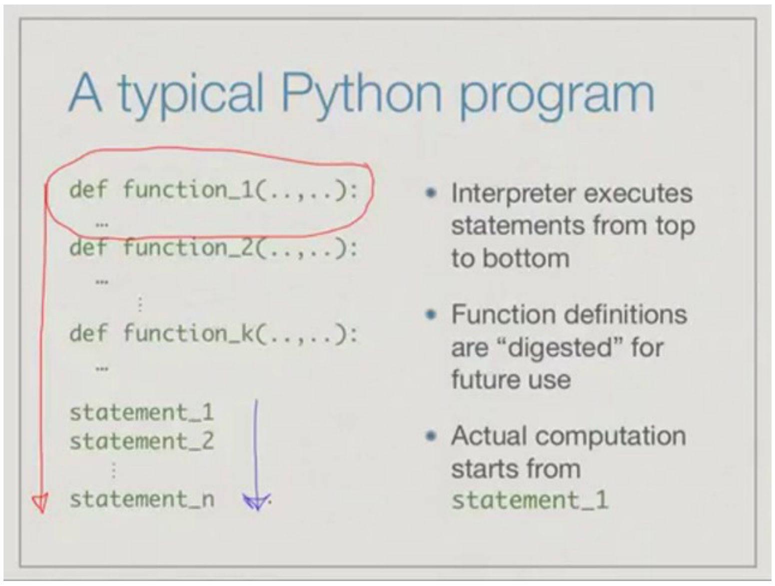
Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Chennai Mathematical Institute,Madras

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)



A typical Python program

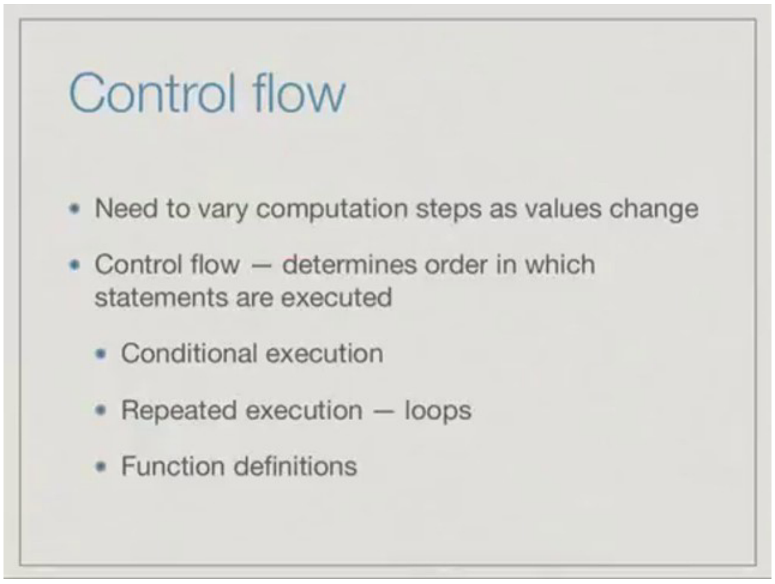
```
def function_1(..,..):
  ...
def function_2(..,..):
  ...
      ⋮
def function_k(..,..):
  ...

statement_1
statement_2
   ⋮
statement_n
```

- Interpreter executes statements from top to bottom
- Function definitions are "digested" for future use
- Actual computation starts from statement_1

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)



Control flow

- Need to vary computation steps as values change
- Control flow — determines order in which statements are executed
  - Conditional execution
  - Repeated execution — loops
  - Function definitions

