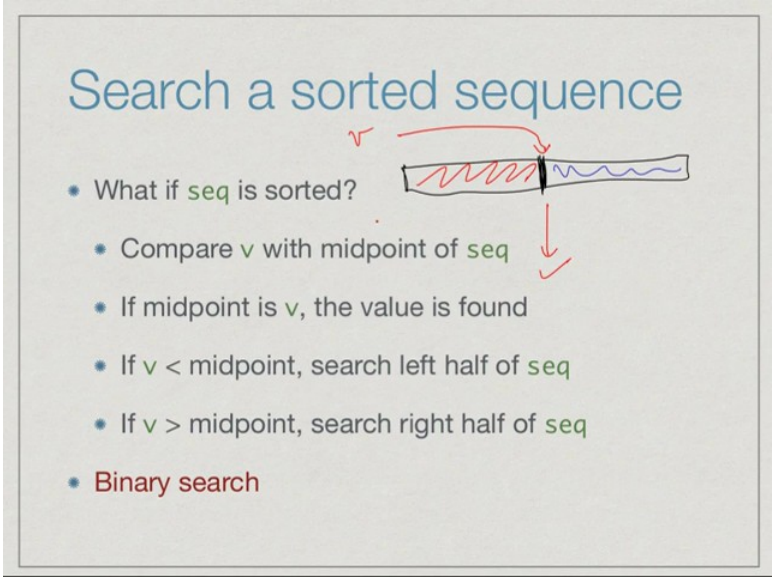
The main point of this function is that we have no solution to search other than to scan from beginning to end. The only systematic way to find out v occurs in the sequence or not is to start at the beginning and go till the end and check every value, because we do not have any idea where this value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

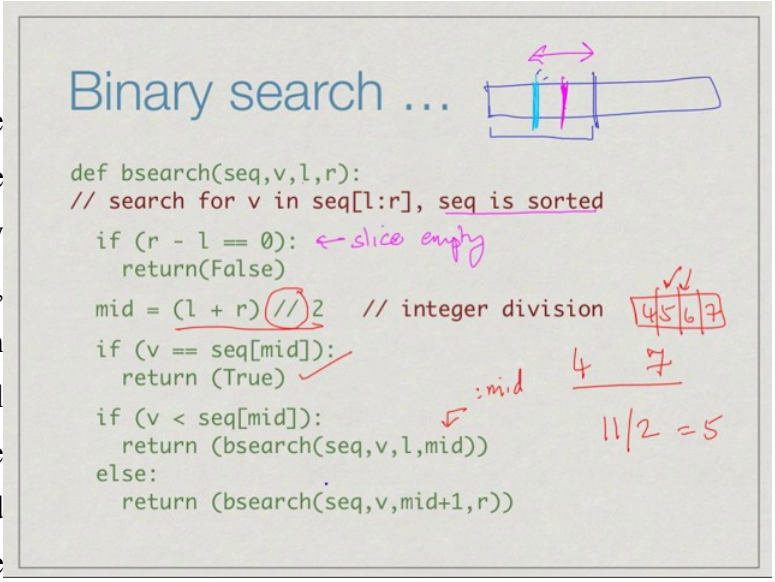
In general if we have a sequence that efficient way to search for this value is to first look at the middle value, so we are looking for v, so we check what happens here. So, there are three cases either we have found it in which ways which case we are good, if we have not found it we compare the value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

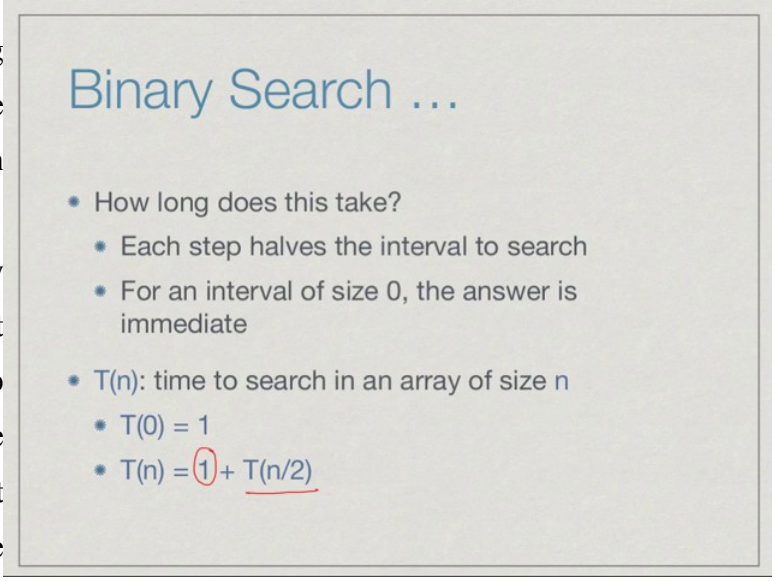
In general binary search is trying to do a binary search for a value in some segment of the list. So we will demarcate that segment using l and r. So, we are looking for this slice sequence starting with l and going to r minus 1, we are assuming that sequence is sorted and we are looking for v there. 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.

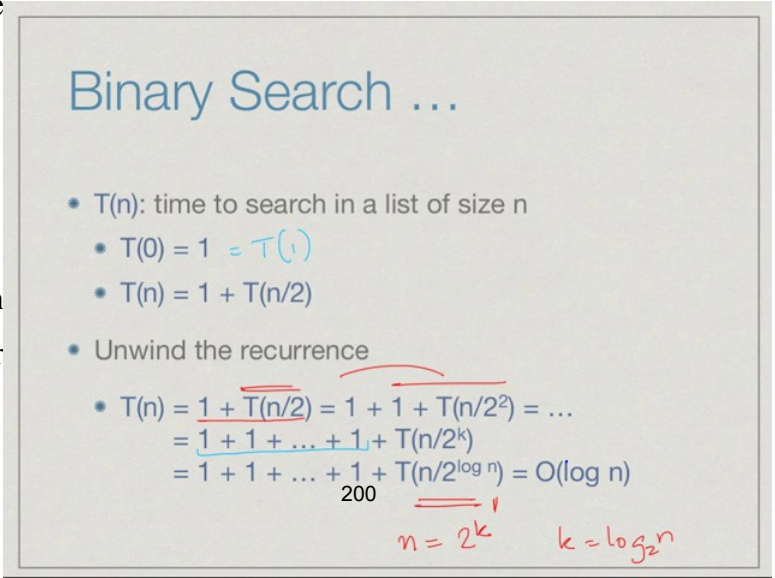
(Refer Slide Time: 18:08)



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)



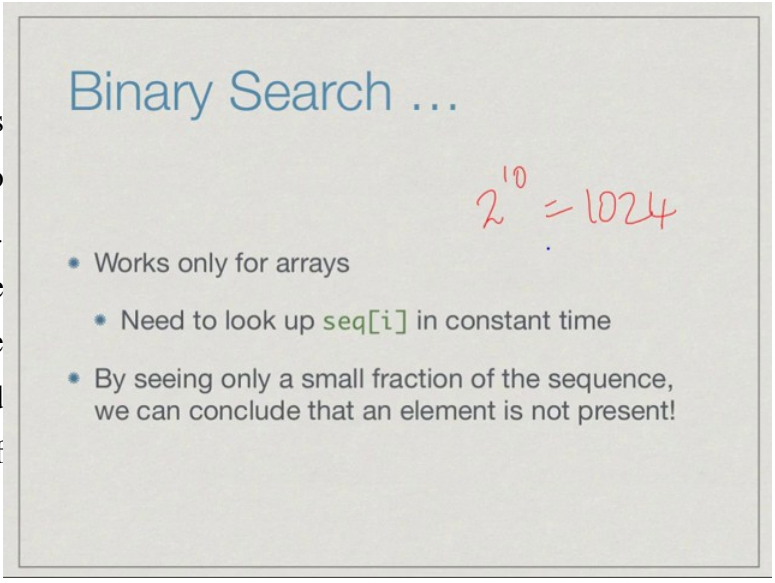
We want an expression for

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)



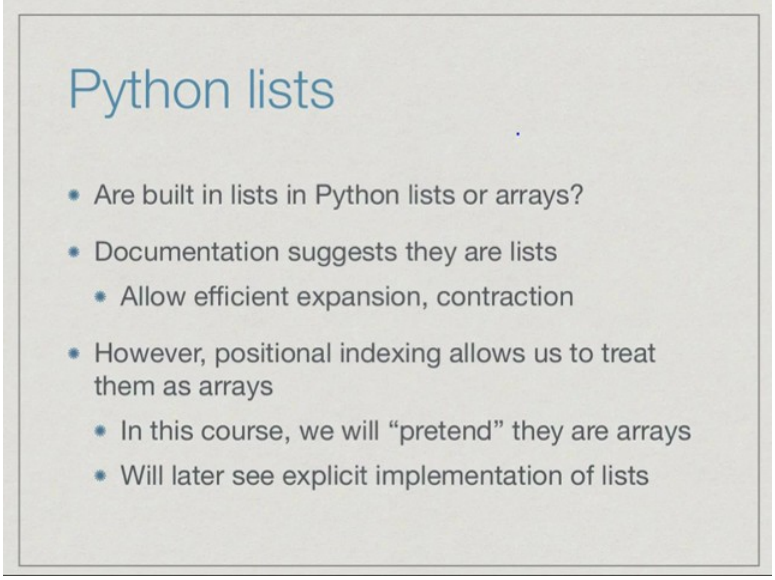
This comes back to another point. Now we have said that if we had a sorted sequence of

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.

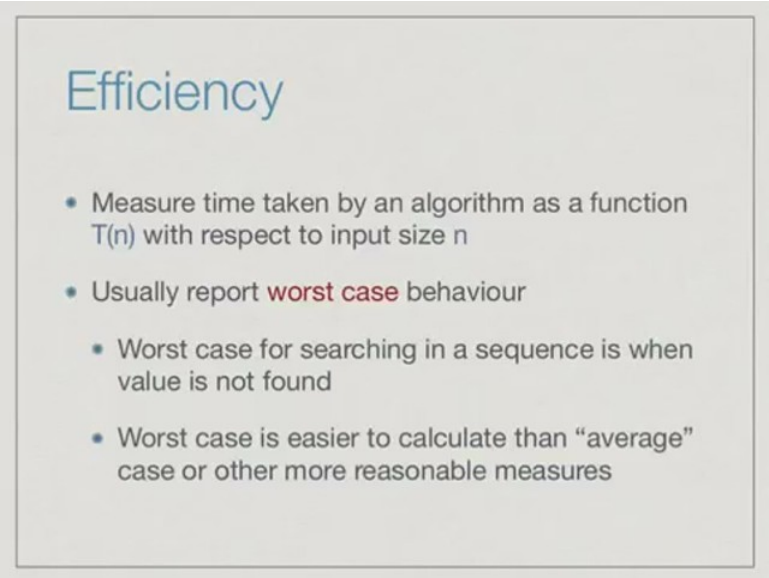
Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

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)



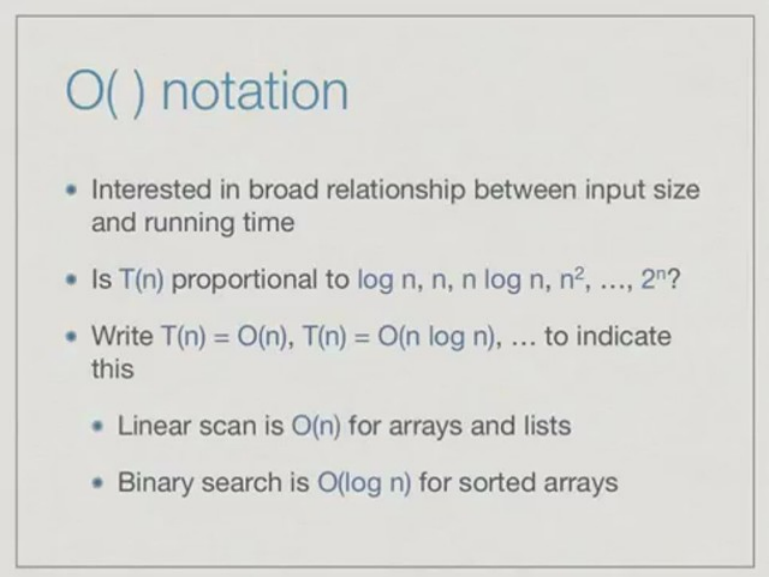
In general an algorithm will work on many different sizes of inputs, so it makes sense to talk about the efficiency as a function of the input size. The input size is n we will use a function such as T of n to talk about that time taken on an input of size n. Of course, even of the same size, different 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)

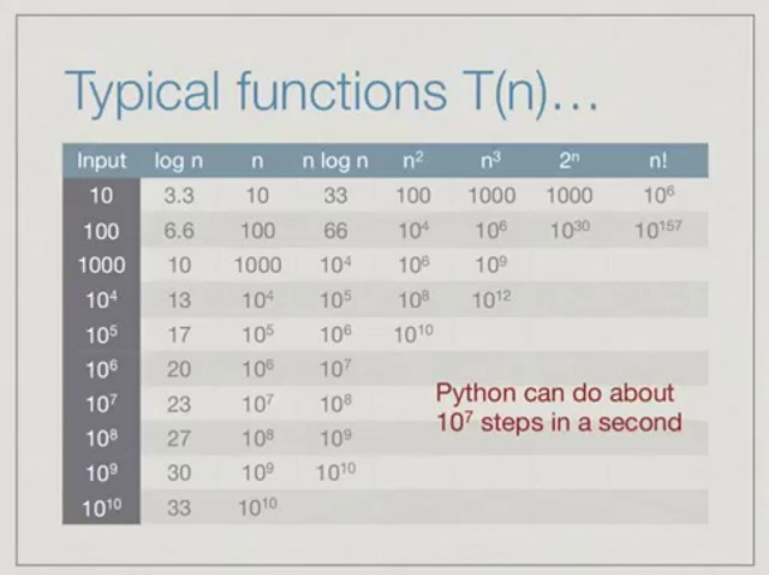


When we talk about efficiency, as we said we are broadly interested in the connection between input size and output size so we express this up to proportionality. So we are not really interested in exact constants we want to know for instance is T of n proportional to log of n, for example in the case 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.

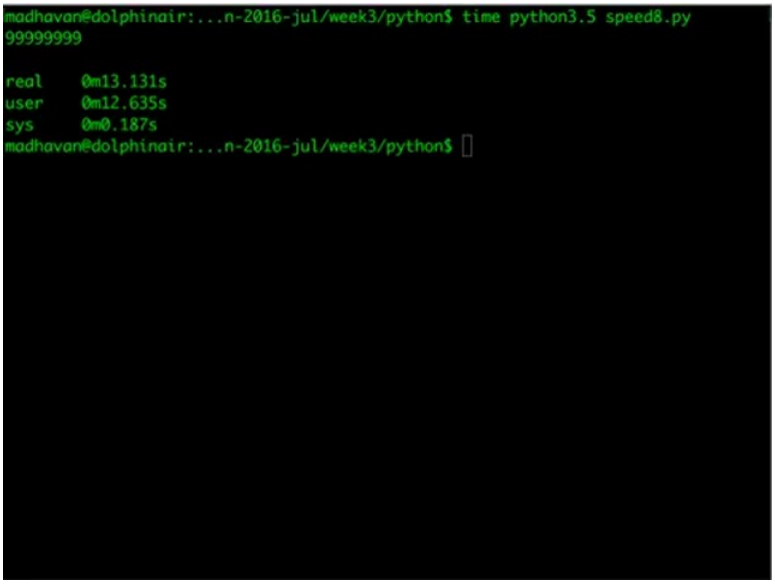
(Refer Slide Time: 03:24)



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.

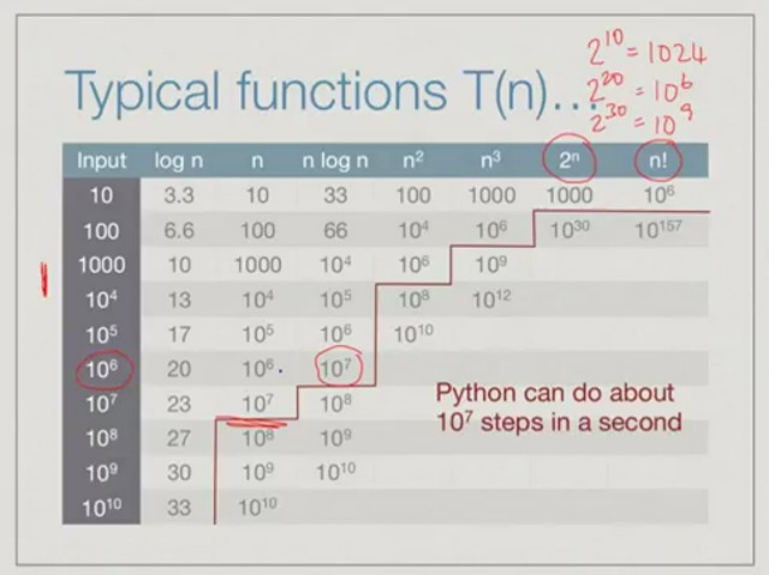
First of all I can take python and I can take directly use a name of the Python program like this. So, I can say Python 3.5 and give the name of this script and it will execute it and give you the answer. But now in addition there is also a useful command called time. So, time tells us how much time 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)



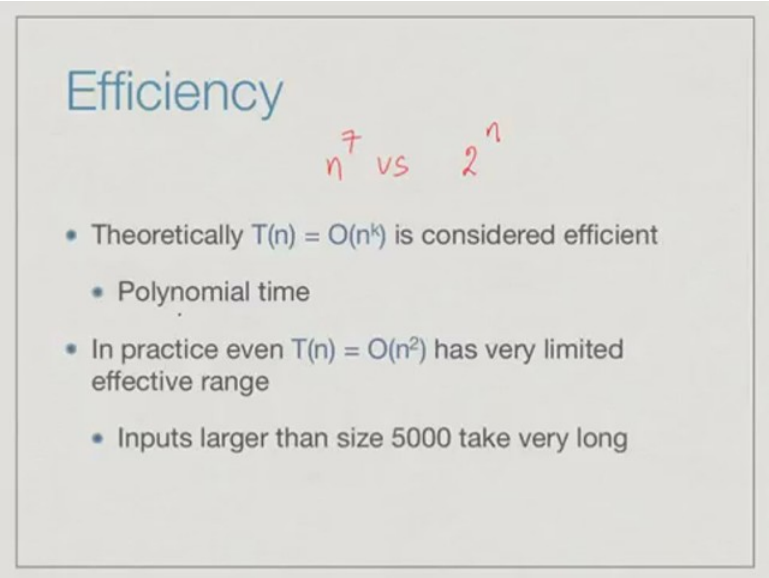
So, coming back to our table assuming that 10 to the 7 is the limit that we are looking at, let us see what happens when we mark of 10 to the 7 on these different columns. It turns out as something takes log n of time then even for 10 to the 10 it takes only 33 steps and we are fine. Of course, if input 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.

