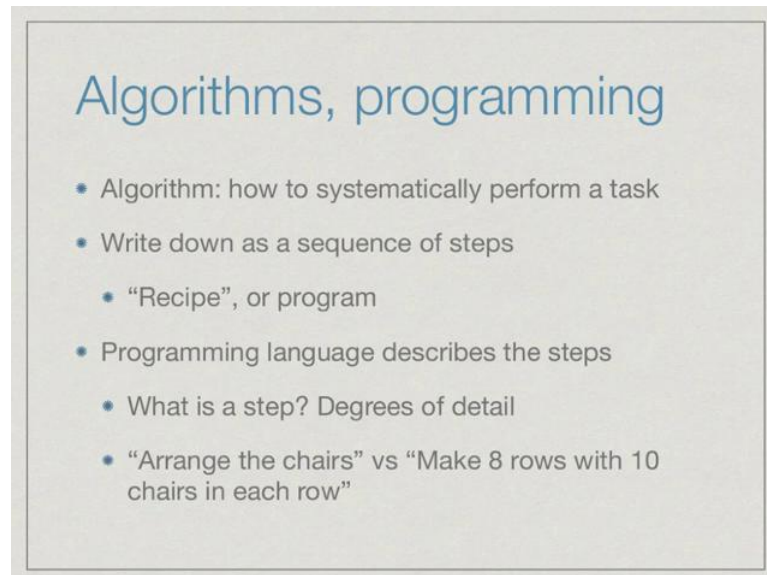**Programming, Data Structures and Algorithms in Python**
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

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.
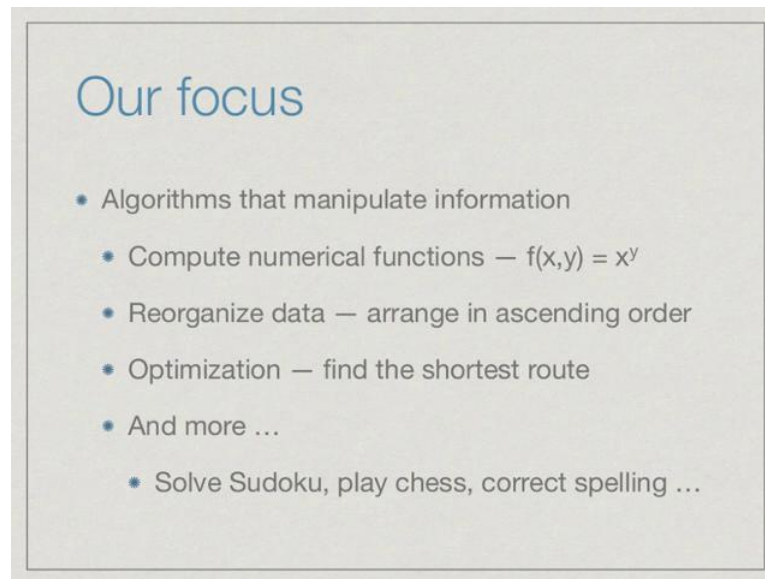
(Refer slide Time: 02:44)

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.
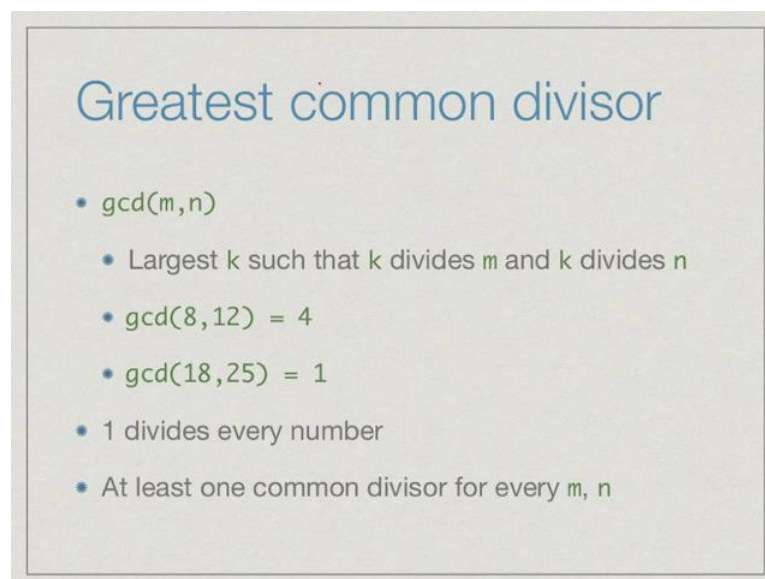
(Refer slide Time: 03:35)