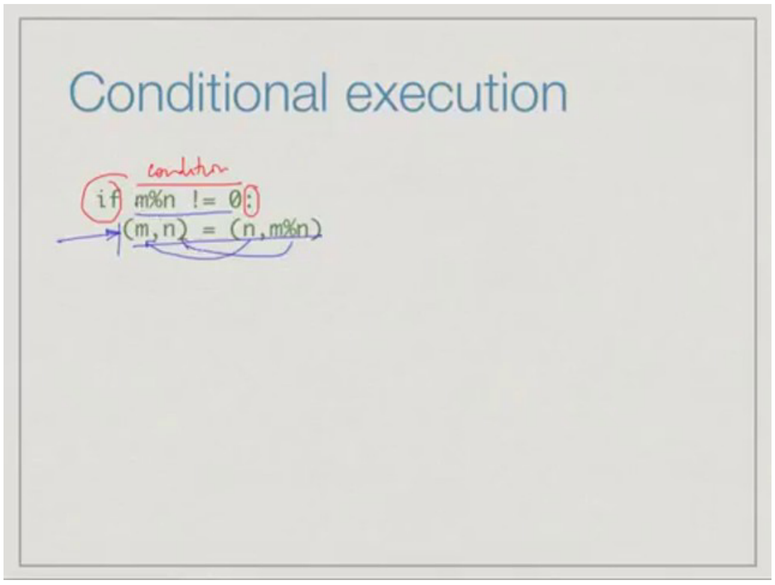
Now in many situations, many realistic situations, we need to vary the next step according to what has happened so far, according to the current values that we see. Let us look at a real life example. Supposing, you are packing your things to leave for the bus stop in the morning, or whether or not you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

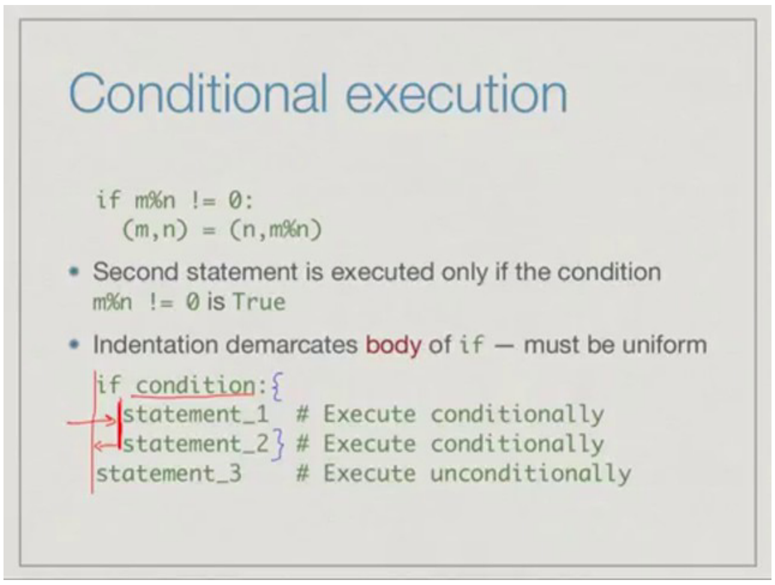


Conditional execution

```
if m%n != 0:
   (m,n) = (n,m%n)
```

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)



Conditional execution

```
if m%n != 0:
  (m,n) = (n,m%n)
```

- Second statement is executed only if the condition `m%n != 0` is `True`
- Indentation demarcates body of `if` — must be uniform

```
if condition:{
  statement_1  # Execute conditionally
  statement_2} # Execute conditionally
statement_3    # Execute unconditionally
```

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.

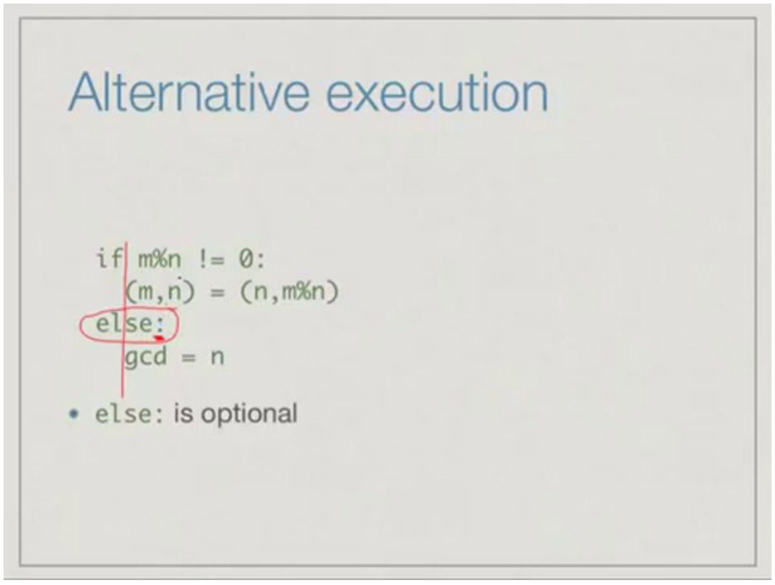
The indentation tells Python that these two statements are conditionally executed depending on the value of this condition. However, statement 3 is not governed by this condition, because it is pushed out to the same level as the if. So, by governing by describing where your text lies, you can decide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