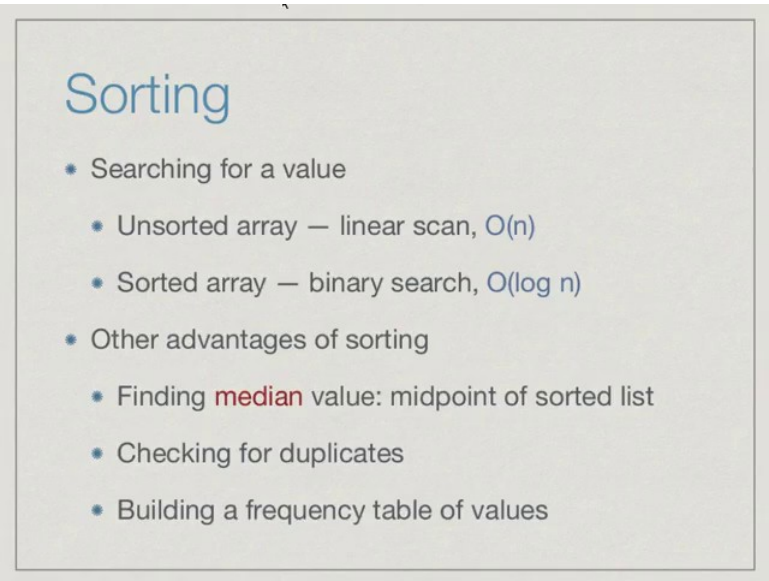
But what the table tells us if you look at the previous table, is that even n square has a very severe limit, we can only do about 4 to 5000. If you are doing something in n squared time we cannot process something larger than a few thousands. Now many of the things that we see in real life, like if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

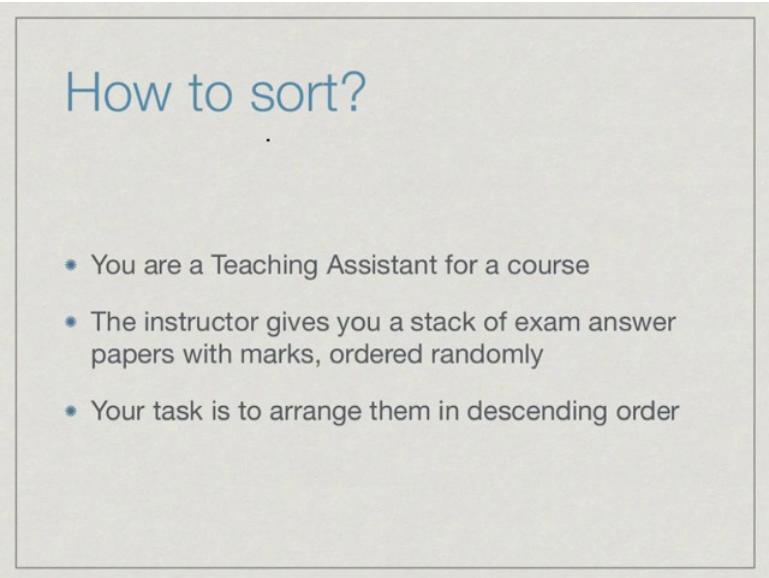
(Refer Slide Time: 00:02)



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.

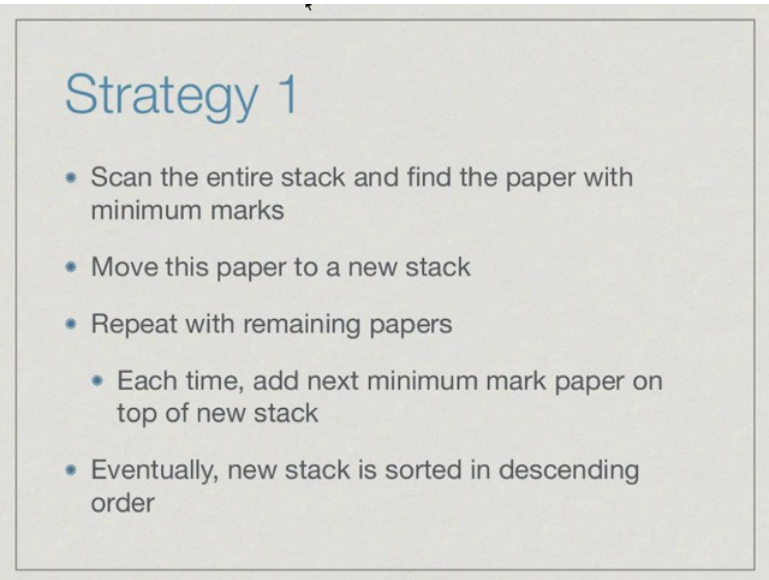
Once we have sorted a sequence, the midpoint automatically gives us the median. We can also do things like building frequency tables or checking for duplicates, essentially once we sort a sequence all identical values come together as a block. So, first of all by checking whether there is a block of 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.

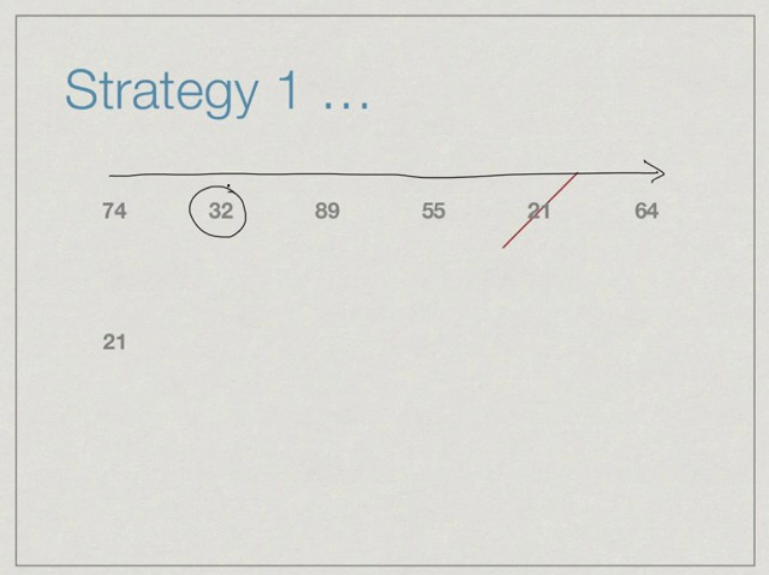
(Refer Slide Time: 01:46)



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.

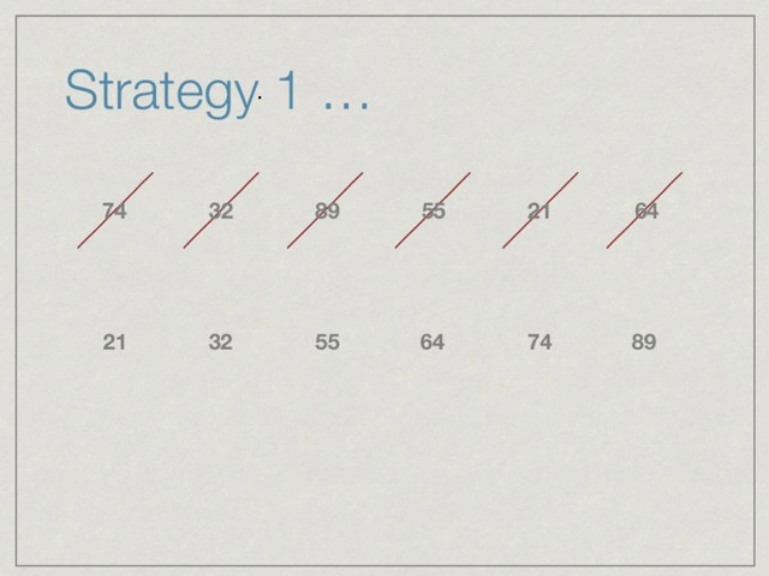
Initially, we assume that the top most paper has the minimum marks and we keep going down and replacing it with any lower mark we find. After this scan, we take the paper we have in our hand and put it aside and make a second stack where this is the bottom most thing. Now we have n minus 1 paper, we 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)



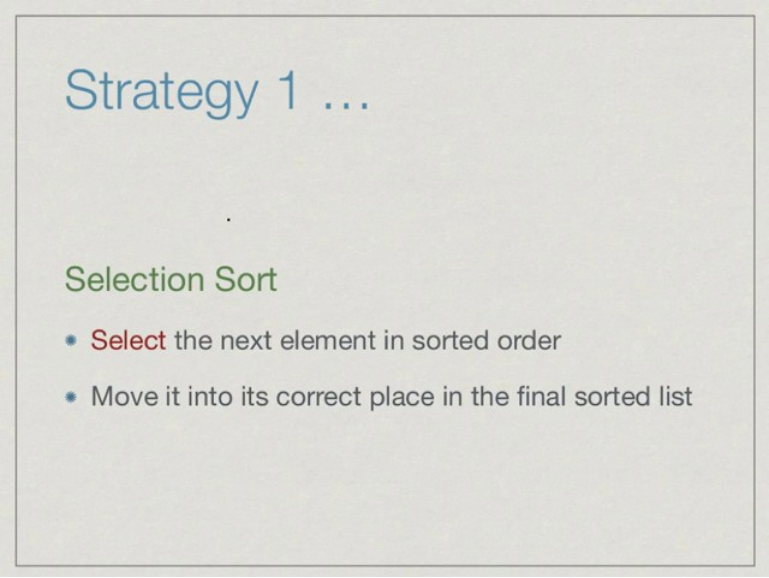
Suppose these are 6 papers. So, we have papers with mark 74, 32, 89, 55, 21 and 64. If we scan this list from left to right, then we will find that 21 is the lowest mark. So, our strategy says pick up the one with the lowest mark and move it to a new sequence or a new stack, so we do that.

(Refer Slide Time: 03:38)



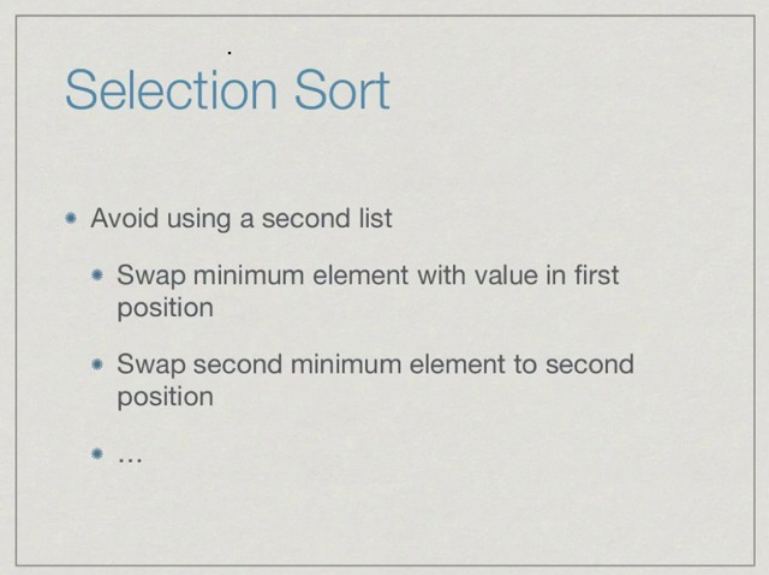
Now again, we scan from left to right this time of course 21 is gone, so we only have five numbers to scan. We will find that 32 is our next. And then proceeding in this way at the next step we will pick up 55 and then 64 and then 74, and finally 89. In this way by doing six scans on our list of six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

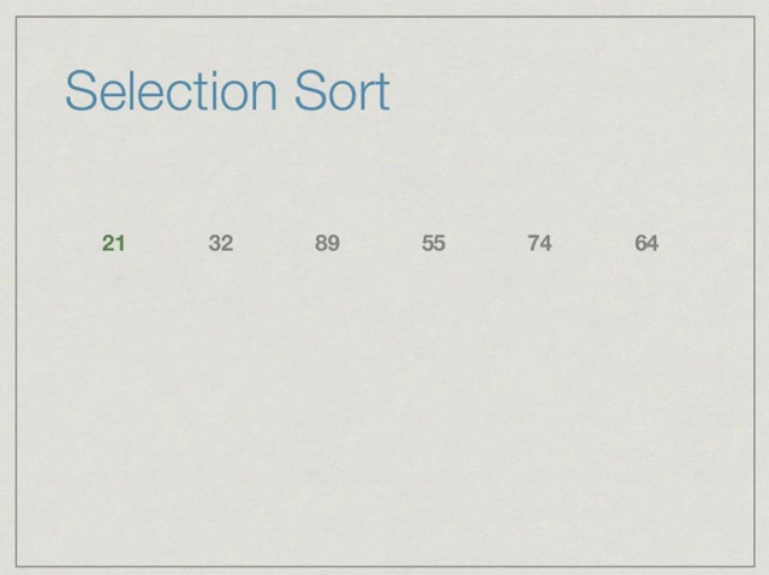


In the algorithm that we executed just now, we needed to build up a second list or a second sequence to store the sorted values. So, we kept pulling out things from the first sequence, and putting it in the second sequence. However, a little bit of thought will tell us that we do not need to do this. 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.

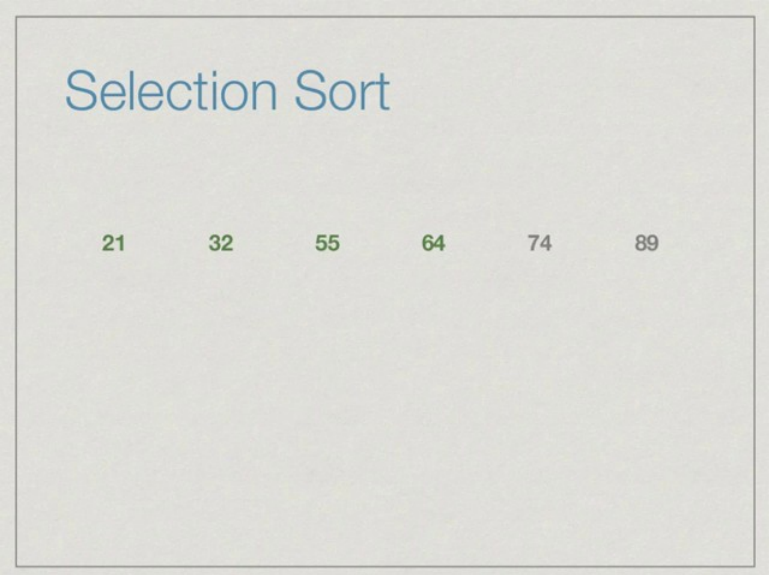
(Refer Slide Time: 05:09)



So, 21 comes in the beginning and 74 goes to the position where 21 was. Now we no

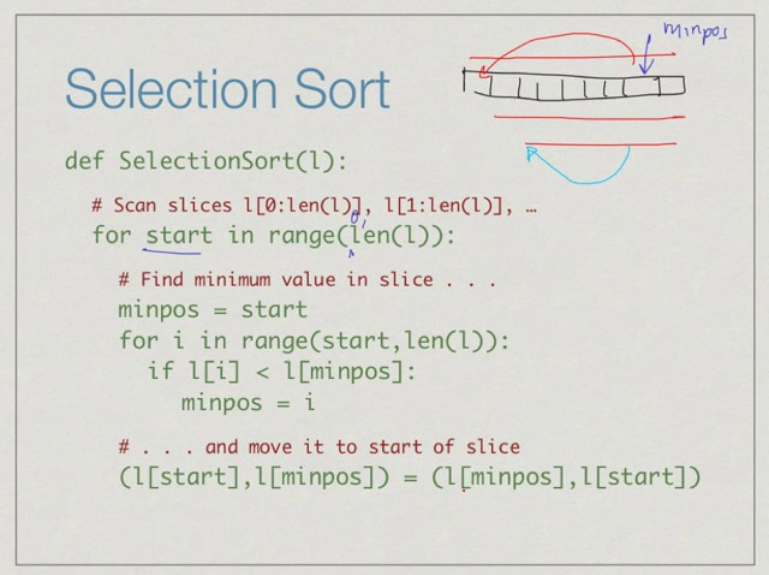
longer have to worry about anything to do with 21, we only need to look at this slice if you want to call it that starting from 32. We do this and we find the second smallest element. Now, the starting element is 32 and the second smallest element also happens to be 32 that is the smallest element in 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)



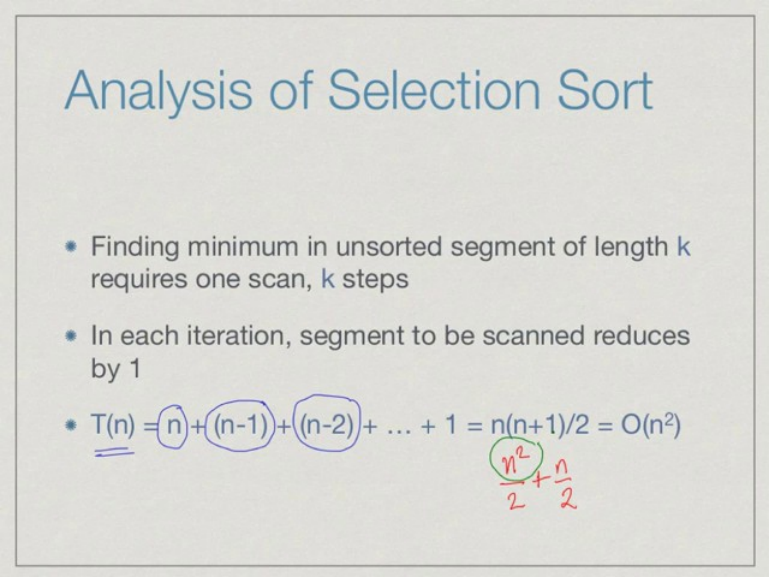
Here is the very simple Python function which implements selection sort. The main idea about selection sort is that we have this sequence which has n elements to begin with. The first time, we will scan the entire sequence, and we would move this smallest element to this point. Then we will scan the 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)

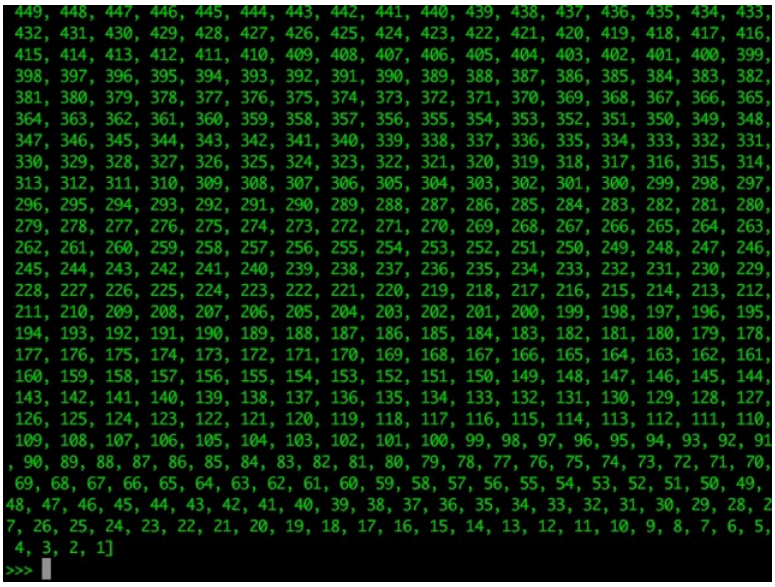
```
madhavan@dolphinair:...eek3/python/selectionsort$ more selectionsort.py
def SelectionSort(l):
  # Scan slices l[0:len(l)], l[1:len(l)], ...
  for start in range(len(l)):
    # Find minimum value in slice . . .  minpos = start
    for i in range(start,len(l)):
        minpos = start
        if l[i] < l[minpos]:
            minpos = i
            # .. and move it to start of slice
            (l[start],l[minpos]) = (l[minpos],l[start])
madhavan@dolphinair:...eek3/python/selectionsort$ python3.5
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from selectionsort import *
>>> l = [3,7,2]
>>> SelectionSort(l)
>>> l
[2, 3, 7]
>>> l = list(range(500,0.-1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
>>> l = list(range(500,0,-1))
>>> ls
```

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

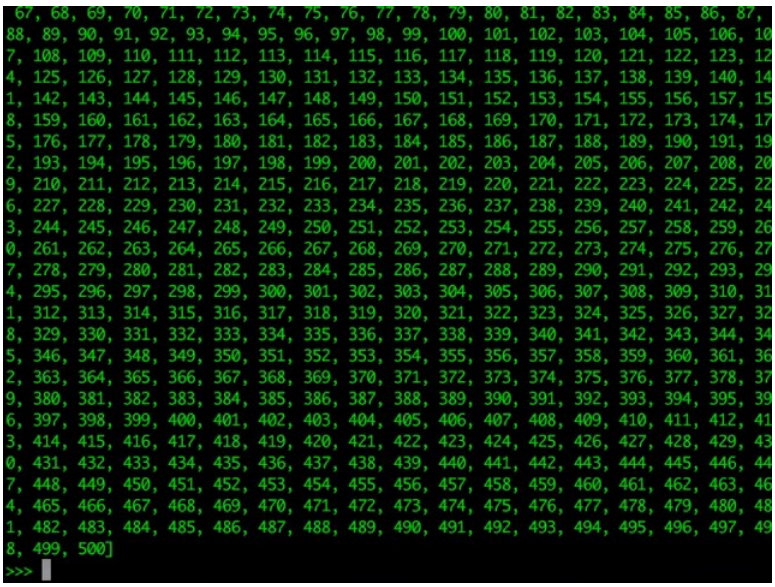
longer list. For instance, we can use this range function and say give me the list which is created by taking the range say from 500 to 0 with step of minus 1. So, this is an descending list of length 500.

