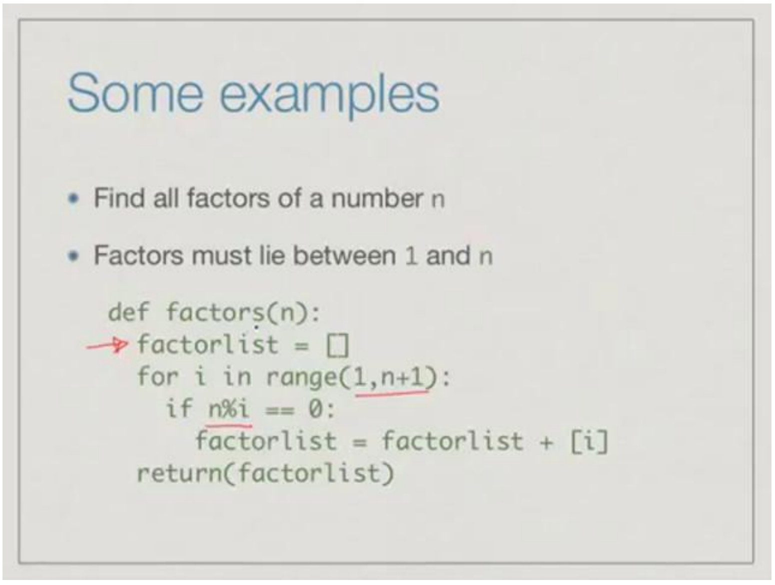
Programming, Data Structures and Algorithms in Python

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)



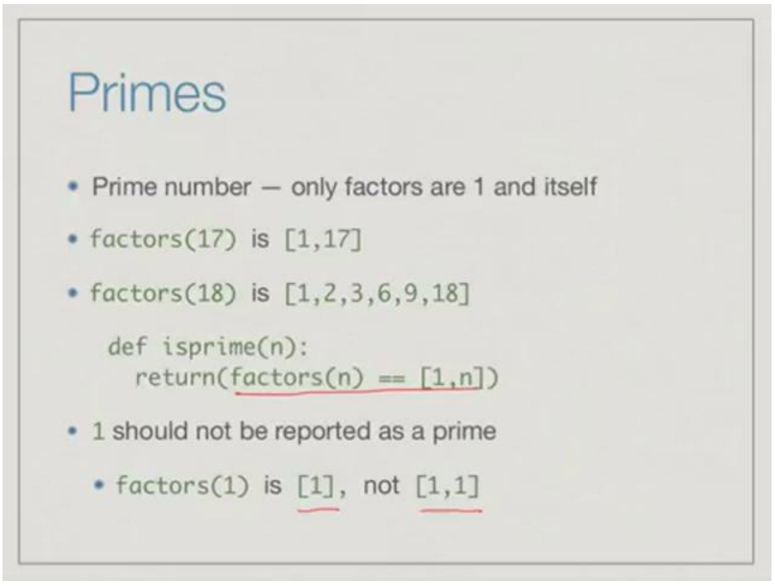
Some examples

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

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

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)



Primes

- Prime number — only factors are 1 and itself
- `factors(17)` is `[1,17]`
- `factors(18)` is `[1,2,3,6,9,18]`

```
def isprime(n):
  return(factors(n) == [1,n])
```

- 1 should not be reported as a prime
  - `factors(1)` is `[1]`, not `[1,1]`

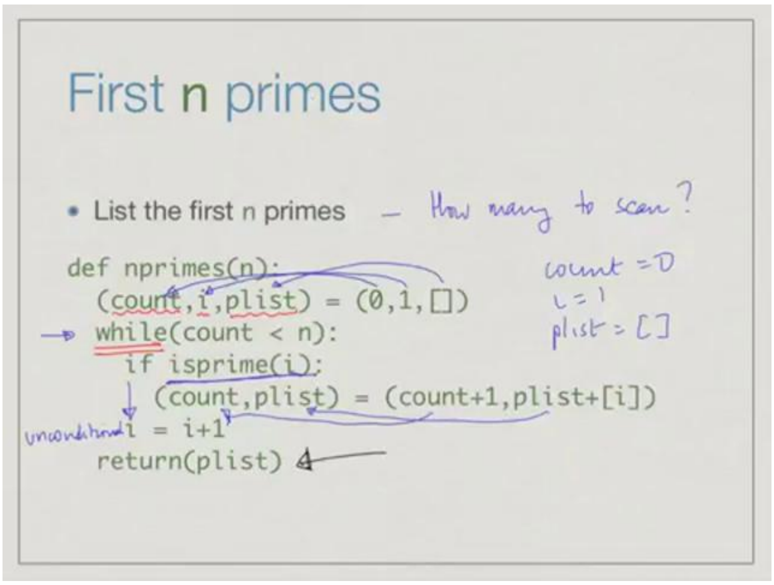
A prime number is one which is divisible by no other number other rather than 1 and itself. In other words, the only factors of n should be 1 and n, if n is a prime. So, 17 for example, which is a prime number has only two factors 1 and 17. Whereas, 18 which is not a prime have many more factors, it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



First n primes

- List the first n primes – How many to scan?

```
def nprimes(n):
   (count,i,plist) = (0,1,[])
   while(count < n):
      if isprime(i):
         (count,plist) = (count+1,plist+[i])
      i = i+1
   return(plist)
```

count = 0
i = 1
plist = []
unconditional

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.

