



IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms using Python
Professor Madhavan Mukund
Python Recap - II

So, let us continue with our recap of python by looking at another example, first we looked at gcd, now let us look at the problem of dealing with prime numbers.

(Refer Slide Time: 0:23)

The image displays two sequential frames from a video lecture. Both frames feature a slide titled "Checking primality" with the IIT Madras logo in the top right corner. The slide content includes:

- A prime number n has exactly two factors, 1 and n
 - Note that 1 is not a prime
- Compute the list of factors of n
- n is a prime if the list of factors is precisely $[1, n]$

Top Frame: The slide shows a Python function `def factors(n):` with the following code:

```
fl = [] # factor list
for i in range(1, n+1):
    if (n%i) == 0:
        fl.append(i)
return(fl)
```

The variable `fl` is circled in red.

Bottom Frame: The slide shows the same content as the top frame, but with an additional function `def prime(n):` and its implementation:

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

Below the code, a handwritten note in red ink states: `factors(1) -> [1] != [1, n]`.

So, we would like to check primality, so remember that a prime number is 1 that is divisible only by 1 and itself, it has no proper divisors other than 1 and itself, so prime number has exactly 2 prime factors, a prime 2 factors 1 and n and it has $(())(0:38)$ of 2, so it is not enough to have only 1 factor so if you look at 1, the number 1, it has only 1 factor namely 1 because it is itself 1.

So, we need two different factors 1 and n so technically speaking 1 is not a prime the smallest prime as you should know is 2. So, one way to check whether a number is prime is to compute the list of factors of that number and then check whether that list of factors contains exactly 1 and n.

So, the list of factors of a number is very similar to the loop that we wrote to compute the common factors between two different numbers. So, in the earlier gcd example that we looked at we computed all the common factors between m and n by running from 1 to the minimum of m and n and for each number they divided both m and n, we appended it to this list of factors.

Here we do not have two numbers we only have one number so we just define this factor list remember like in the common factor list earlier we have to assign it to empty to make sure that python knows it is a list. Now, we run from, so we are looking factors of n, so we run from 1 to n plus 1 because we want n also to be in our loop, so we run try out all numbers 1, 2, 3, 4, up to n whenever we find a number which divides n such that the remainder is 0 when divided by i for each such i we appended.

So, this is very similar to the earlier common factor list except it is the list of factors of a single number n. And now having got this it is a simple matter to write a function which checks if a number is prime you just compute the factors of the number you are trying to check and verify that it is the list 1 comma n.

So, this again tells us that 1 is not a prime because if I say factors of 1, what I am going to get is just the list 1 which is not equal to 1 comma 1. So, that is why this function will correctly compute that the smallest prime is actually 2.

(Refer Slide Time: 2:40)

Counting primes

- List all primes upto m

```
def primesupto(m):  
    pl = [] # prime list  
    for i in range(1,m+1):  
        if prime(i):  
            pl.append(i)  
    return(pl)
```

Madhavan Mukund Python Recap - II

Counting primes

- List all primes upto m
- List the first m primes
 - Multiple simultaneous assignment

```
def firstprimes(m):  
    (count,i,pl) = (0,1,[])  
    while (count < m):  
        if prime(i):  
            (count,pl) = (count+1,pl+[i])  
        i = i+1  
    return(pl)
```

Madhavan Mukund Python Recap - II

So, now that we know how to check a given prime let us see what we can do with this. So, the first thing we might ask is what are all the primes up to 100 say or what are all the primes up to 1000 so we want to list all the primes up to some number m , so again it is a very simple matter we just start off with an empty list of primes and we run a for loop saying for everything from 1 up to m , so remember the range function has to go to m plus 1 so that we capture m , for everything from 1 to m if the number i is a prime and we can use that earlier function to decide this if the earlier function returns true then i gets appended to this list of primes. So, it is a very straightforward thing to compute the list of primes from 1 up to any given number m .

On the other hand, supposing we do not want the list of primes up to a given number m but we want the first m primes. Now, primes are infinite but there are gaps so a priori it is not very clear if I ask you for the first 2000 primes, it is not very clear what is the 2000th prime in value, how many numbers should I search for the first 2000 primes.