(Refer Slide Time: 11:28)



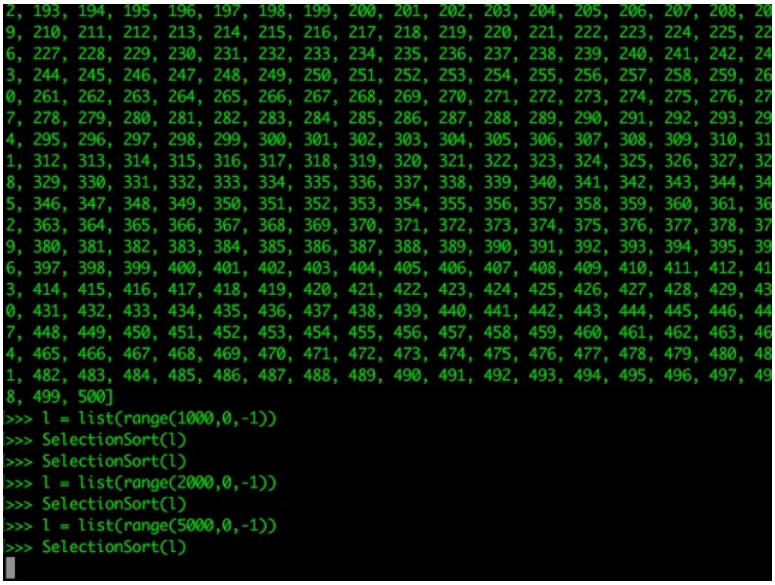
If I look at l, it is 500 down to 1.

(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)



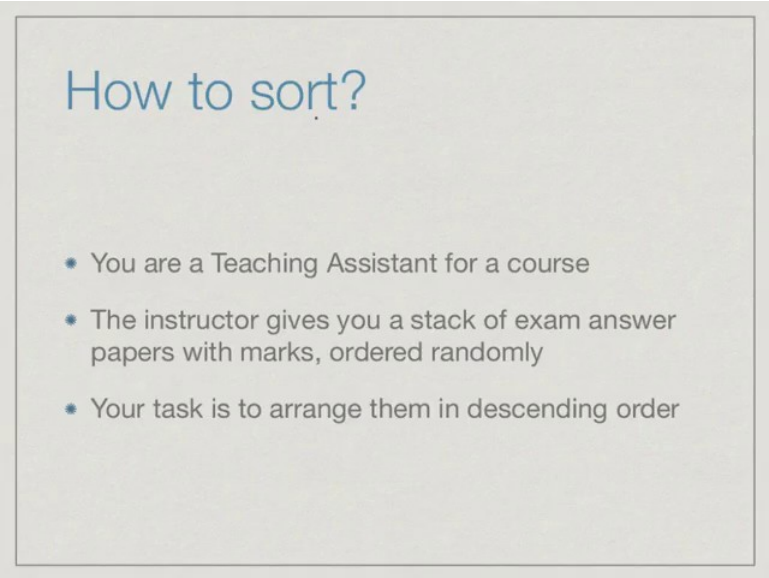
Now our claim is that this will stop working effectively around 5000. So, let us see if I make this as 1000 instead of 500, and run selection sort then you can see there is an appreciable gap. Now if I do it for say 2000, then there is slightly longer gap. If I do it for 5000 then you can see it takes 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

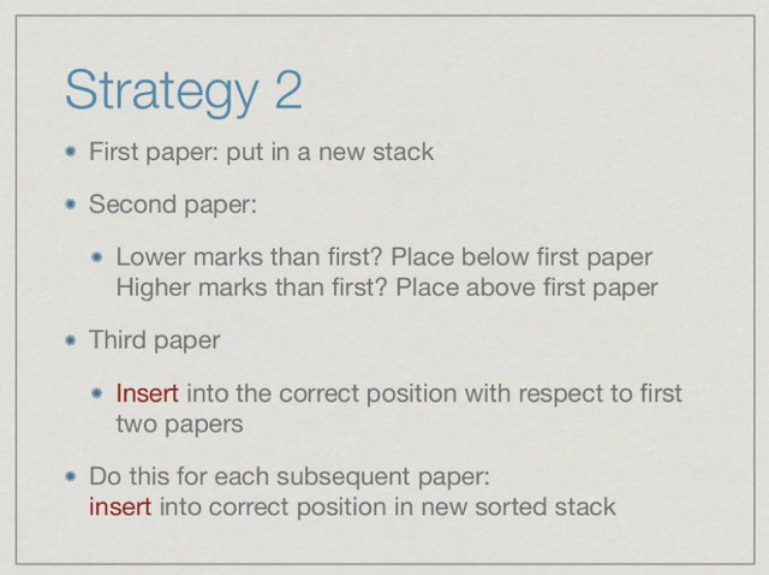
In the previous lecture we saw one natural strategy for sorting, which you would apply when we do something by hand namely selection sort.

(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:

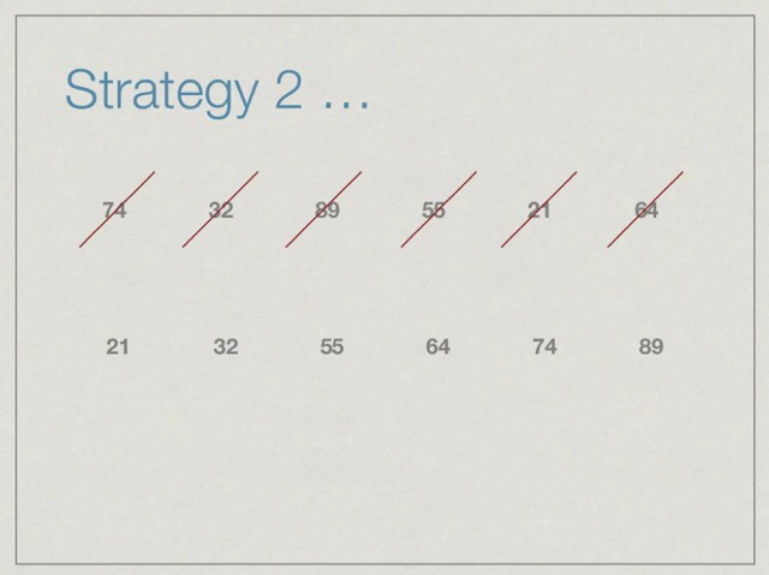
(Refer Slide Time: 00:17)



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.

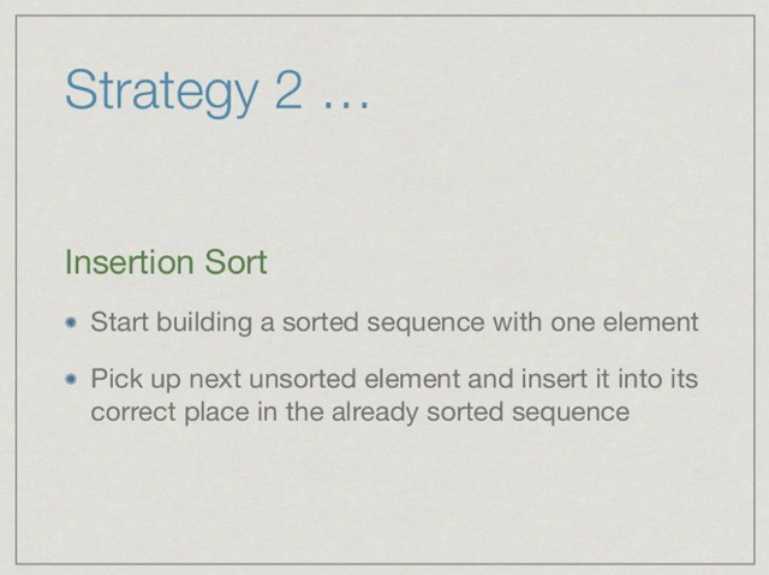
(Refer Slide Time: 01:31)



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.

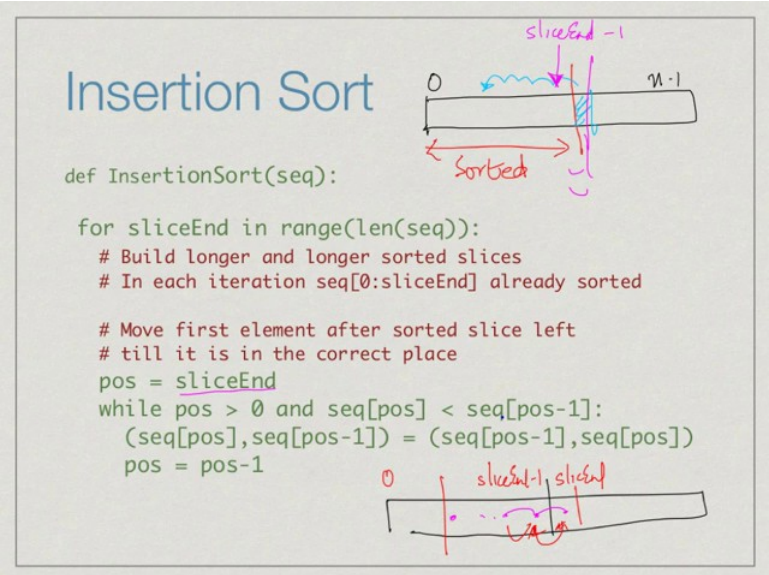
So, eventually it settles down as 32, 55, 74, 89. 21, similarly I have to start from the top and say it is smaller than 89 smaller than 74 smaller than 55 smaller than 32, so it goes all the way to the left. And finally, 64 will move down to positions past 84 and 89 and 74, but it will stop above 55. 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

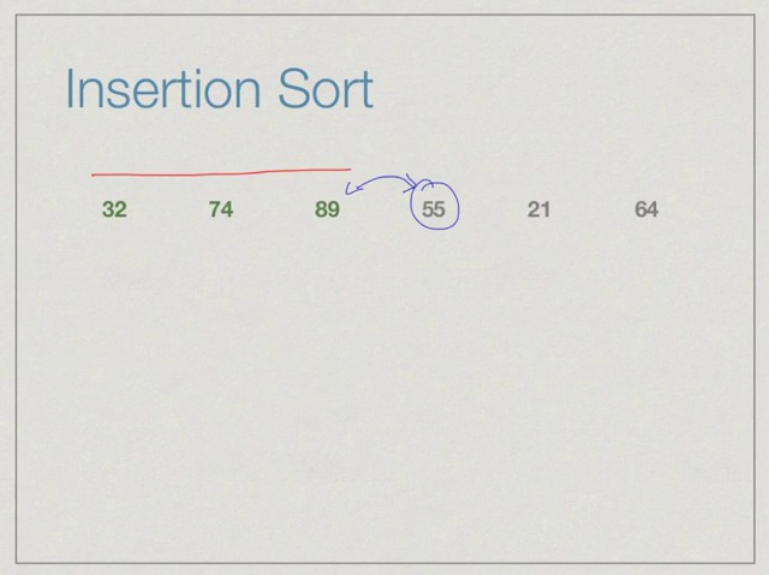
that at any given point, we have our sequence from 0 to n minus 1 and there are some positions, so that up to this point everything is sorted.

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.

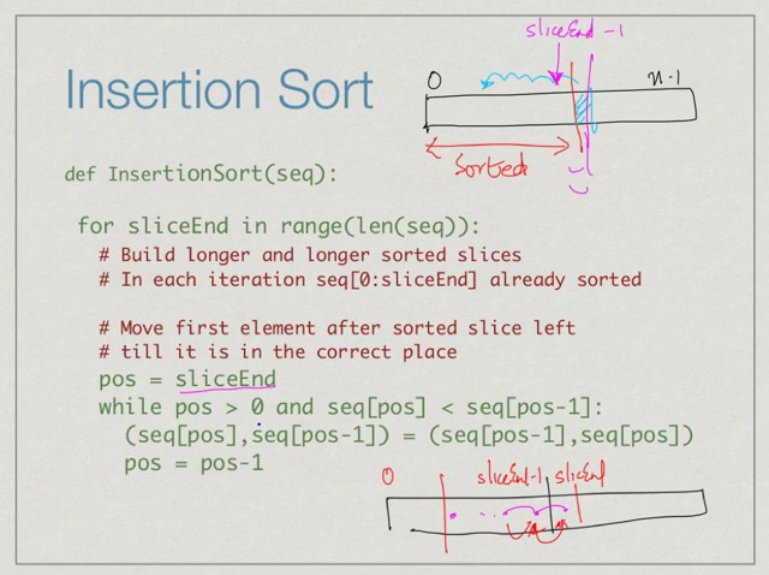
(Refer Slide Time: 05:29)



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.

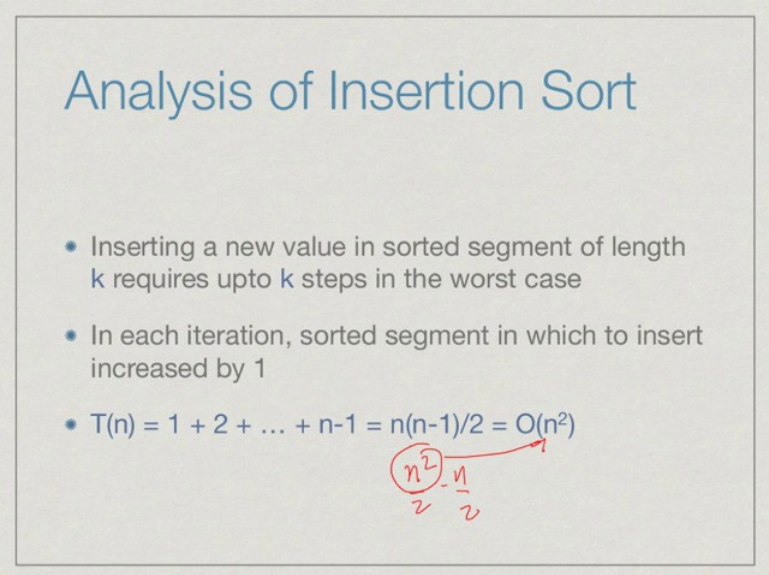
Now we will compare 55 again to the value to its left again, we will exchange. Now we will compare 55 to the value to its left and there is no change. Now we have a sorted list of length 4. Similarly, we will take 21 right, and we will compare it to 89; since 21 is smaller than 89, it will swap; since 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)

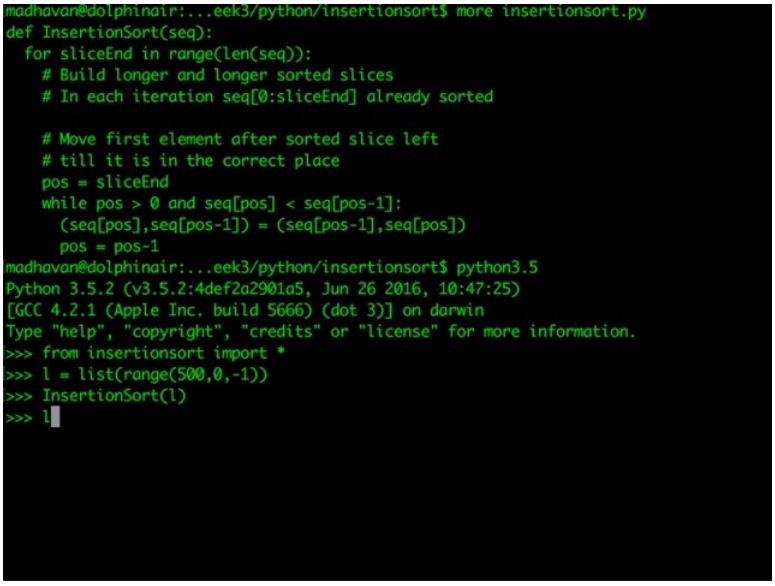


How do we analyze this? Well, at each round, what are we doing, we are inserting a new

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.