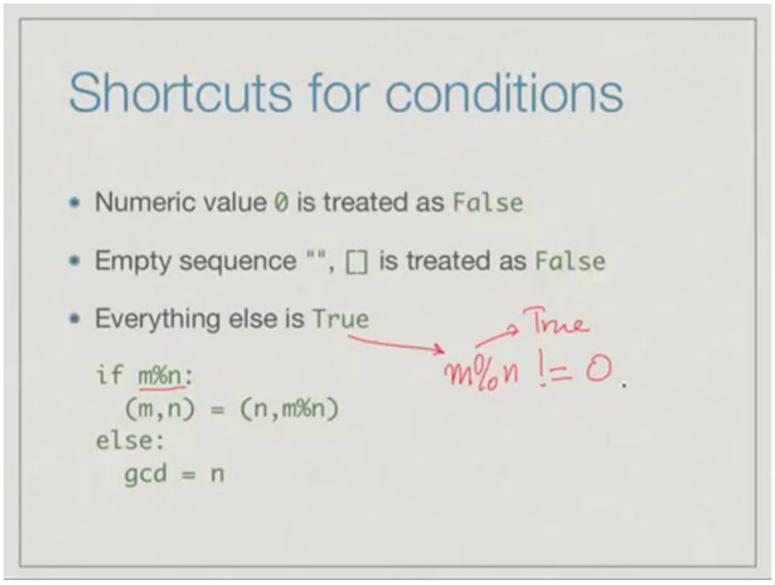
Alternative execution

```
if m%n != 0:
   (m,n) = (n,m%n)
else:
   gcd = n
```

- `else:` is optional

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)



Shortcuts for conditions

- Numeric value 0 is treated as False
- Empty sequence "", [] is treated as False
- Everything else is True

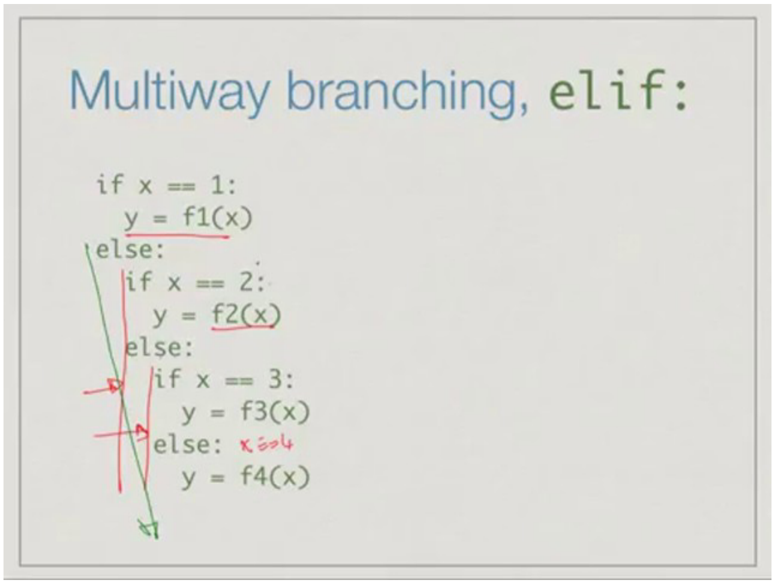
```
if m%n:
  (m,n) = (n,m%n)
else:
  gcd = n
```

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)