So, here instead of using a for loop we will use a while loop and count the primes as we get them and when we reach the number m we will stop. So, we keep a counter to count how many primes we have seen, we keep incrementing i and when we have reached m primes in our counter we stop.

So, one thing to note in this function is that we have used this feature in python called multiple assignment. So, we want to keep track of a count, how many primes we have seen so far and that is initially 0. We want to now use a while loop for this because we do not know how many so in the earlier thing we ran i from 1 to a fixed number m plus 1 but here we do not know how many i 's we have to look at, so we start with i equal to 1 and we explicitly increment it inside the while loop.

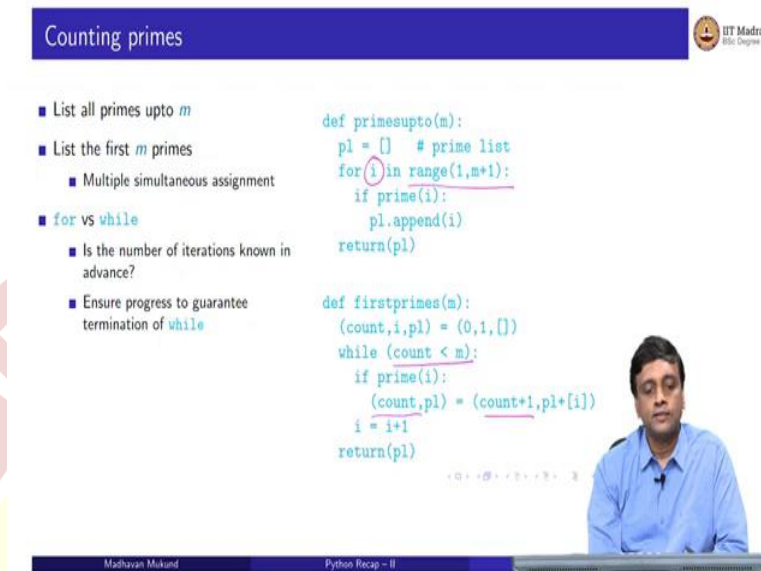
So, this is the second initialization and the third one is we need to of course keep track of the prime list. So, these are three separate assignments count equal to 0, i is equal to 1 and pl is equal to the empty list, all collapsed into one single tuple assignment. So, the first element of the tuple count is assigned to the first element on the right 0, i is assigned to 1, pl is assigned empty.

And now what is our while loop say it says as long as I have not yet found m primes I will check if the current number I am looking at is a prime and if it is a prime then I do two things, I increment the count so again this is a multiple assignment or a multiple update, i count becomes count plus 1 and pl now I use this plus for list which basically concatenates two lists, I already have some primes and I want to add 1 to, i to it earlier I had use the append function here I am using a plus function so this takes a list of primes already given to me, it takes a singleton list with the new prime I have just found and combines it into a longer list pl plus i .

And whatever I have done whether I found a prime or not I must move on to the next i so I have to increment i so that after 1 I will look at 2, after 2 I will look at 3 and so on but i is not determining how when the loop ends, i will keep on incrementing it is only when count reaches the value that I want the threshold that I want that this loop will terminate. So, I

cannot tell you in advance even knowing m it is difficult to predict what value i will reach, I will only know at the end what is the largest prime that I saw.

(Refer Slide Time: 6:21)



Counting primes

- List all primes upto m
- List the first m primes
 - Multiple simultaneous assignment
- for vs while
 - Is the number of iterations known in advance?
 - Ensure progress to guarantee termination of while

```
def primesupto(m):  
    pl = [] # prime list  
    for i in range(1, m+1):  
        if prime(i):  
            pl.append(i)  
    return(pl)  
  
def firstprimes(m):  
    (count, i, pl) = (0, 1, [])  
    while (count < m):  
        if prime(i):  
            (count, pl) = (count+1, pl+[i])  
        i = i+1  
    return(pl)
```