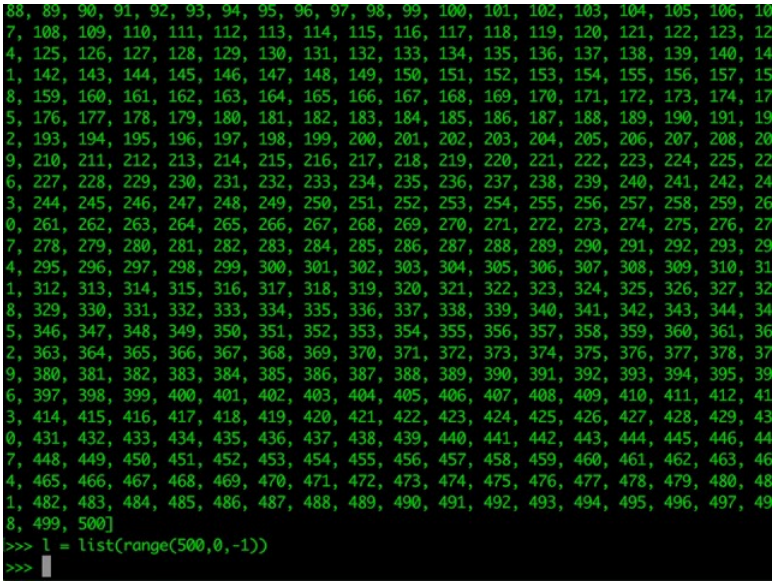
Sorting a segment of length k in the worst case takes k steps, so again we have the same recurrence not expression that we had for selection sort says that T of n is 1 plus 2 plus 3 up to n minus 1 which is n into n minus 1 by 2 which is order n square. So, again remember that this is n square by 2 minus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

(Refer Slide Time: 08:10)



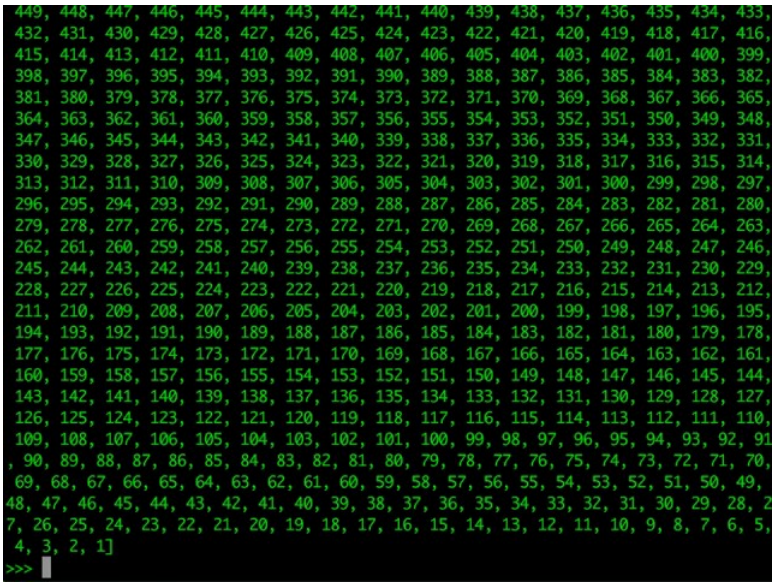
Once again let us see how insertion sort actually works in the python interpreter and we will see something slightly different from selection sort when we run it. First, let us look at the code. This is the code that we saw in the slide. We just keeps scanning segments, keeps taking a value at a position 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.

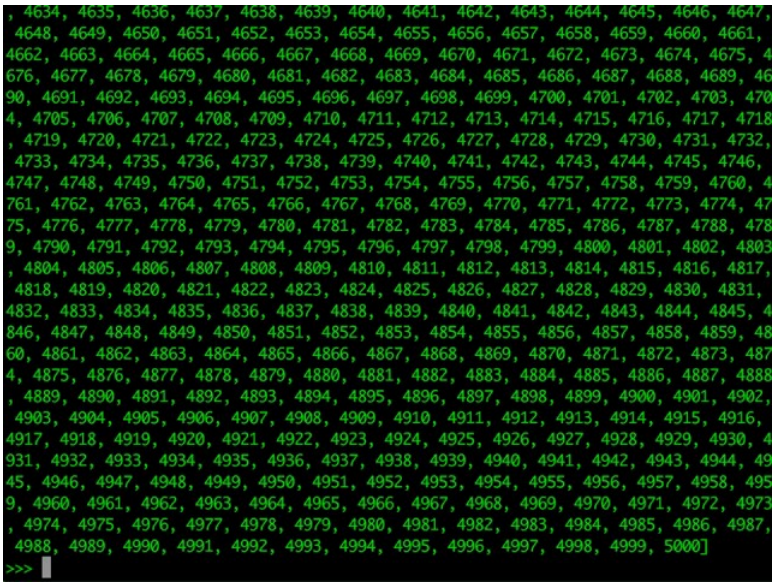
(Refer Slide Time: 09:01)



(Refer Slide Time: 09:03)

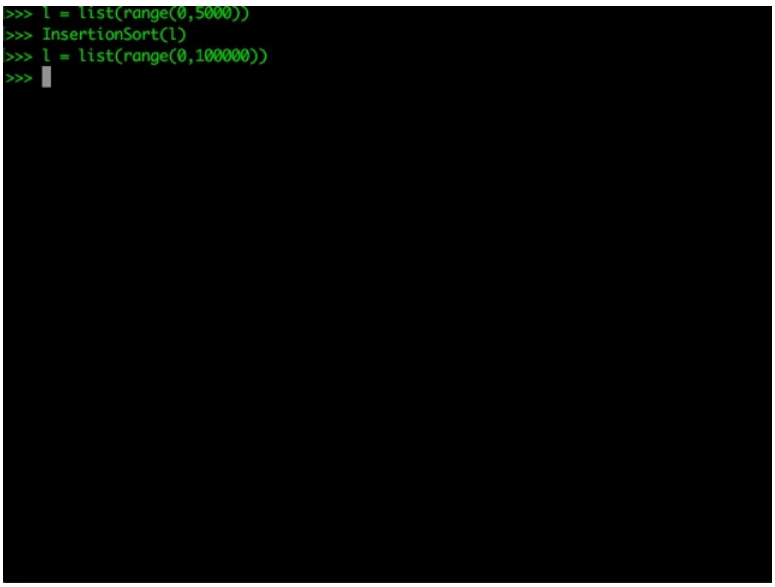
Now as before what we said is that if we try to do this for a length of around 5000 then it will be much smaller and much slower right. So, you can see it takes a long time and that is because InsertionSort, it is again order n square sort.

(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

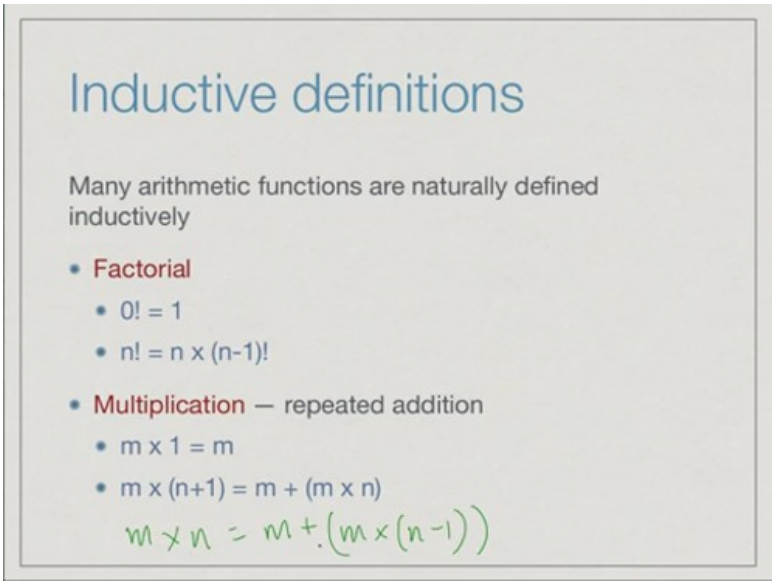
are order n squared sorts.

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)

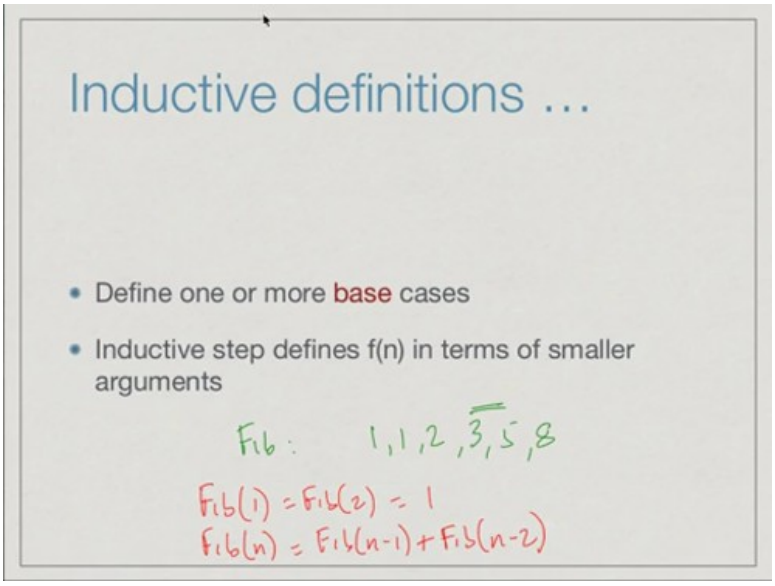


Recursive functions are typically based on what we call inductive definitions. So, in arithmetic many functions are naturally defined in an inductive way. Let us explain this by some examples. The first and most common example of a function defined inductively is the factorial function. So, we say that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

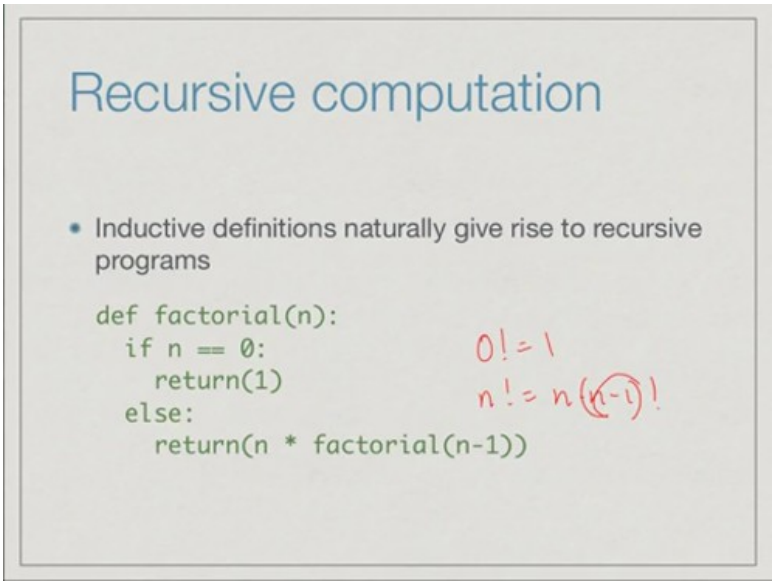
(Refer Slide Time: 01:50)



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.

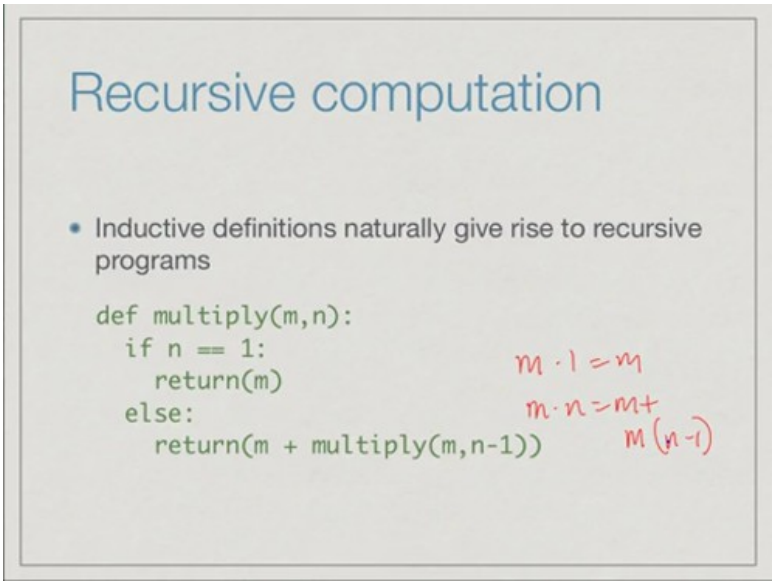
If you have seen the Fibonacci series, the Fibonacci series starts with 1 2 3 5 and so on and this is obtained by taking the previous two values and then adding. So, the general rule for Fibonacci is that the first value is equal to the second value is equal to 1 and after the second value Fibonacci 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)

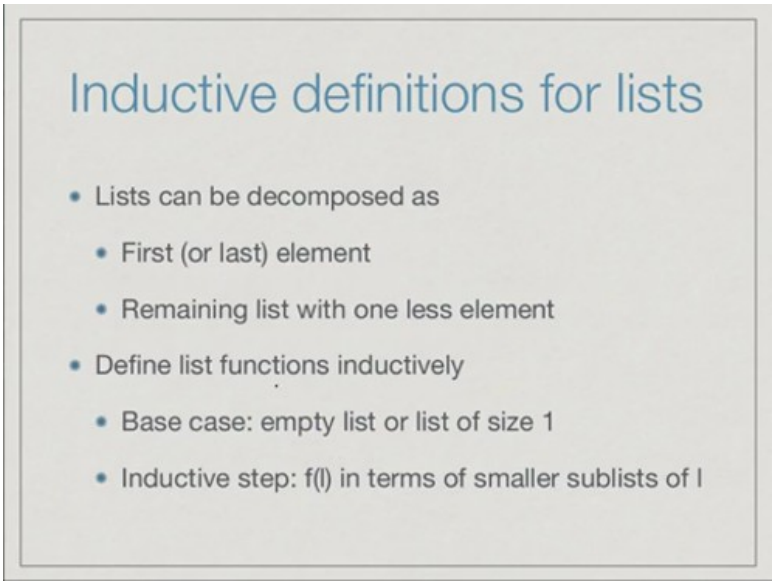


This is very clearly what we said before; it says zero factorial is 1 and otherwise if n is not 0, n factorial is n times n minus 1 factorial. So, this is exactly what we wrote before

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)

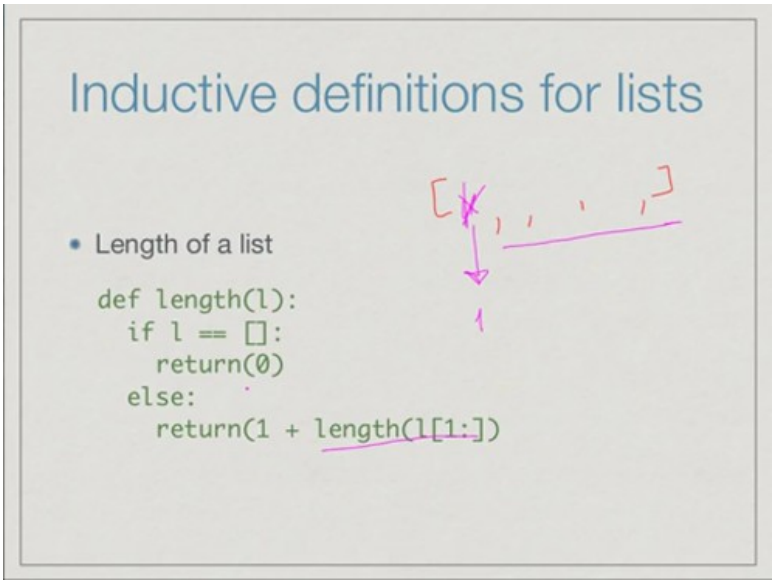


Now, what may be less obvious is that we can do the same thing for structures like list. A list has an inductive structure a list can be thought of as being built up from an empty list by adding one element at a time. So, we can think of decomposing a list, reversing the step. So, we start building a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

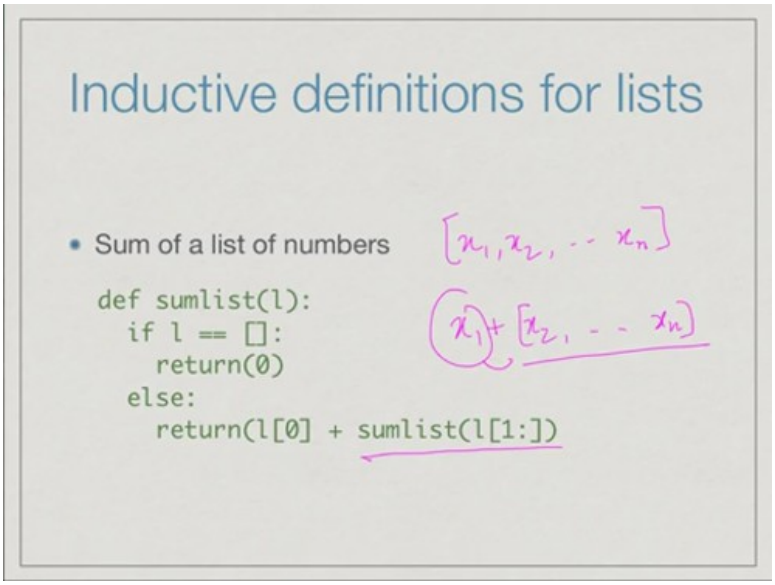
(Refer Slide Time: 05:15)



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.

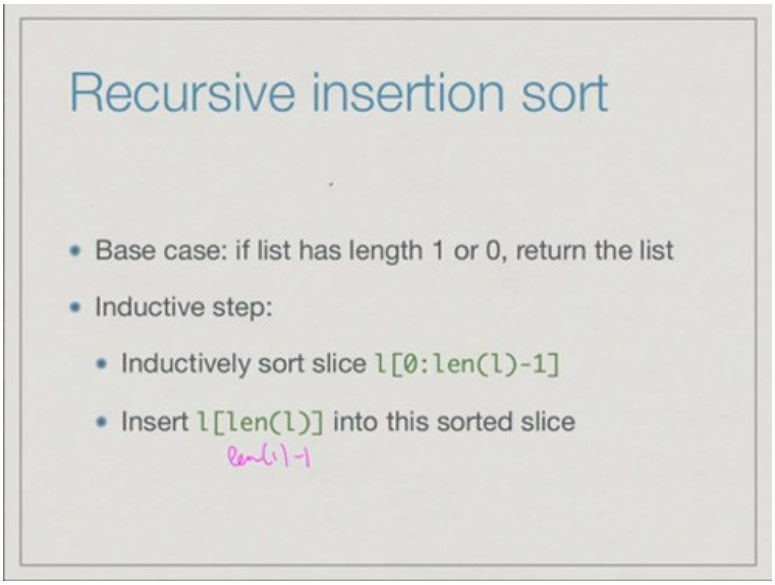
(Refer Slide Time: 06:00)



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.