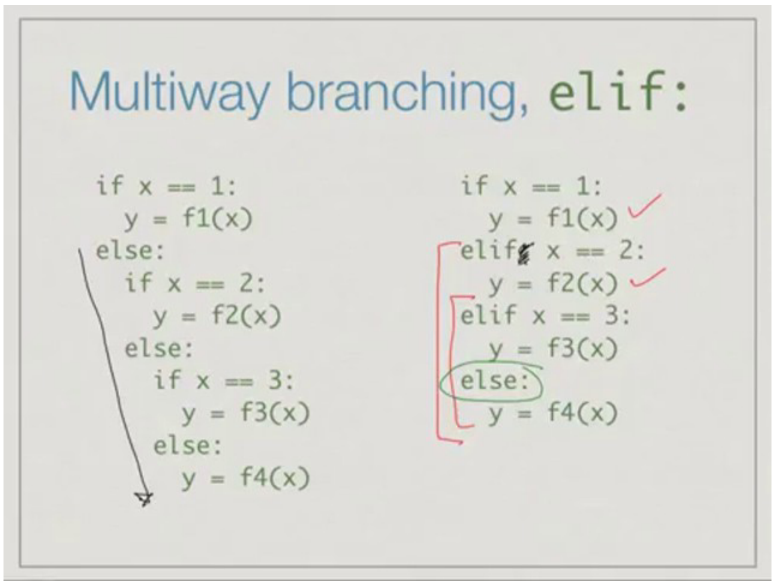
Multiway branching, elif:

```
if x == 1:
  y = f1(x)
else:
  if x == 2:
    y = f2(x)
  else:
    if x == 3:
      y = f3(x)
    else:
      y = f4(x)
```

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)



Multiway branching, elif:

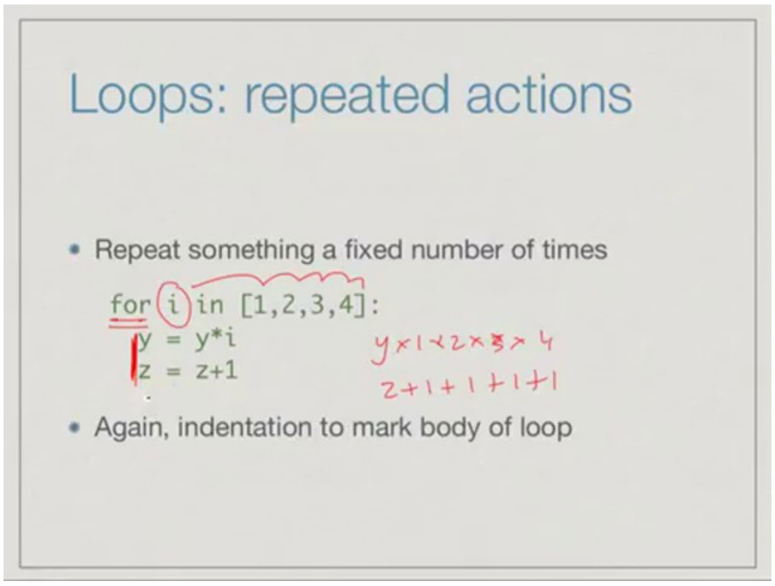
```
if x == 1:
  y = f1(x)
else:
  if x == 2:
    y = f2(x)
  else:
    if x == 3:
      y = f3(x)
    else:
      y = f4(x)
```

```
if x == 1:
  y = f1(x)
elif x == 2:
  y = f2(x)
elif x == 3:
  y = f3(x)
else:
  y = f4(x)
```

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)



Loops: repeated actions

- Repeat something a fixed number of times

```
for i in [1,2,3,4]:
    y = y*i
    z = z+1
```