Madhavan Mukund Python Recap - II

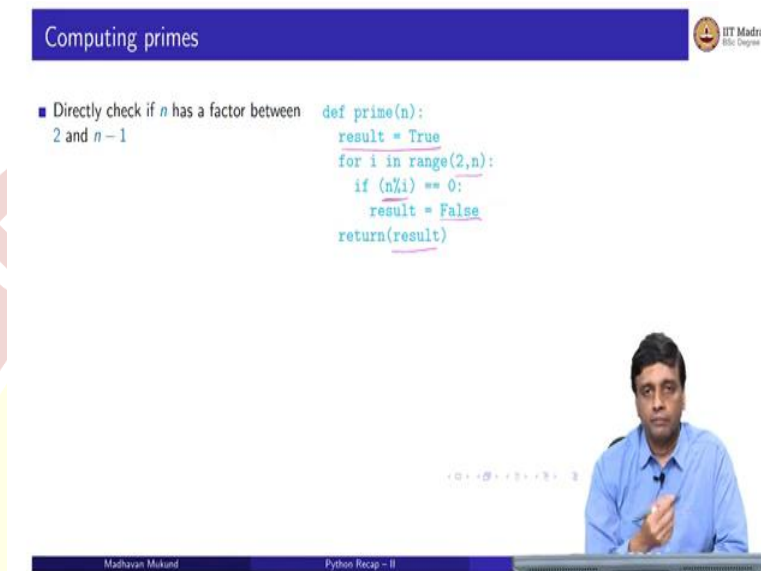
So, this is used so these two examples are basically to illustrate the difference between these two types of loops that python has and almost any other programming language has. So, there is a for loop and a while loop. So, a for loop is something which is predictable, we know in advance how many times we are going to execute it.

So, in the kind of for loop we have used so far, we are running from 1 to m but in general as you know in a python for loop you can run over a sequence or a list, so you can say for i in l . So, it runs over but l is a fixed list you should not be changing that list for example, so that is a very bad and dangerous practice, if you are using a for loop do not change either the index variable or what it is ranging over inside the loop otherwise your program will be very unpredictable. So, the whole purpose of a for loop is to be predictable I know I am going to run this loop a fixed number of times.

On the other hand, when I do not know how many times I am going to execute the loop but the loop will terminate based on some condition that will eventually become true then I use a while loop, like here when I am trying to compute the first m primes but when I am writing a function with a while loop you must make sure that the while loop will actually terminate. Because if you do not ever make that condition true for example if this condition does not become false rather, if this condition remains true forever then the while loop will keep on running indefinitely.

So, you will have a function or a program which does not terminate. So, you have to make sure that you are doing something inside your loop, in this case count equal to count plus 1 which is taking me towards this progress of making sure that the loop will eventually execute, will eventually exit.

(Refer Slide Time: 8:04)



Computing primes

■ Directly check if n has a factor between 2 and $n-1$

```
def prime(n):  
    result = True  
    for i in range(2,n):  
        if (n%i) == 0:  
            result = False  
    return(result)
```

Madhavan Mukund Python Recap - II


So, here is a different strategy for checking primes, we do not actually have to compute the entire list of factors and then check that is 1 comma n . All I have to do is check whether there is any factor between 2 and n minus 1. So, this is again to illustrate a style that we have seen long ago in computational thinking what also you should have seen in other situations. We will assume that a number is a prime.

So, we start off assuming in this name result that the number n we are considering is a prime. Now, we are looking for proper factors so we are not interested in the factor 1, we are not interested in the factor n , so we are going to run from 2 to n minus 1, so the range function goes from 2 to n because if I have the upper limit as n the actual upper limit that we are looking at is n minus 1.

So, in this range from 2 to n if I actually find a prime then I mean if I actually find a factor rather than I know that this is not a prime so my initial assumption gets reversed and I change it to false. At the end of this loop either I did not find a factor in which case my assumption held and it was true or I found a factor and it became false and it is not a prime so whatever it is at the end I return the result. So, this is a more direct way of checking if a number is prime, you just cycle through all the factors and if you find a nontrivial factor between 2 and n minus 1, you switch your rating of this number from true to false.