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.
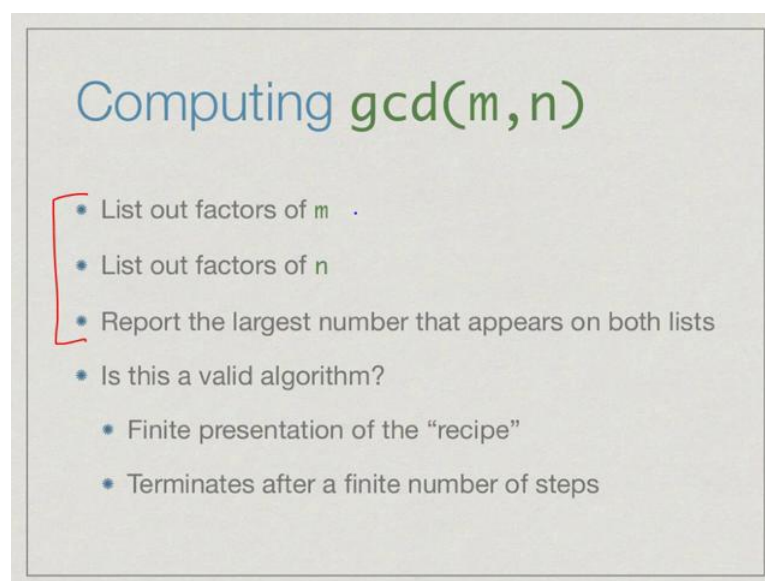
(Refer slide Time: 05:36)

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)

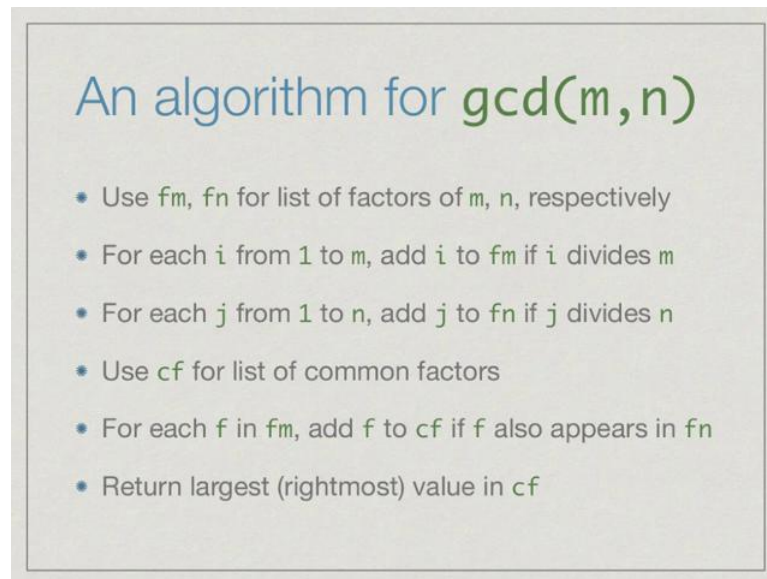If you have to write it down in little more detail, then we could say that we need to notice that we need to remember these lists, right, and then come back to them. So, we need to compute the factors of 14 keep it side we need to write it down somewhere we need to compute the factor of 63 write it down 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.