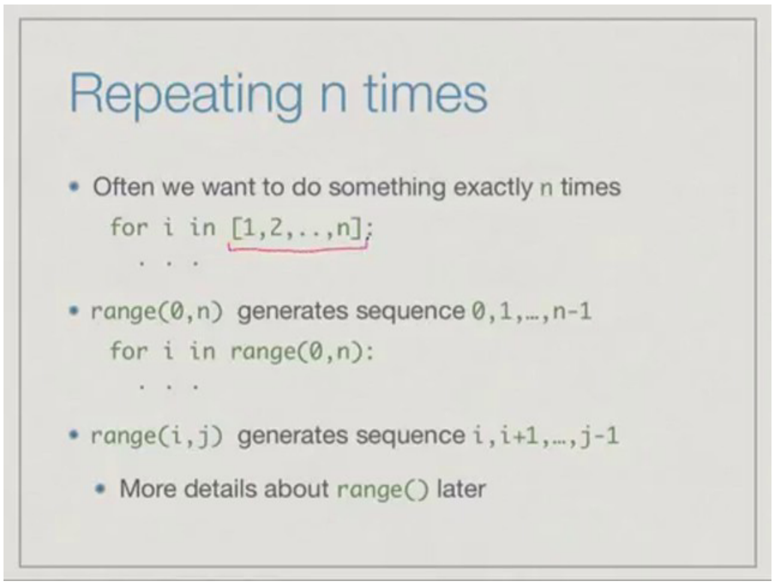
y×1×2×3×4
z+1+1+1+1

- Again, indentation to mark body of loop

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)



Repeating n times

- Often we want to do something exactly n times

```
for i in [1,2,..,n]:
   . . .
```

- range(0,n) generates sequence 0,1,…,n-1

```
for i in range(0,n):
   . . .
```

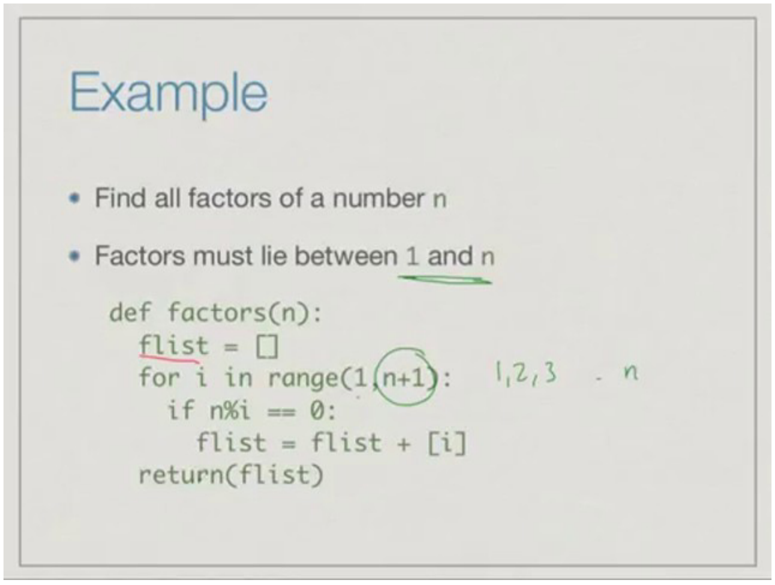
- range(i,j) generates sequence i,i+1,…,j-1
  - More details about range() later

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)



Example

- Find all factors of a number n
- Factors must lie between 1 and n

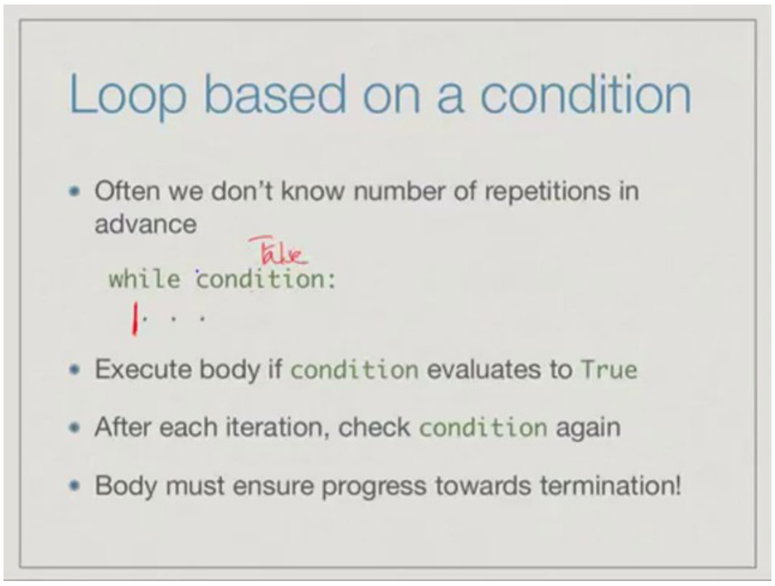
```
def factors(n):
  flist = []
  for i in range(1,n+1):
    if n%i == 0:
      flist = flist + [i]
  return(flist)
```