(Refer Slide Time: 9:34)

Computing primes



- Directly check if n has a factor between 2 and $n-1$
- Terminate check after we find first factor
 - Breaking out of a loop

```
def prime(n):  
    result = True  
    for i in range(2,n):  
        if (n%i) == 0:  
            result = False  
    return(result)  
  
def prime(n):  
    result = True  
    for i in range(2,n):  
        if (n%i) == 0:  
            result = False  
            break # Abort loop  
    return(result)
```




Madhavan Mukund Python Recap - II

Now, of course a nonprime number will have many factors. So, there is no point in looking once we have found the first factor that we are looking for, once we have found 1 factor we are done. So, we can actually terminate this thing as soon as we find a factor and that one way to do that is to use this function called break.

So, what break does is that it breaks out of the loop inside which it is so this break will terminate the for loop the moment I find some i between 2 and n minus 1 which sets this result to be false. So, I do not if I find 2 itself for example supposing the number is very large but it is an even number the moment I know that 2 divides that number I can stop I know it is not a prime.


(Refer Slide Time: 10:22)

Computing primes



- Directly check if n has a factor between 2 and $n-1$
- Terminate check after we find first factor
 - Breaking out of a loop
- Alternatively, use `while`

```
def prime(n):  
    result = True  
    for i in range(2,n):  
        if (n%i) == 0:  
            result = False  
            break # Abort loop  
    return(result)  
  
def prime(n):  
    (result,i) = (True,2)  
    while (result and (i < n)):  
        if (n%i) == 0:  
            result = False  
            i = i+1  
    return(result)
```



Madhavan Mukund Python Recap - II

So, of course breaking out of a loop is sometimes unavoidable, but very often it leads to confusion about what is actually happening in the code, so you should use breaks with caution, it is nicer usually to have this kind of a loop controlled by a while. So, what we can now say is that I will use this result itself to control the loop, so long as I have not contradicted my assumption that the number is a prime I will keep searching and the moment I find a prime that is not a prime I will exit.

So, I assume that result is true and I start with 2. So, I am basically going to simulate this range 2 to the n by a while loop as we did earlier. But now what I do is I check whether I have two things now, I check whether I have violated this so if the result becomes false that is I have found a factor I will exit the loop or if I have exceeded the number of the range of factors I am going to try.

So, if I have cross from n minus 1 to n then I no longer have to continue I can terminate with whatever result I have. So, I need both result to be true, I have still not discovered that it is not a prime and I have not run out of factors to try and if that is the case then I try out the current factor and if the current factor actually divides or the current candidate factor actually divides then I set the result to false so the next loop will terminate and then I increment i so that progress so eventually even if no factors are found i will eventually reach n and in both cases this loop will terminate. So, remember when you write a while it is your job to make sure that the while actually terminates at some point.

(Refer Slide Time: 12:05)

Computing primes

- Directly check if n has a factor between 2 and $n-1$
- Terminate check after we find first factor
 - Breaking out of a loop
- Alternatively, use `while`
- Speeding things up slightly
 - Factors occur in pairs
 - Sufficient to check factors upto \sqrt{n}
 - If n is prime, scan 2, ..., \sqrt{n} instead of 2, ..., $n-1$

```
import math
def prime(n):
    (result, i) = (True, 2)
    while (result and (i < math.sqrt(n))):
        if (n%i) == 0:
            result = False
            i = i+1
    return(result)
```

Handwritten notes on the slide:

- $n = 10^6$
- $\sqrt{n} = 1000$
- $1 = \sqrt{n} \cdot \sqrt{n} = n$

Person in the bottom right corner: Madhavan Mukund, Python Recap - II

So, finally one thing we can do with primes is to speed up this thing a little bit because factors as we all know occur in pairs so if something, if 2 is a factor then n by 2 will be a factor, so if 2 divides 16 then 16 by 2, 8 is also a factor so I do not need to separately check 8, every time I find a factor I find a matching factor.