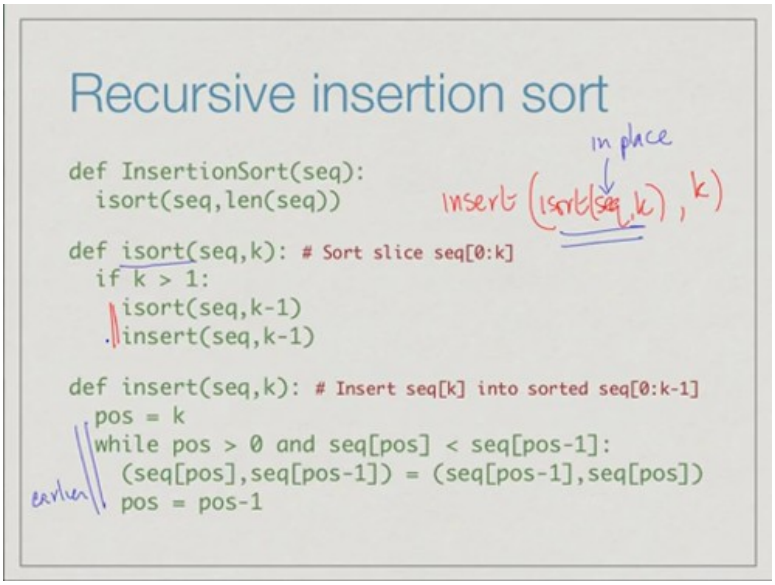
(Refer Slide Time: 06:45)



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)



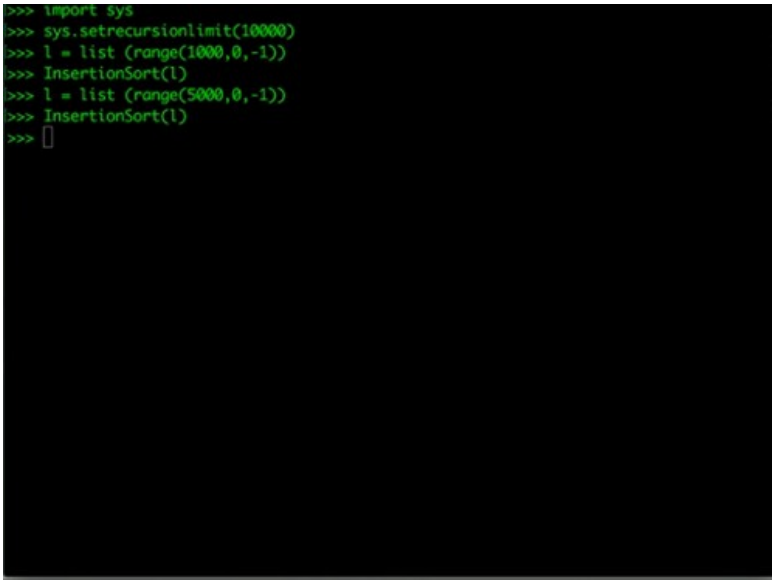
Here is a recursive definition of insertion sort in python. The natural thing in python or in any other thing would be to say that we want to insert the last position into the result of sorting the sequence up to, but excluding the last position, but the way we have written our insertion sort; this function 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.

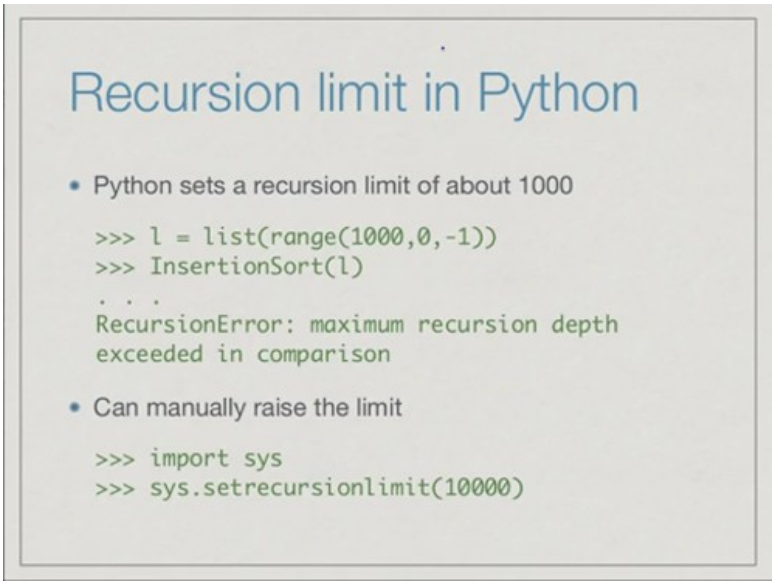
Now, fortunately there is a way to get around this. So, you can set this recursion limit and the way you do it is the following. You first have to import a bunch of functions which are called system dependent functions by saying import sys and then, in sys there is a function called set recursion limit 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

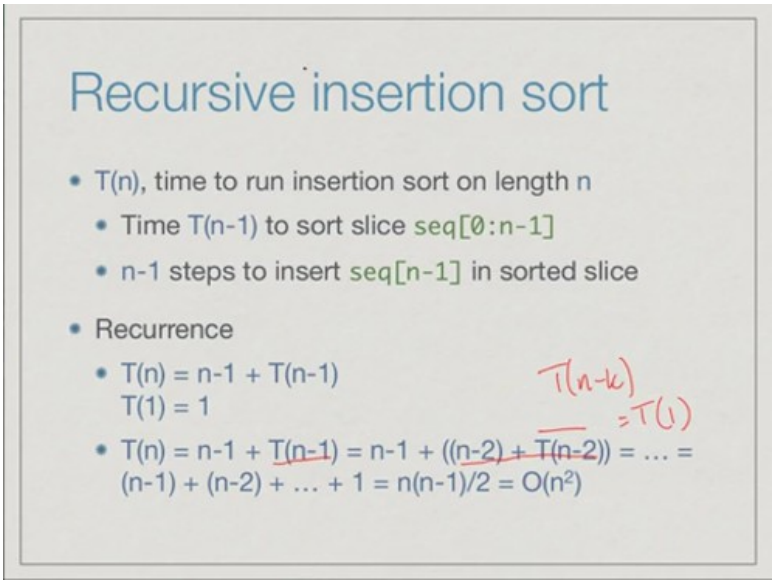
stops this and forces you to go and examine the code is by having a recursion limit saying beyond a certain depth, it will refuse to execute the code.

(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)

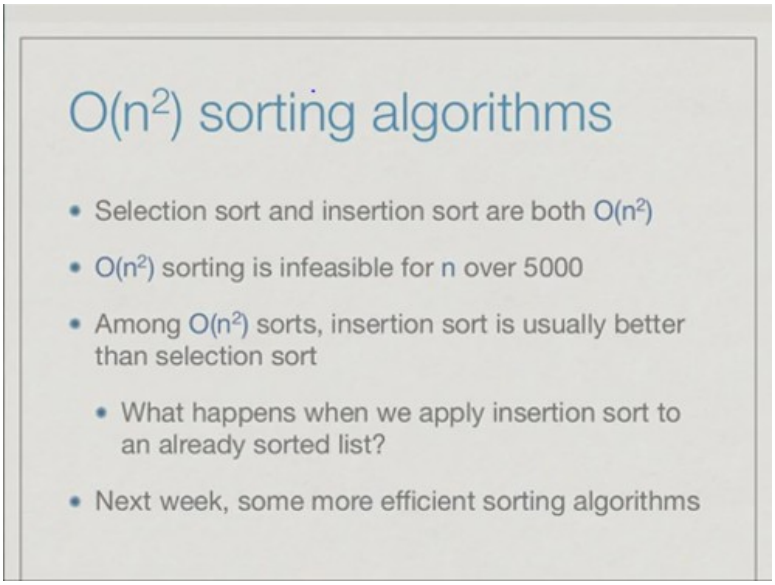


So how would we analyse the complexity of recursive insertion sort? So as before, we use T of n to denote the time it takes to run insertion sort on an input of length n. Insertion sort at the highest level consists of two steps. We first have to sort the initial

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.

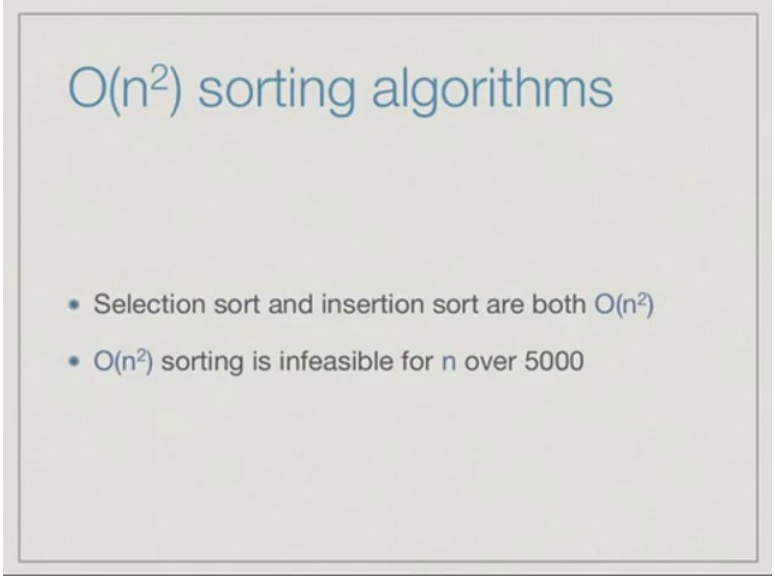
So if we have a list which is already sorted, then insertion sort will actually work much better than n squared, but we cannot rely on this, and anyway we are counting worst case complexity. So, we have to take the fact that both of these will in general not work very well for lists larger than 5000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

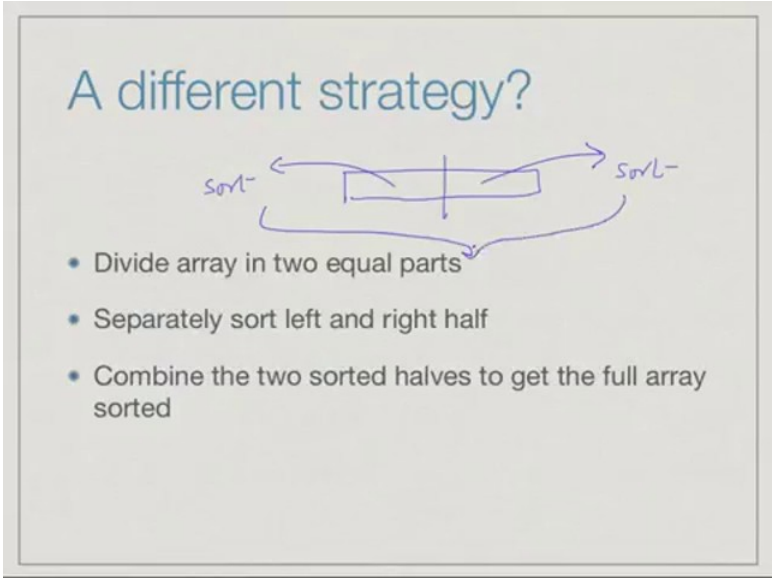
(Refer Slide Time: 00:02)



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.

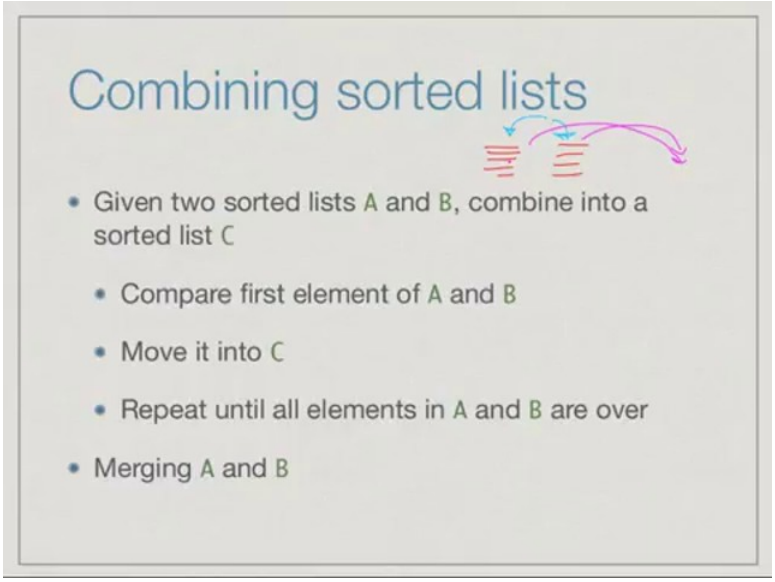
(Refer Slide Time: 00:37)



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.

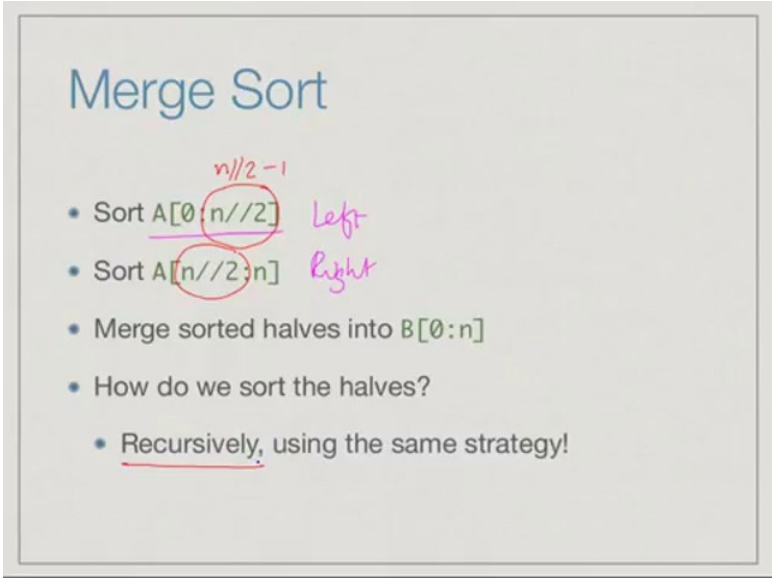
Now you have the second highest on the right hand side and the first the highest on the left hand side. Again, you look at the bigger one and move that one here and so on. So, at each time, you look at the current head or top of each of the lists and move the bigger one to the output, right. And if you 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)



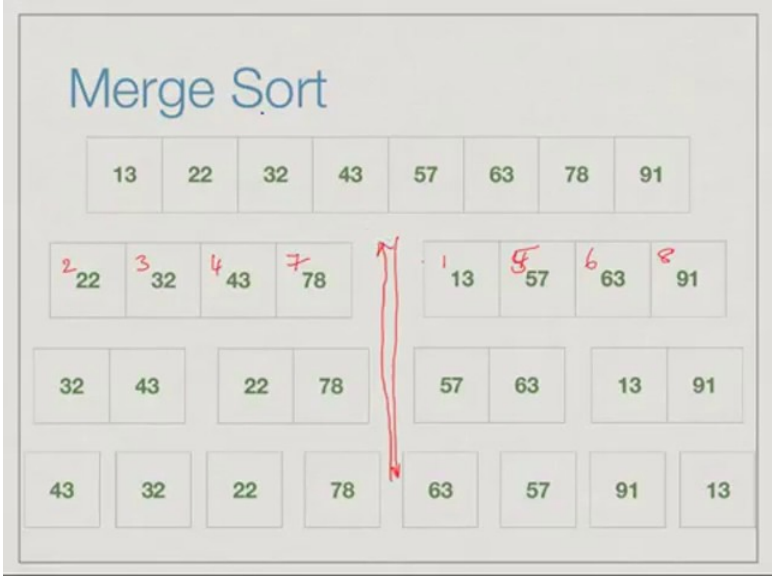
Having done this, now that we have a procedure to merge two-sorted list into a single-sorted list; we can now clarify exactly what we mean by merging the things using merge sort. So, merge sort is this algorithm which divides the list into two parts and then combines the sorted halves into a single part. 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.

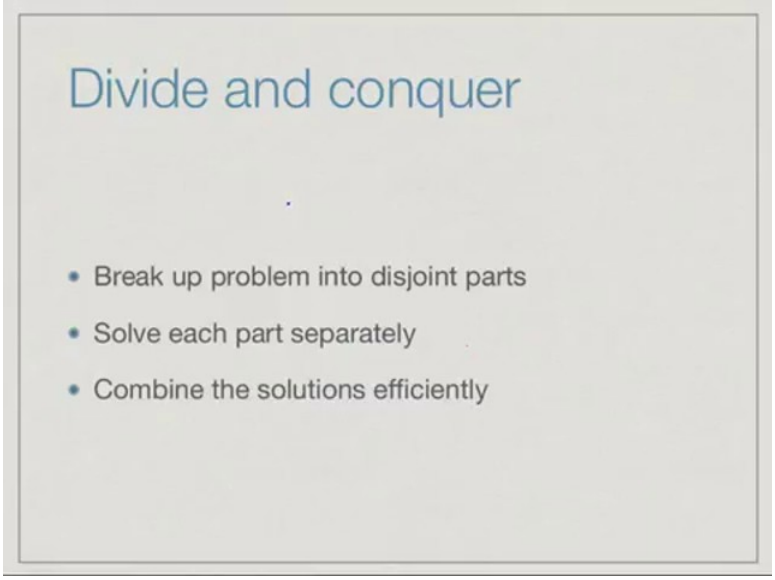
Now we have list of length two, we could sort them by hand, but we say that we do not know how to sort anything except a list of length 1 or 0, so we further break it up. Now, we have trivial lists 43 and 32 on the left, 22 78 and so on, so we end up with eight lists of length one which are automatically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

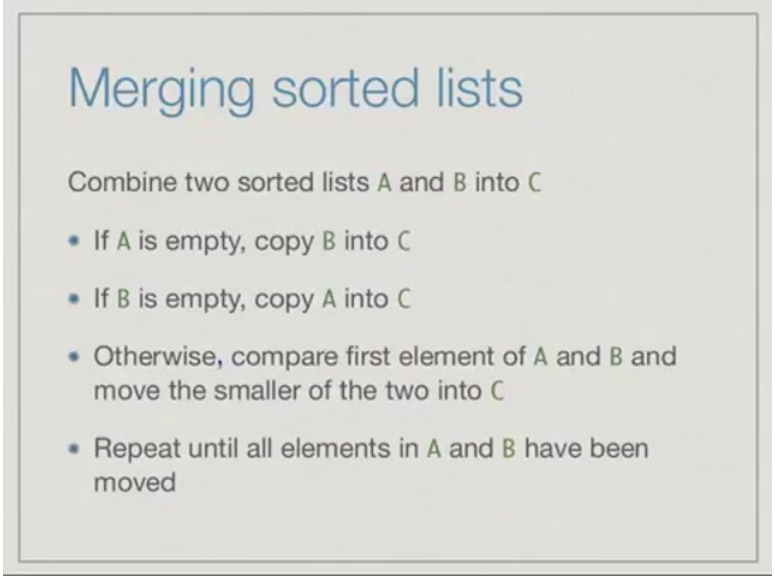


This strategy that we just outlined from merge sort is a general paradigm called divide and conquer. So, if you have a problem where you can break the problem up into sub problems, which do not have any interference with each other. So, here for instance, sorting the first half of the list and sorting 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.