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)



Loop based on a condition

- Often we don't know number of repetitions in advance

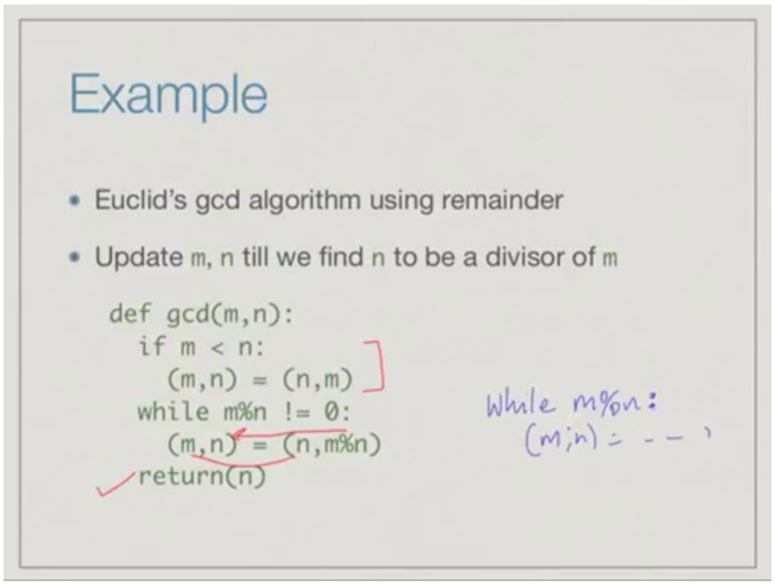
```
while condition:
    . . .
```

- Execute body if condition evaluates to True
- After each iteration, check condition again
- Body must ensure progress towards termination!

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)



Example

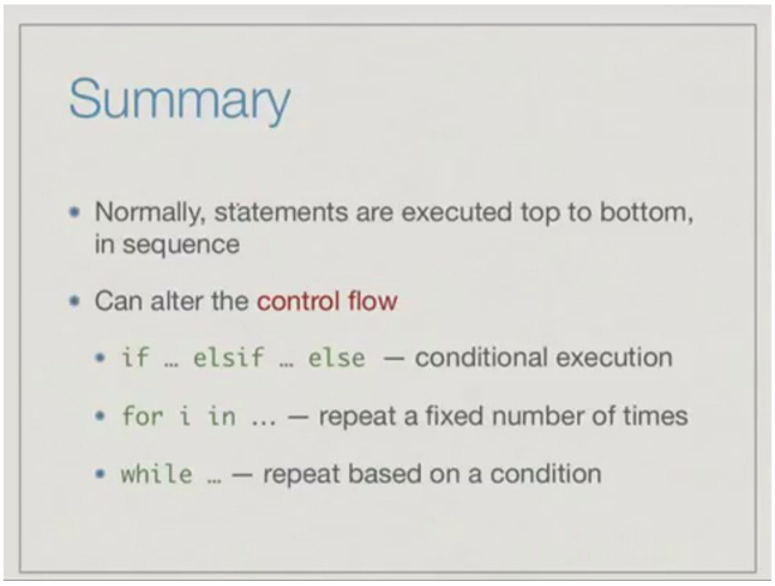
- Euclid's gcd algorithm using remainder
- Update m, n till we find n to be a divisor of m

```
def gcd(m,n):
  if m < n:
    (m,n) = (n,m)
  while m%n != 0:
    (m,n) = (n,m%n)
  return(n)
```

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)



Summary

- Normally, statements are executed top to bottom, in sequence
- Can alter the control flow
  - `if … elsif … else` — conditional execution
  - `for i in …` — repeat a fixed number of times
  - `while …` — repeat based on a condition

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.