So, in particular the factors divide as those which are smaller than square root of n and those which are bigger. So, remember square root of n is a number, so that square root of n is itself its pair, so every number, every factor smaller than square root of n will have a factor bigger than square root of n .

Of course in general square root of n will not be an integer unless n is a perfect square but it is enough to look from 1 to square root of n because all the factors beyond square root of n are paired with all the factors before square root of n . So, if n is a prime I just need to check for factors from 2 to square root of n rather than 2 to n minus 1.

So, in order to use square root I need to import this math function so I import math and now the difference between the previous one and this is that instead of checking earlier I was checking i less than n , now I am stopping at a smaller number i less than square root of n everything else is the same. So this marginally improves it I mean in a sense it drastically improves because square root of n will be smaller than n so if say for example n is say 1 million then square root of n will be 1 thousand, so there will be a drastic improvement in that sense. But actually we would like something even more efficient than this but that is beyond the scope of this course.

(Refer Slide Time: 13:51)

The image shows two screenshots of a presentation slide titled "Properties of primes" from IIT Madras. The slide content is as follows:

- There are infinitely many primes
- How are they distributed?
- Twin primes: $p, p+2$

Handwritten notes on the top screenshot:

- 2, 3, 5, 7
- 41, 43, 47

Handwritten notes on the bottom screenshot:

- 17 19
- 5 7
- 41 43
- A circle containing $2^k - 1$ and $2^k + 1$

So, as a final exercise using primes let us look at some properties of primes. So, one of the things that is a very classical result about prime numbers is that there are infinitely many of them, there is no such thing as the largest prime, there are any number of proofs and if you do not know some proofs I encourage you to find out because there are some very simple proofs of this.

Now, primes become larger and larger so one of the natural questions that people have asked is how are they distributed, how are the gaps between the primes. Obviously as you get larger and larger you would some, you can notice even if you look at small numbers like you know numbers between 1 and 1000 between say 1 and 10 you have prime numbers like 2, 3, 5, 7 so they are pretty close.

But if you start going up the ladder then by the time you reach say the 40s you have 41, 43 then you have a big gap like 47 and so on. So, the gaps become larger between the primes as the numbers become bigger so how are the primes distributed. So, you would be inclined to think that as the numbers become larger the gaps will keep on increasing.

So, a particular interesting difference is two. So, we say that two primes are twin primes if they are difference of two. And why will there be a difference of two? Typically, they are two odd numbers and they are usually of the form 2 to the k minus 1 or 2 to the k plus 1 this is a typical type of format of two prime numbers.

So, it is not always the case but there are many interesting primes which have this kind of a distribution. For example, 15 and 17, no 15 and 17 are not prime so let us not worry about this but there are these twin primes and some of them have this format but twin primes do exist like 17 and 19 are twin primes because they are 2 apart, 5 and 7 are twin primes because they are 2 apart and so on 41 and 43 are also twin primes.

(Refer Slide Time: 16:01)

Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: $p, p+2$
- Twin prime conjecture
There are infinitely many twin primes
- Compute the differences between primes
- Use a dictionary
- Start checking from 3, since 2 is the smallest prime