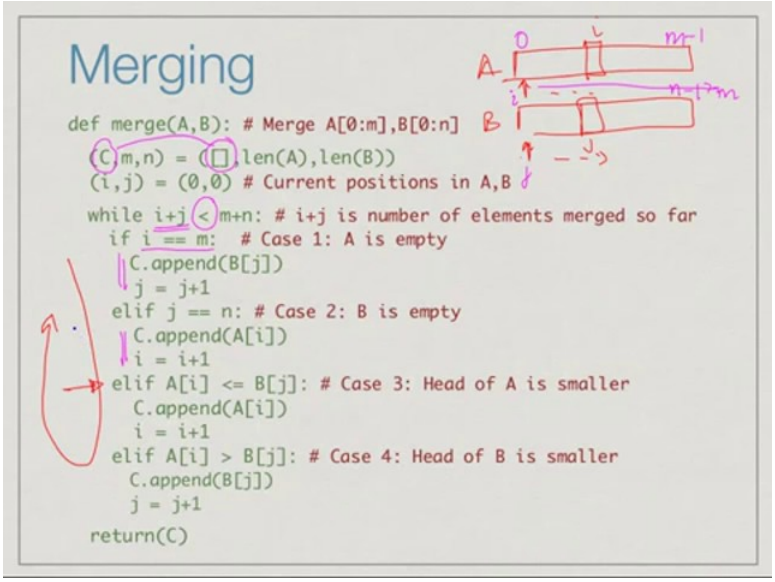
(Refer Slide Time: 08:08)



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.

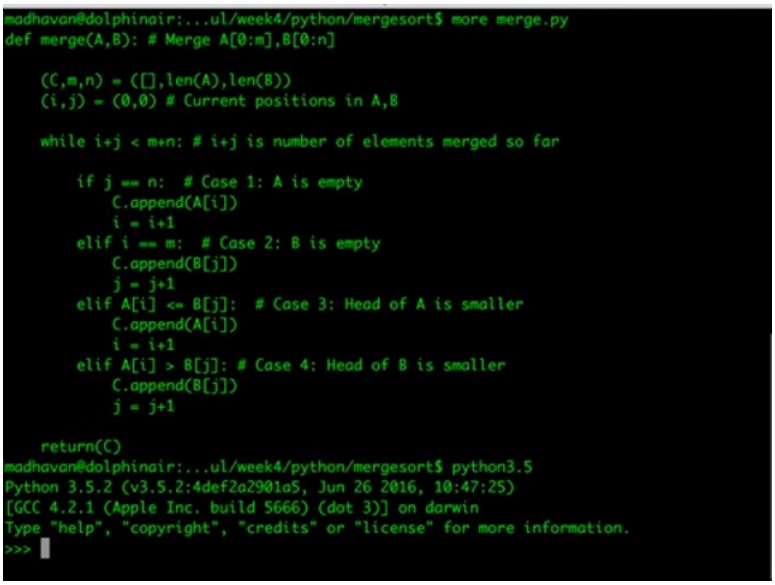
And initially, these indices point to the starting element which is 0. So, as we move along if we move i from 0 to 1 that means, we have processed one element in A; if we move it to 3; we have processed three elements in A and so on. So, at any given time i plus j will tell us how many elements have 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.

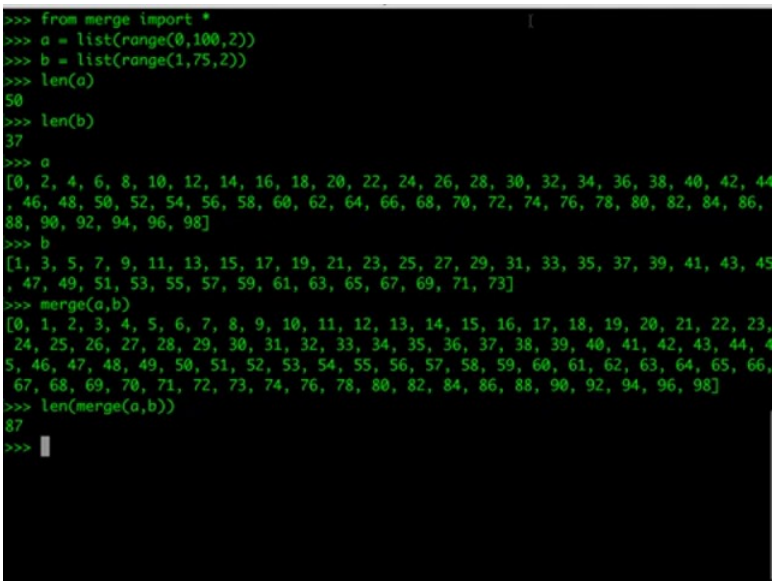
Now, if neither list is exhausted then we have to do a comparison. So, we come to this case and we check whether the element in A is smaller than or equal to the element in B. So, at this point we are in general looking at some A i and some B j. So, we have to decide which of these two goes into C next. 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)

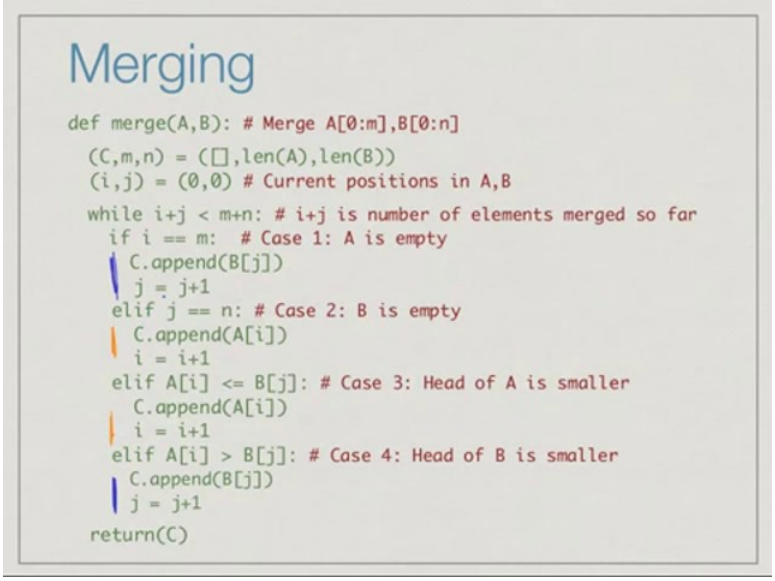


The simplest way to do this is to try and construct two lists; suppose, we take a list of numbers where the even numbers from 0 to 100. So, we start at 0 and go to 100 in steps

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.

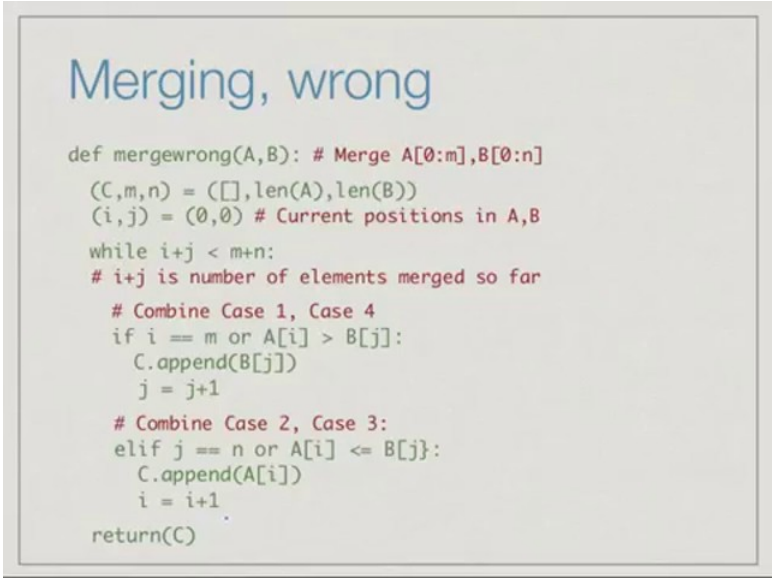
Now if we say merge a, b, and then we get something which actually returns this merge list. Notice that up to 73, which is the last element in b, we get all the numbers. And then from here, we get only the even numbers because we are only copying from a. And if you want to check the length of the merge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

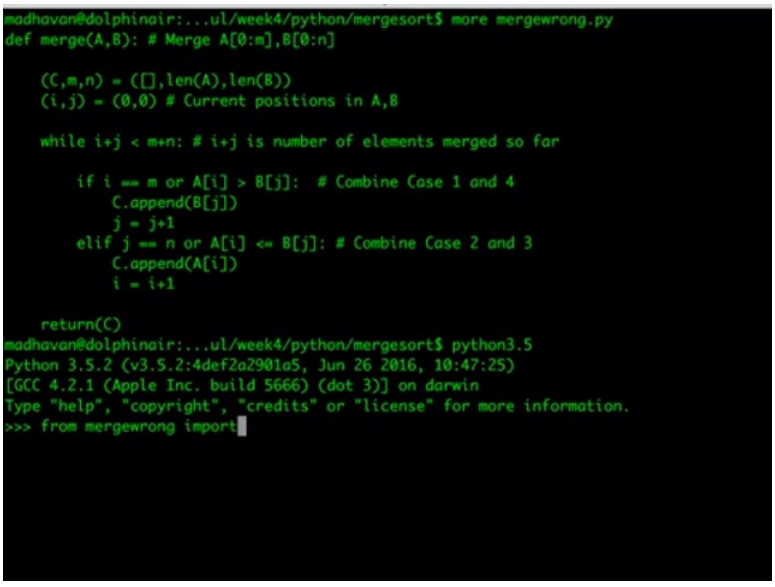
(Refer Slide Time: 14:57)



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.

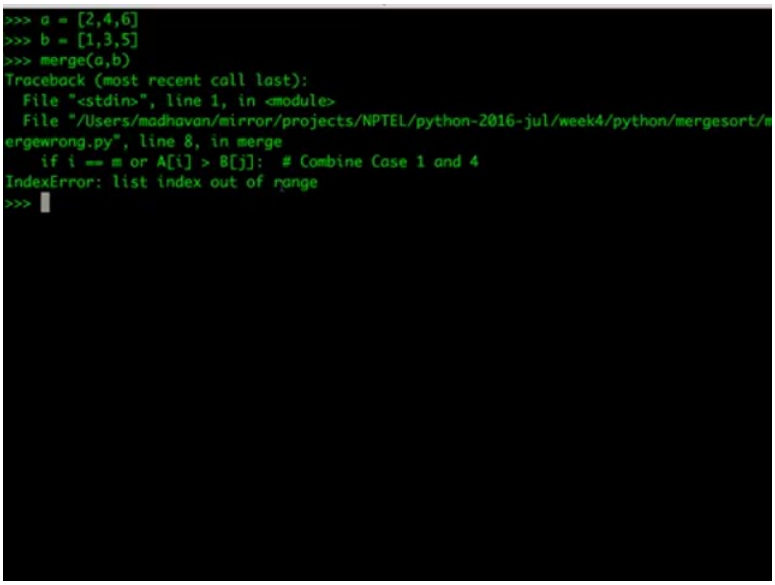
On the other hand, either if B is empty or A has a smaller value then you take the value from A and append the index in that right. Let us see what happens if we try to run this code.

(Refer Slide Time: 15:20)



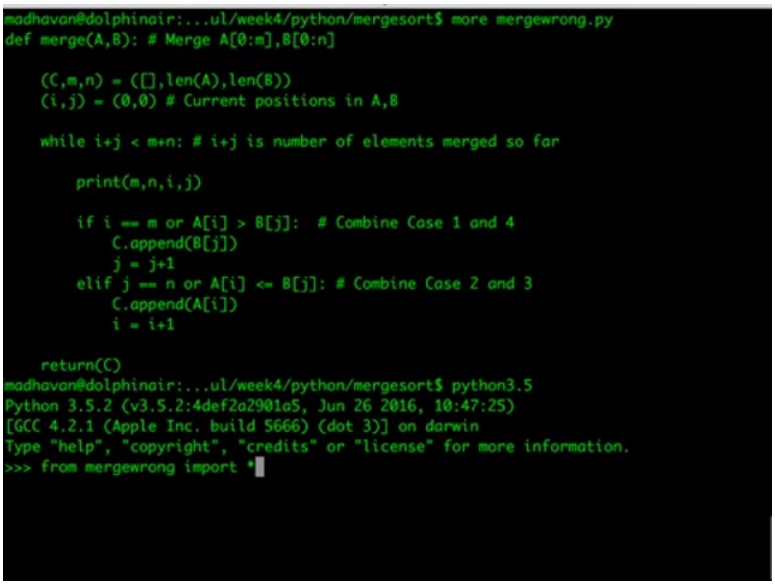
Here we have a file merge wrong dot py, the name suggests that is going to be a problem, where we have combined case 1 and 4, where we append the element from B into C and 2 and 3 where we combine the element append the element from A into C. Let us run this and see. Now, we take merge wrong at the starting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

(Refer Slide Time: 15:46)



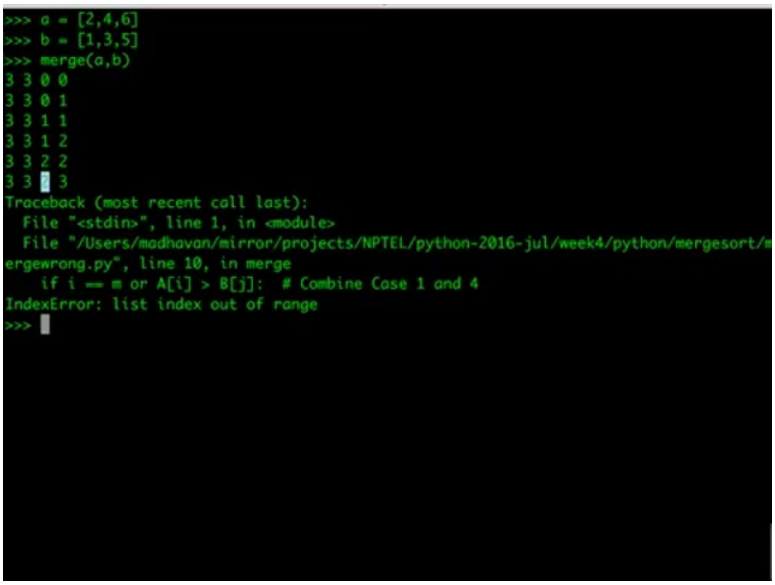
And let us just do a simple case. Supposing we take a as 2, 4, 6 and b as 1, 3, 5 then we have to expect 1, 2, 3, 4, 5, 6. Let us try to merge a and b right and now we get an error which says that we have a list index out of range. List index out of range suggests that we are trying to access some element 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)



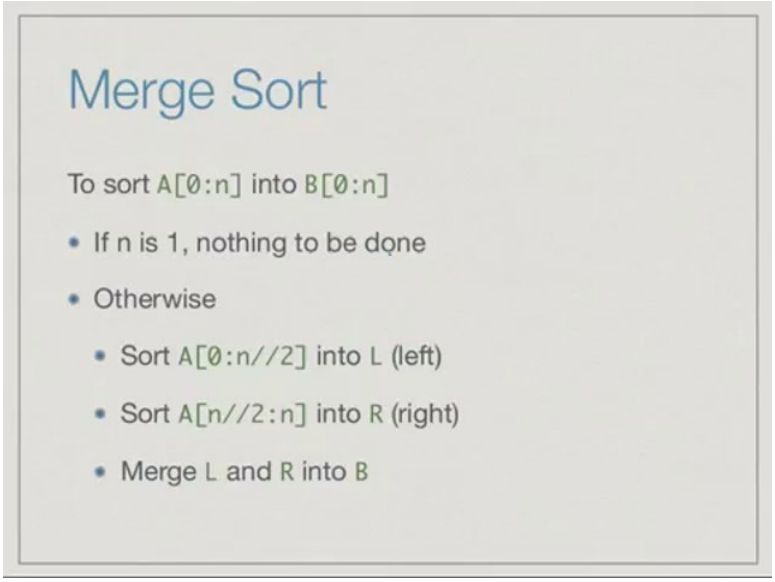
Setup a and b as before. So, a is 2, 4, 6; b is 1 3 5. And now we run merge. And now we see what is happening. So, m and n are the initial lengths 3 and 3. And these are the values 0 and 0 are i and j the pointers. So, j becomes 1 then i become 1 and so on.

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.

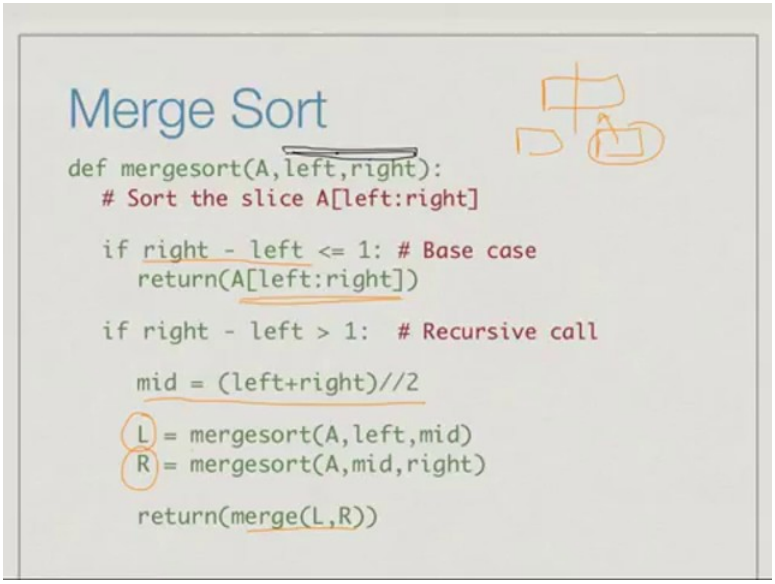
Otherwise, we have to have a separate condition saying if i is equal to m or j is less than n and which becomes more complicated than the version we had with four explicit cases. So, we may as well go back to the version with four explicit cases.

(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.