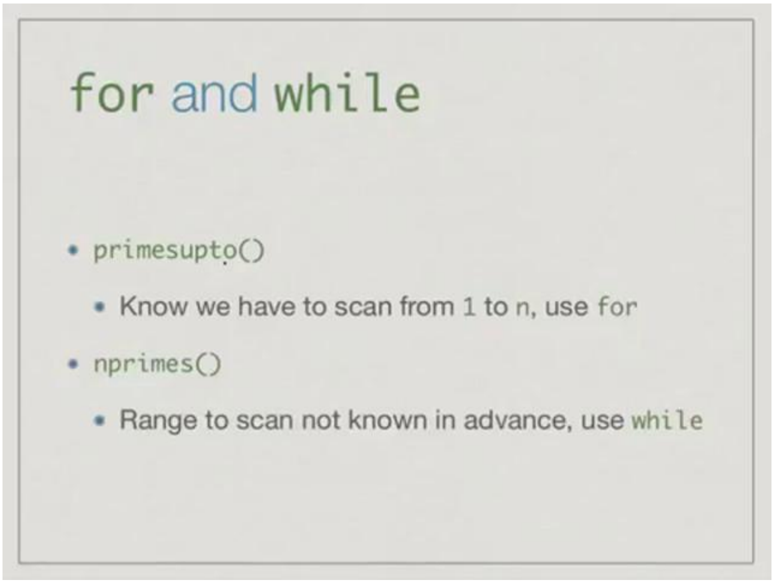
Primes up to n, we knew in advance that we have to check all the numbers from 1 to n, so we could use for. What if we change our requirements saying that we do not want primes up to n, but we want the first n primes. Now, if we want the first n primes, we do not know how many numbers to scan. We do not 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)

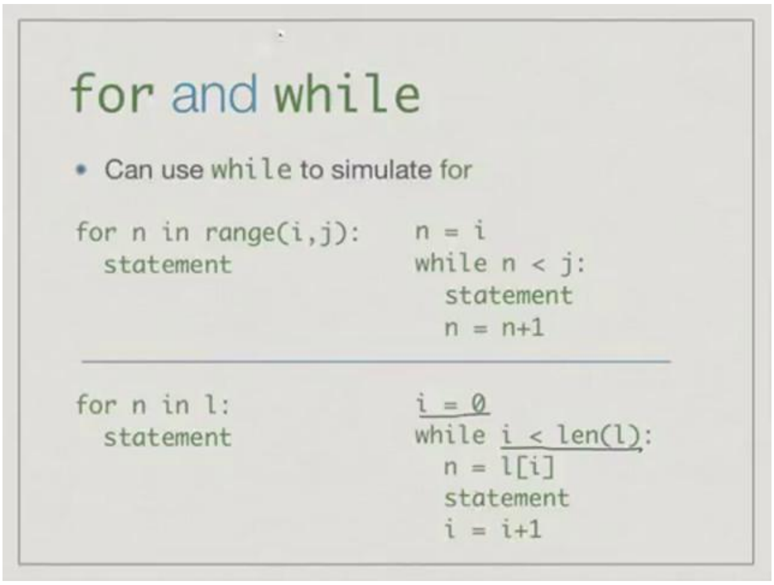


for and while

- primesupto()
  - Know we have to scan from 1 to n, use for
- nprimes()
  - Range to scan not known in advance, use while

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)



for and while

- Can use while to simulate for

```
for n in range(i,j):        n = i
  statement                 while n < j:
                              statement
                              n = n+1
```

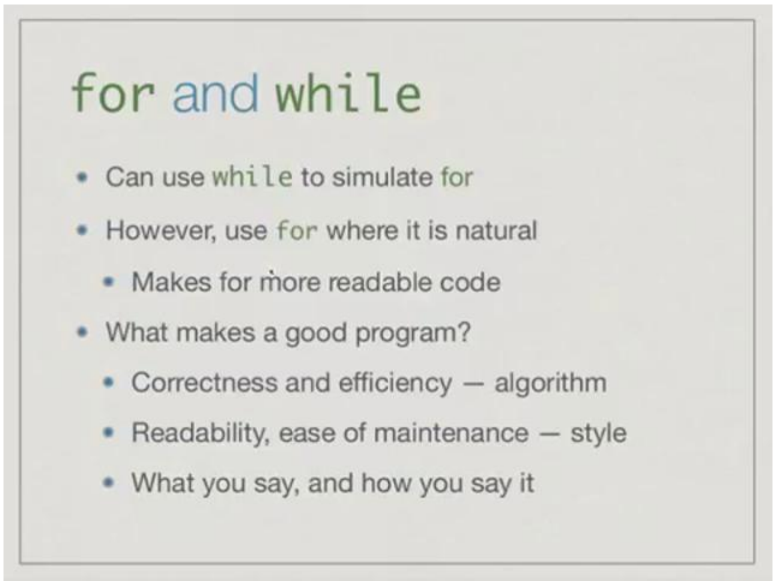
```
for n in l:                 i = 0
  statement                 while i < len(l):
                              n = l[i]
                              statement
                              i = i+1
```

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)



for and while

- Can use while to simulate for
- However, use for where it is natural
  - Makes for more readable code
- What makes a good program?
  - Correctness and efficiency — algorithm
  - Readability, ease of maintenance — style
  - What you say, and how you say it

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.