key \rightarrow Value
definition \rightarrow frequency

Madhavan Mukund Python Recap - II

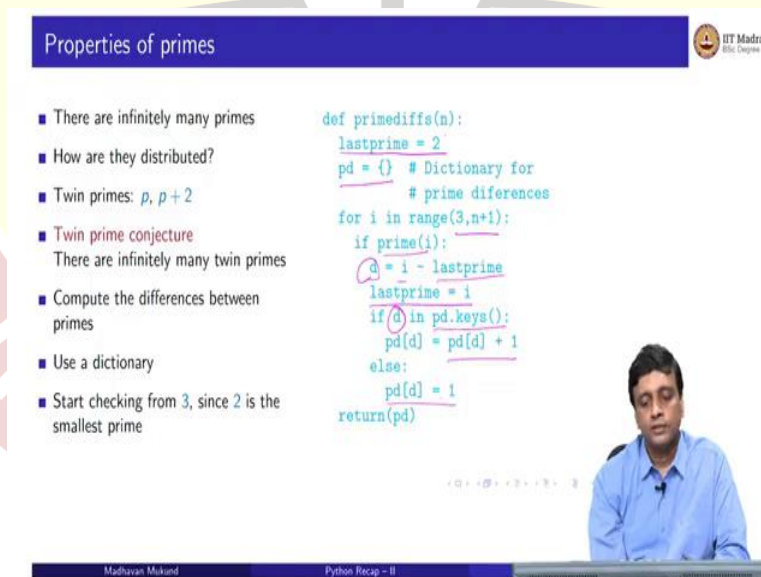
So, the twin prime conjecture is one of the questions which is yet not resolved in number theory is are there infinitely many twin primes. So, as we go into the larger and larger numbers and look at all the primes will we still find primes of the form p and p plus 2. So, we will of course not be able to solve the twin prime conjecture using python, but we can at least ask the following question which is if I give you a range of numbers like we saw earlier how to compute the primes up to a number m .

So, instead of just computing the primes up to a number m supposing I ask you what are the differences that I observe between the primes between 1 and m . Now, you want to keep track of just the differences, we are not interested in the primes themselves, we are interested in the difference between each prime and the previous prime.

So, this brings us to another very common data structure that is available in python. So, we have seen lists which have you have used so far to compute the list of prime numbers but now we do not want a list, we want to keep track of how many times each difference occurs. So, we want to keep track of a certain numbers and for each of them we want to keep a counter.

So, this is what we call a dictionary. So, we have a dictionary which will take a key and associate with it a value, so the key will typically be in our case a difference and the value is going to be the frequency of that difference. So, if I see that the number of twin primes is actually infinite then the number of times I see a difference of 2 we will keep on growing, but if maybe for some other number like 17 maybe or 18 rather the number will be finite. But what we want to do is generally compute for a finite range between 1 and m say what are the differences that I get and how many times do I see each of these differences.

(Refer Slide Time: 17:48)



Properties of primes

- There are infinitely many primes
- How are they distributed?
- Twin primes: $p, p+2$
- Twin prime conjecture
There are infinitely many twin primes
- Compute the differences between primes
- Use a dictionary
- Start checking from 3, since 2 is the smallest prime

```
def primediffs(n):  
    lastprime = 2  
    pd = {} # Dictionary for  
           # prime differences  
    for i in range(3, n+1):  
        if prime(i):  
            d = i - lastprime  
            lastprime = i  
            if d in pd.keys():  
                pd[d] = pd[d] + 1  
            else:  
                pd[d] = 1  
    return(pd)
```

Madhavan Mukund Python Recap - II

So, the first difference will be between the second number and the because if I start with 2 as a smallest prime it has no difference with the previous prime because there is no previous prime. So, for this prime difference function it makes sense to start from 3. So, the difference between 3 and 2 is the first difference that I observe and then the difference between 5 and 3 will be the next difference and so on.

So, I will keep track of the last prime number that I have seen and I will initialize this to 2. So, I assume that I have seen 2 as the smallest prime because 2 is always the smallest prime. And now I have a dictionary which is going to keep track of the frequency of each difference. So, the keys of this dictionary are going to be the differences, it is a difference, it is the number which represents a difference between 1 prime and the next prime and the value is going to be how many times I see 2 primes with that difference.

So, I start with 3 and I go until n so as usual range goes to n plus 1. For each prime number that I see I compute this difference what is the difference between the prime number I just found and the last prime number which I had seen before and then I reset this number to be the last prime because next time this is going to be the last prime.

Now, having done this now I have to update my dictionary. So, I check whether I have seen this difference before. So, does this difference appear in the keys of this dictionary or not, if this appears in the keys of dictionary I have already seen this difference before so I just increment the current value of that difference by 1.

If I have not seen this difference before then I create a new entry for this dictionary. See these are all very familiar patterns which you have seen before right from computational thinking onwards but it is important to just recognize how to use these things, so we are just reviewing this. So, this is a kind of an introduction to various things, so we have seen for loops, while loops and we have seen dictionaries in the context of computing prime numbers.