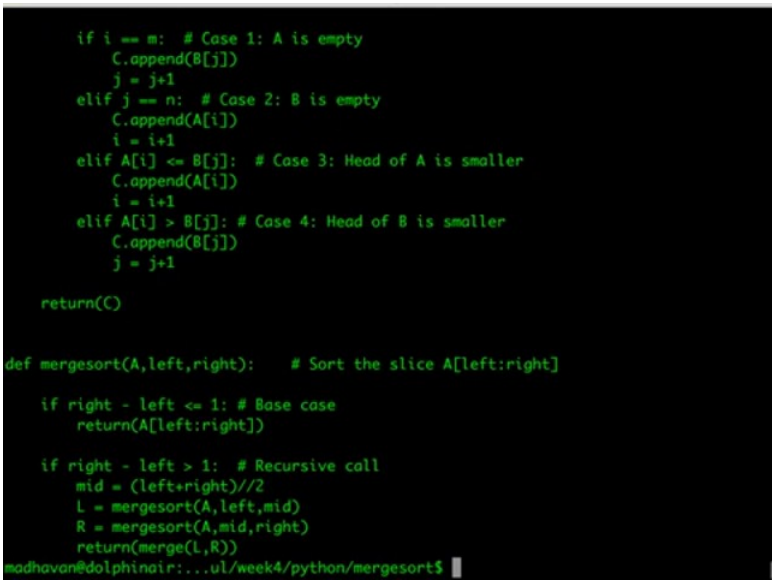
The only thing to remember is that we have to augment our merge sort function with these two things, the left point and the right point. We had a similar thing if you remember for binary search, where we recursively kept having the interval to search. So, we have to keep telling it in which interval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

(Refer Slide Time: 20:53)

```
def merge(A,B): # Merge A[0:m],B[0:n]

    (C,m,n) = ([],len(A),len(B))
    (i,j) = (0,0) # Current positions in A,B

    while i+j < m+n: # i+j is number of elements merged so far

        if i == m:  # Case 1: A is empty
            C.append(B[j])
            j = j+1
        elif j == n:  # Case 2: B is empty
            C.append(A[i])
            i = i+1
        elif A[i] <= B[j]:  # Case 3: Head of A is smaller
            C.append(A[i])
            i = i+1
        elif A[i] > B[j]: # Case 4: Head of B is smaller
            C.append(B[j])
            j = j+1

    return(C)


def mergesort(A,left,right):    # Sort the slice A[left:right]

    if right - left <= 1: # Base case
        return(A[left:right])
mergesort.py
```

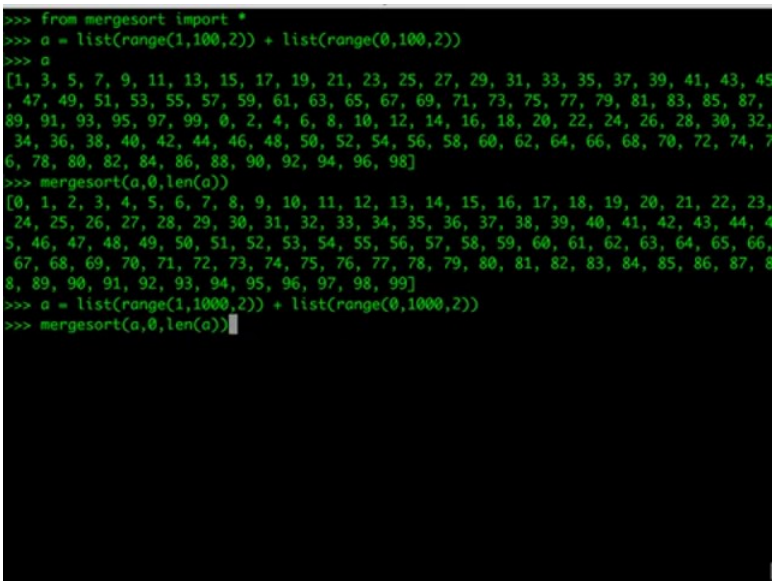
Let us look at a python implementation of merge sort. So, here we have a file merge sort dot py. We start with the function merge, which we saw before with a four way case split in order to shift elements from A and B to C.

(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)



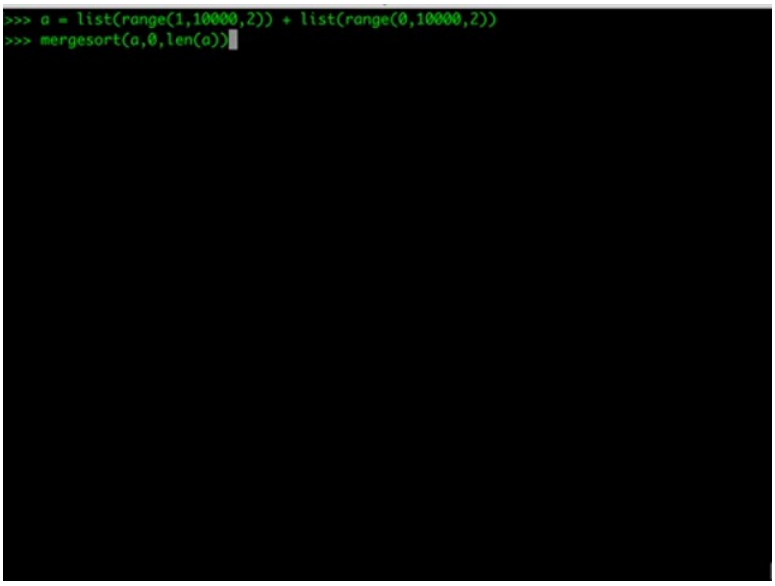
We take python and we say from merge sort, import all the functions. And now let us take a larger range. Suppose, if we take all the odd numbers followed by all the even numbers. So, we say range 1 to 100 in steps of 2. Those are the odd numbers and then all

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)

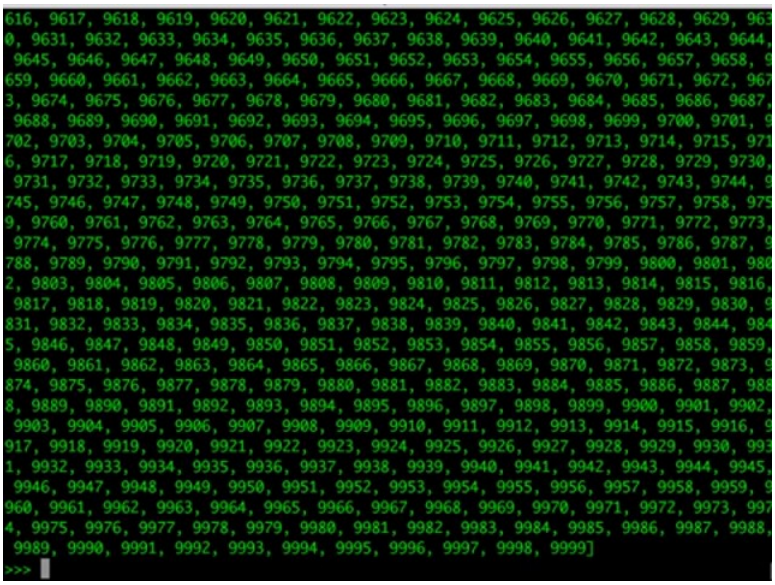
Now what if I take a larger list 1000 then I get, again everything sorted.

(Refer Slide Time: 22:16)



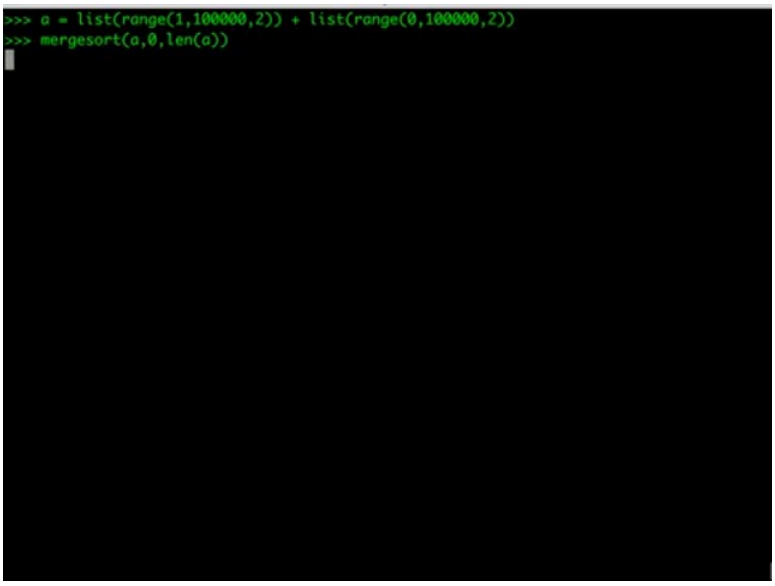
Now our claim is that this is an order n log n algorithm. It should work well for even bigger list. If I say 10000 which remember would take a very long time with insertion sort or selection sort.

(Refer Slide Time: 22:26)



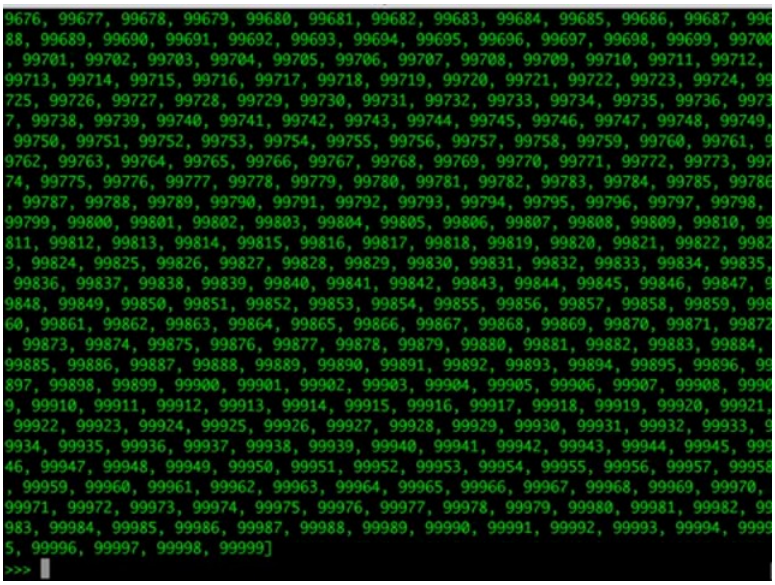
Question is how long it takes here and it comes out quite fast.

(Refer Slide Time: 22:29)



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

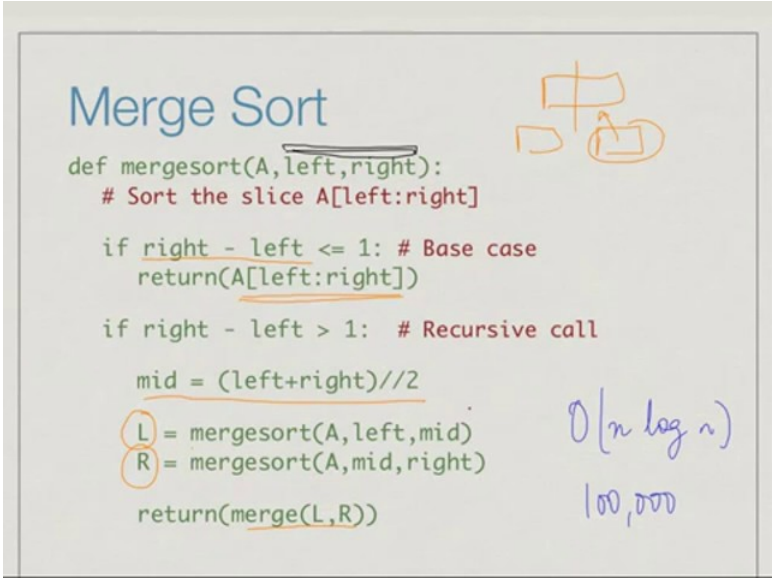
(Refer Slide Time: 22:37)



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

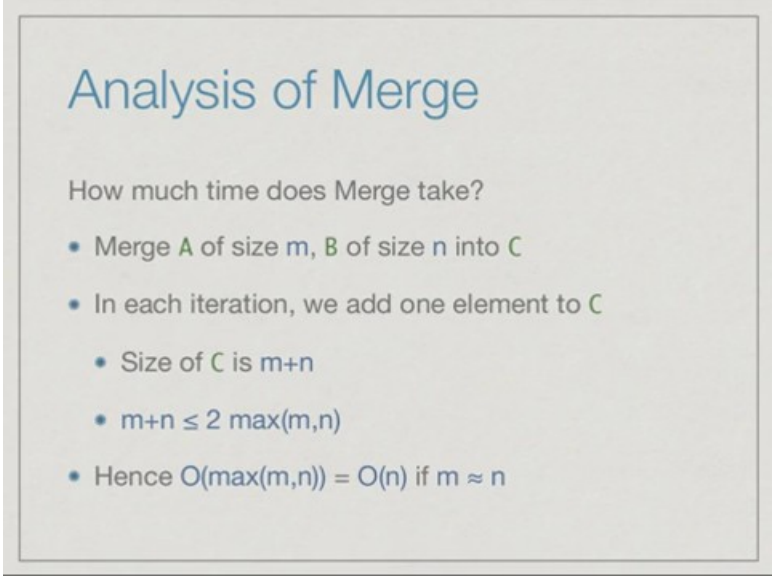
Week - 04
Lecture - 02
Merge Sort, Analysis

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)



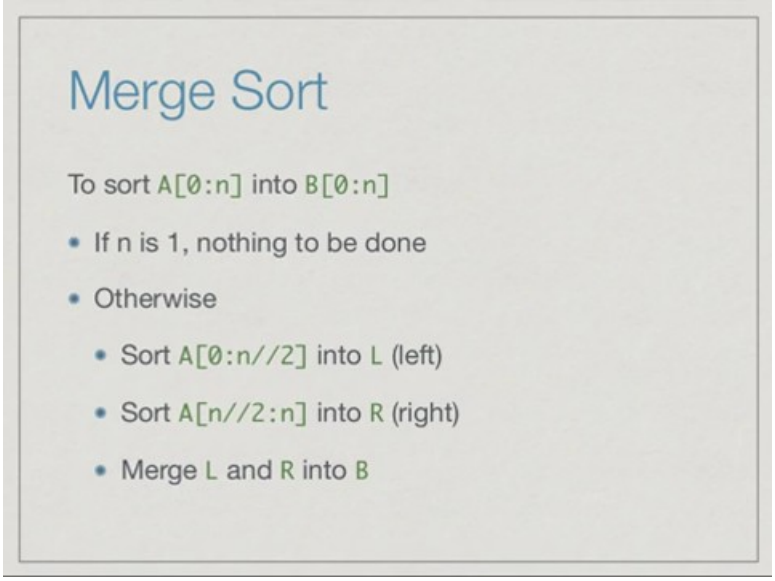
In order to analyze merge sort, the first thing we need to do is to give an analysis of the merge function itself. How much time does merge take in terms of the input sizes of the two lists, A and B. So, suppose A has m elements and B has n elements and we want to put all these elements together into 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

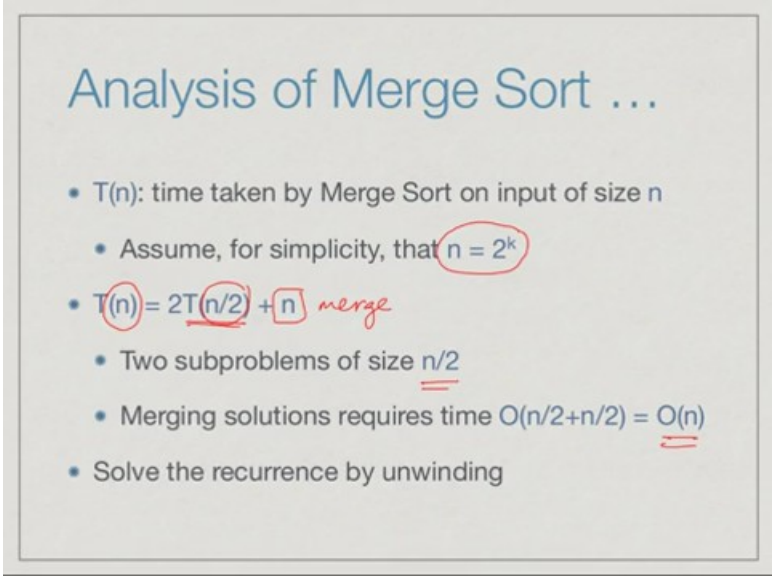
the same, approximately the same size, then the max of m and n is just one of them in itself. Essentially merge is linear in the size of the input list.

(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.

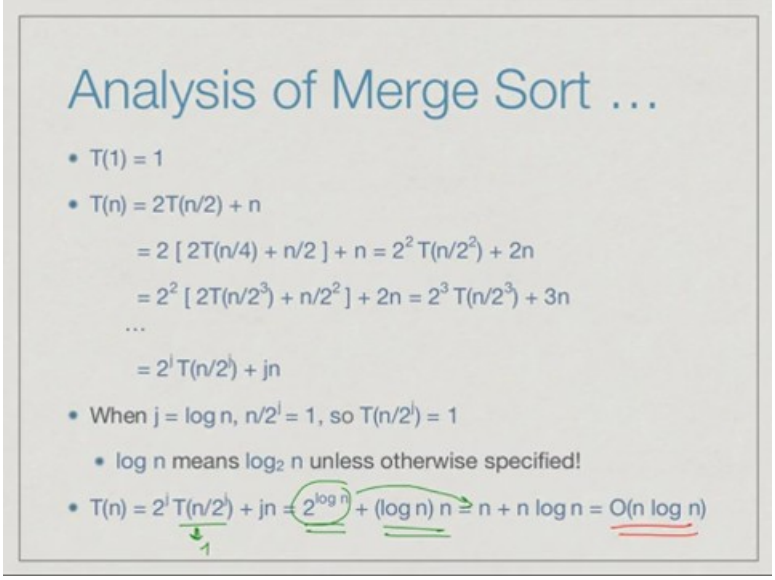
(Refer Slide Time: 02:23)



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)

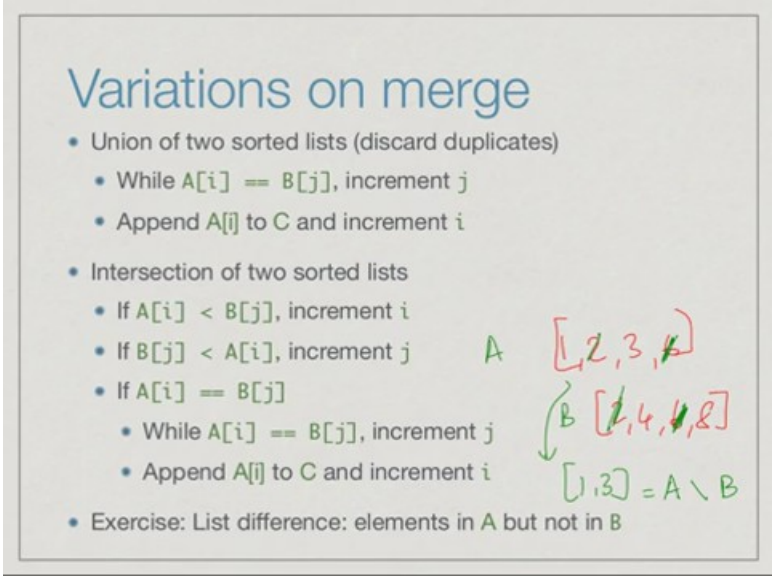


We start with the base case. If we have a list of size 1, then we have nothing to do. So, T of n is 1 and T of n in general is two times T n by 2 plus n. If we expand this out, we read substitute for T n by 2 and we get two times T n by 4 plus n by 2 because that if we take this as a new input, this 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

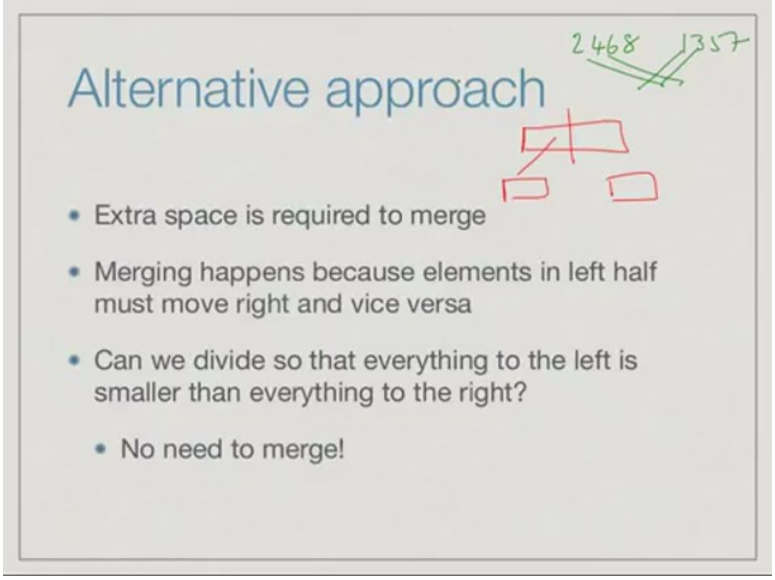
Quicksort

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.