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.
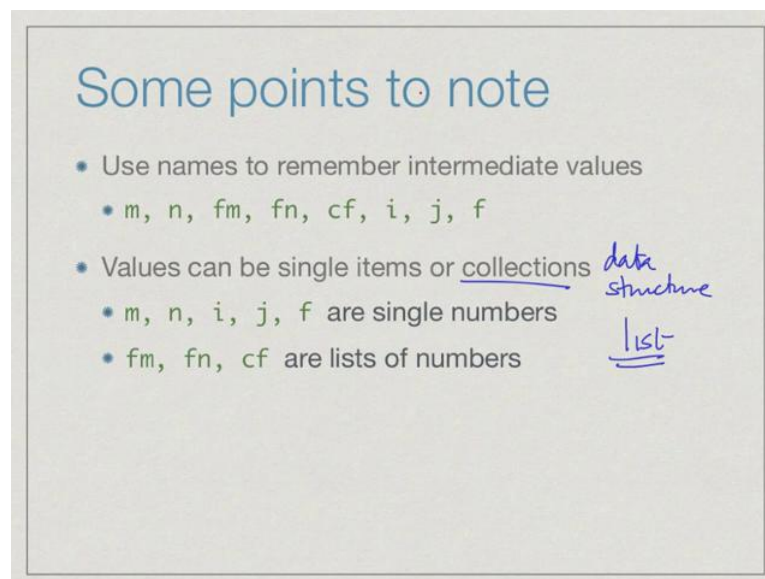
So, the same function append is being use throughout. Just take a list and add a value. Which value? We add the value that we are looking at now provided it satisfies the conditions. So, earlier we were adding provided the divisor was 0 uh the remainder was 0, now we are adding it provided it appears 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.
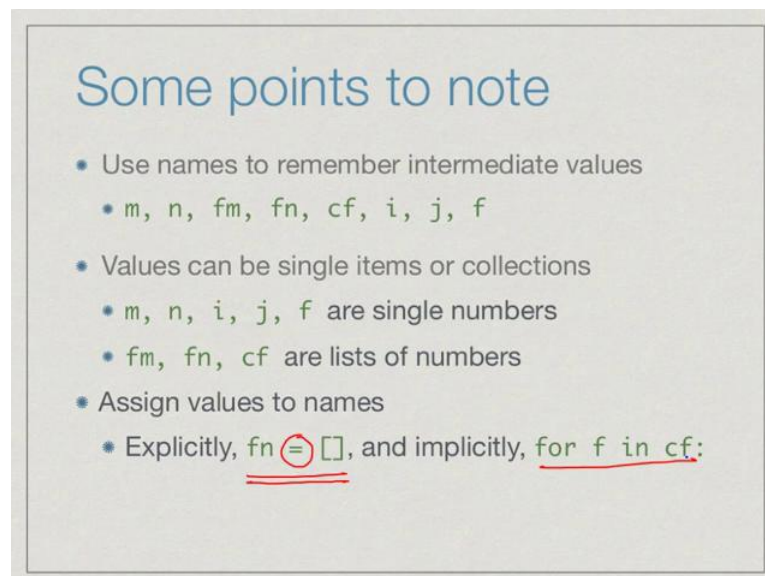
(Refer slide Time: 22:08)

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.
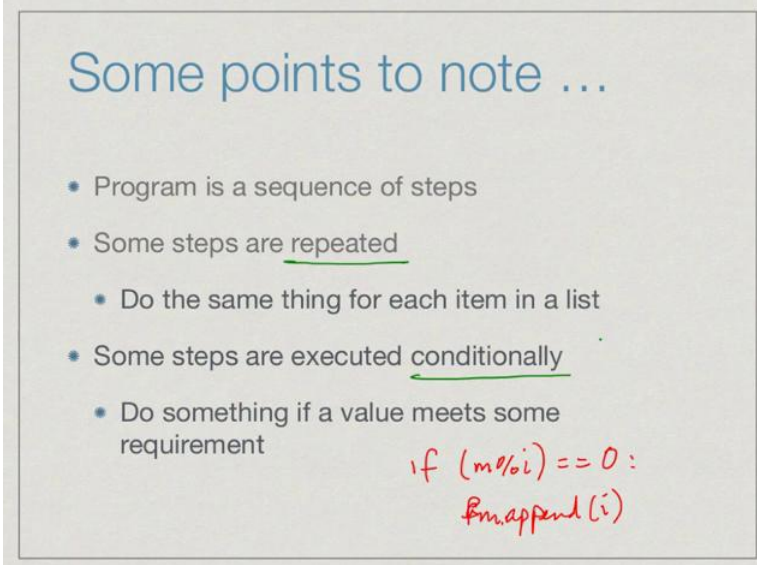
(Refer slide Time: 24:37)

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.