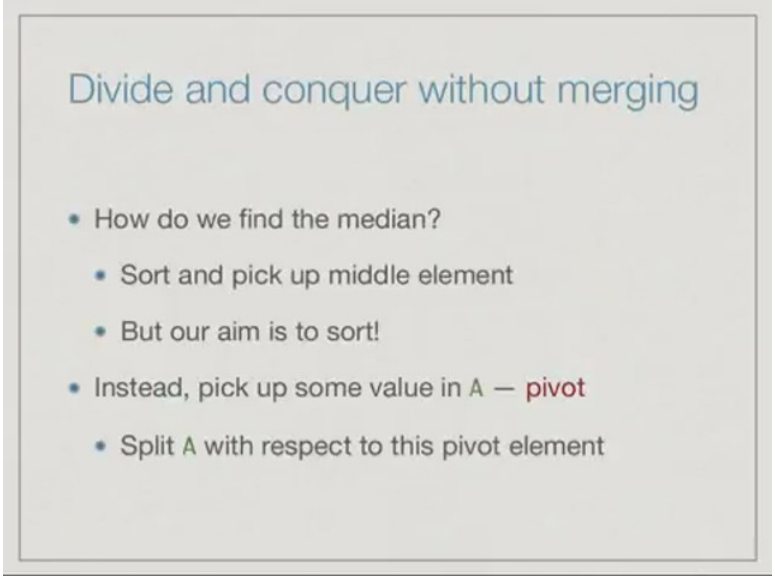
(Refer Slide Time: 00:23)



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.

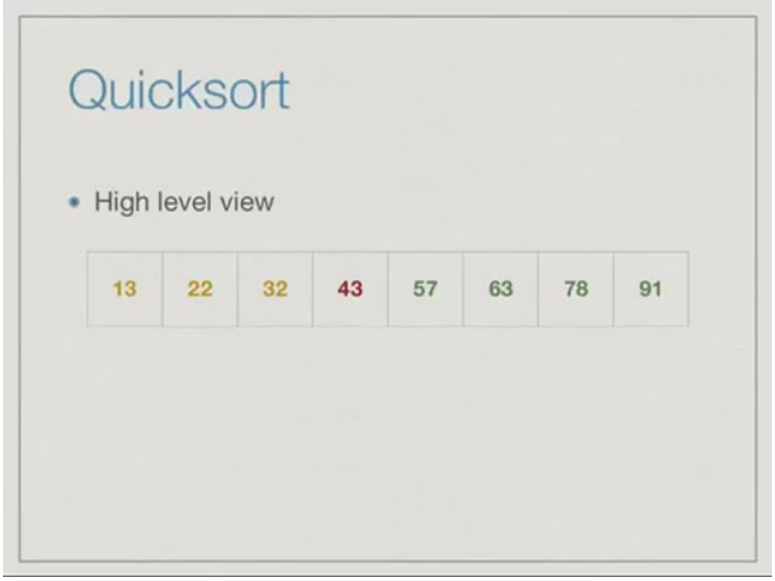
Having done this rearrangement moving all the smaller values to the left half and the bigger values to the right half then we can recursively apply this divide and conquer strategy and sort the right and the left half separately and since we have guaranteed that everything in the left half is smaller 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)



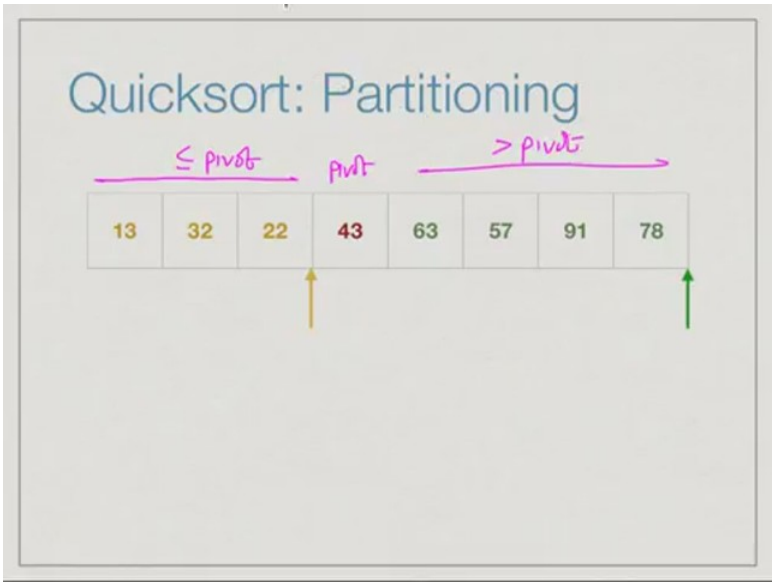
This algorithm is called Quicksort, it was invented by a person called C.A.R Hoare in the 1960s and is one of the most famous sorting algorithms. So, we choose a pivot element which is usually the first element in the list of the array. We partition A, into the lower part and the upper part with respect 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.

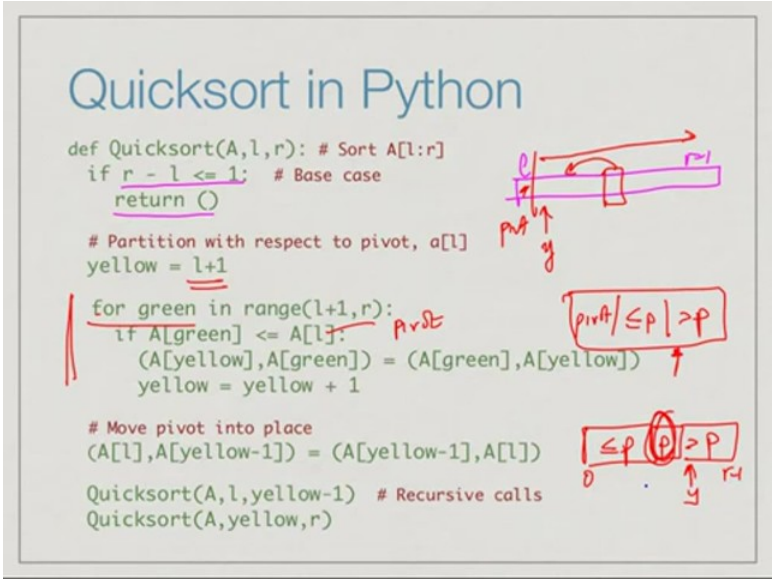
Now, we look at 63, again 63 extends the green zone, 57 extends the green zone, 91 extends the green zone. Now, we have to do something when we find 13. So, 13 is an element which has to be put into the yellow zone, one strategy would be to do a lot of shifting. We move 13 to where 22 is or after 22 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)



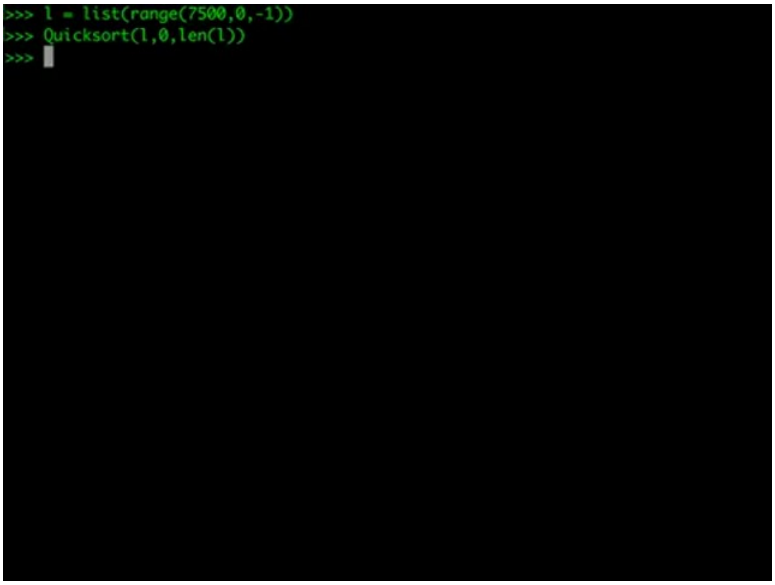
Here is an implementation in Python. So, remember that quicksort is going to have like merge sort and like binary search, we repeatedly apply it in smaller and smaller segments. In general, we have to pass to it the list which we call A, and the end points to the segment the left and the right. If we 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

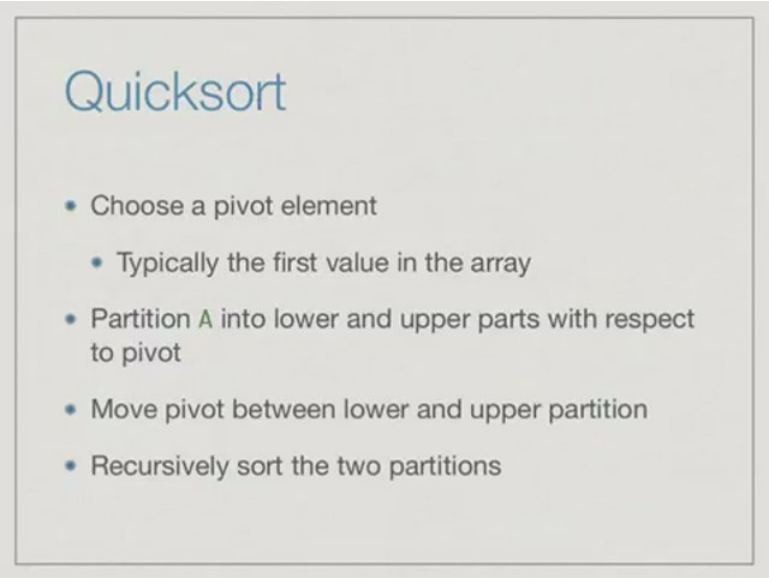
time. We saw that merge sort which was n log n could do 5000 and 10000 and even 100,000 instantaneously.

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

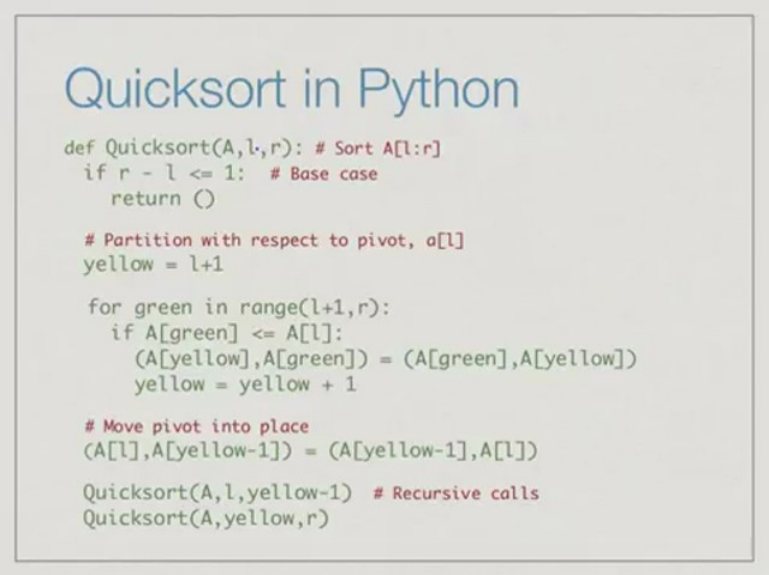
Week - 04
Lecture - 04
Quicksort Analysis

(Refer Slide Time: 00:01)



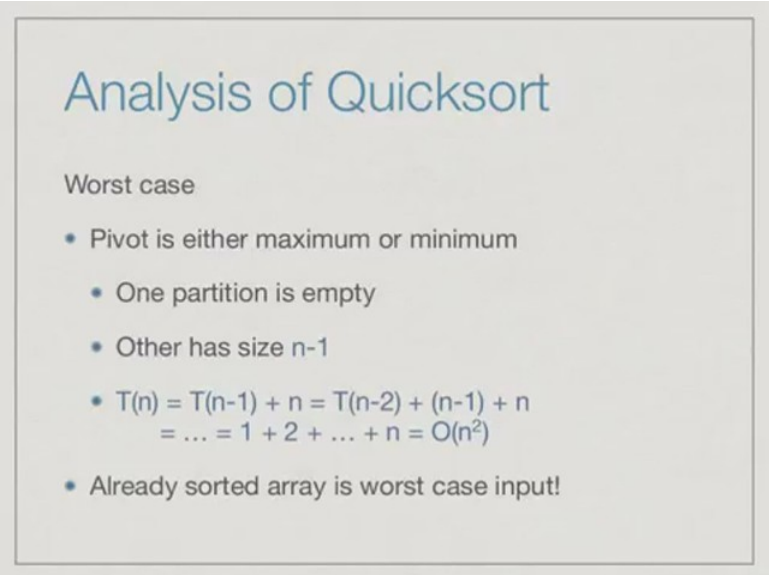
In the previous lecture, we saw quicksort which works as follows. You first rearrange the elements with respect to a pivot element which could be, well, say the first value in the array. You partition A, into the lower and upper parts, those which are smaller than the pivot, and those which are bigger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

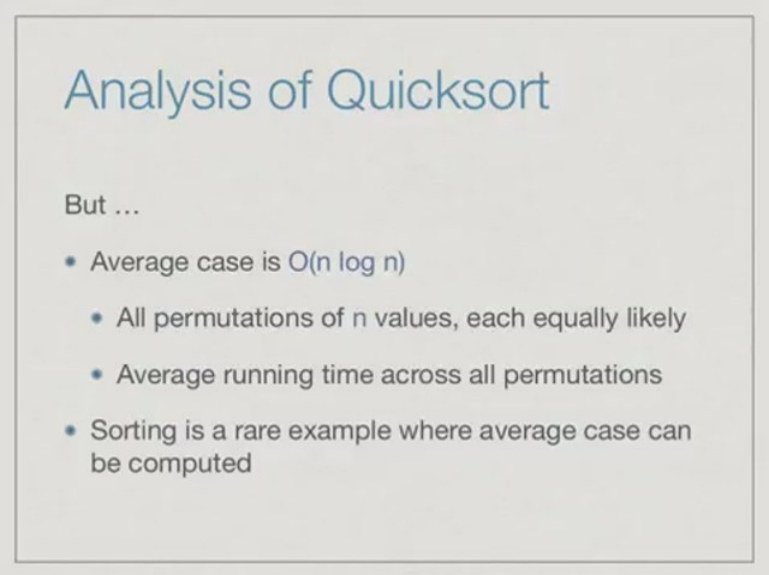
comes when the pivot is very far from the ideal value we want. The ideal value we want is median, the median would split the list into two equal parts and there by divide the work into two equal parts, but if the pivot happens to be either the minimum or the maximum value in the overall array, then supposing 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.

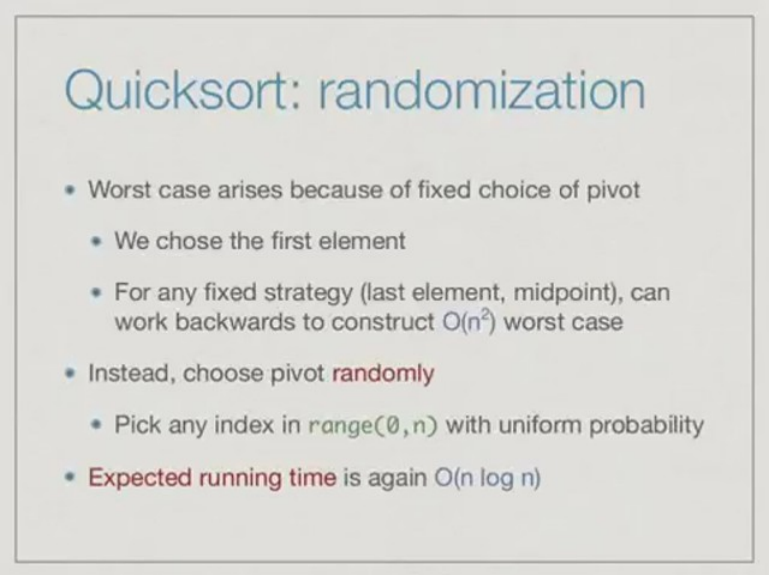
(Refer Slide Time: 02:40)



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)

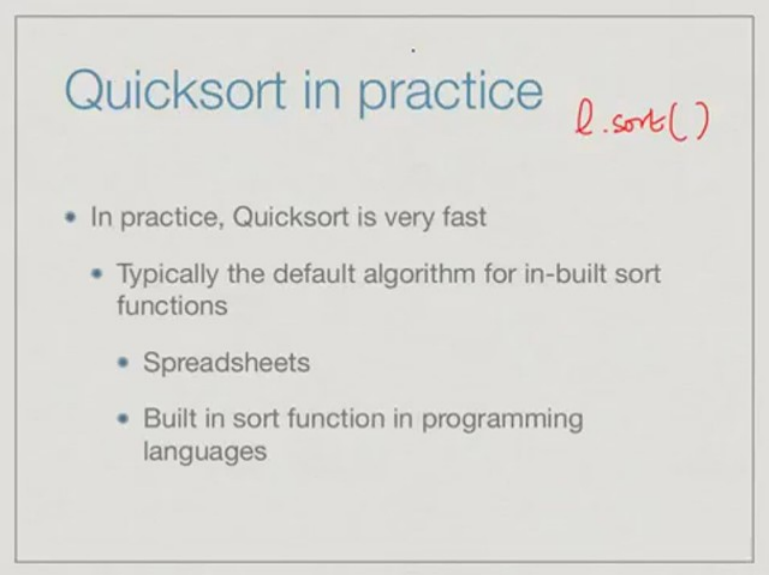


The worst case actually arises because of a fixed choice of the pivot element. So, we choose the first element as the pivot in our algorithm and so in order to construct the worst case input, if we always put the smallest or the largest element at the first position we get a worst case input. This tells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