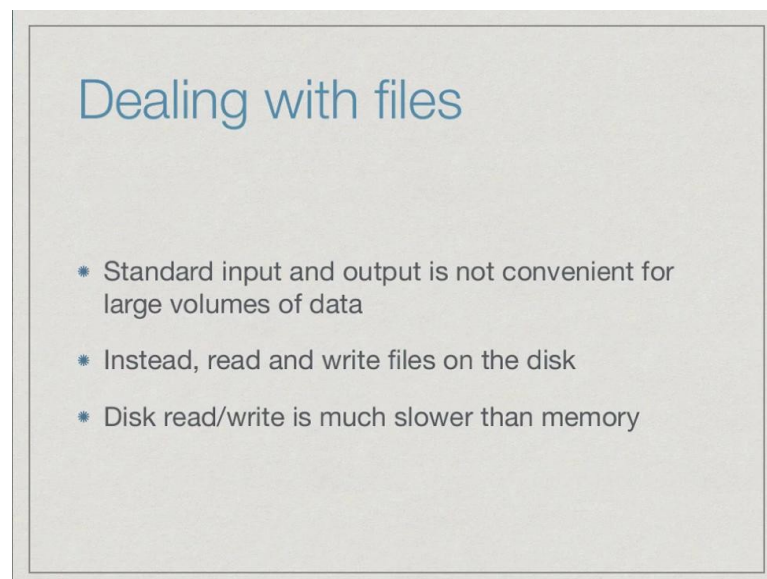**Programming, Data Structures and Algorithms in Python**
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 05
Lecture - 03
**Handling files**

(Refer Slide Time: 00:02)

In the last lecture we saw how to use the input and print statements to collect input from the standard input that is the keyboard, and to display values on the screen using print.

Now, this is useful for small quantities of data, but we want to read and write large quantities of data. It is impractical to type them by hand or to see them as a scroll pass in this screen. So, for large data we are forced to deal with files which reside on the disk. So, we have to read a large volume of data which is already written on a file in the disk and the output we compute is typically return back into another file on the disk. Now, one thing to keep in mind when dealing with disks is that disk read and write is very much slower than memory read and write.
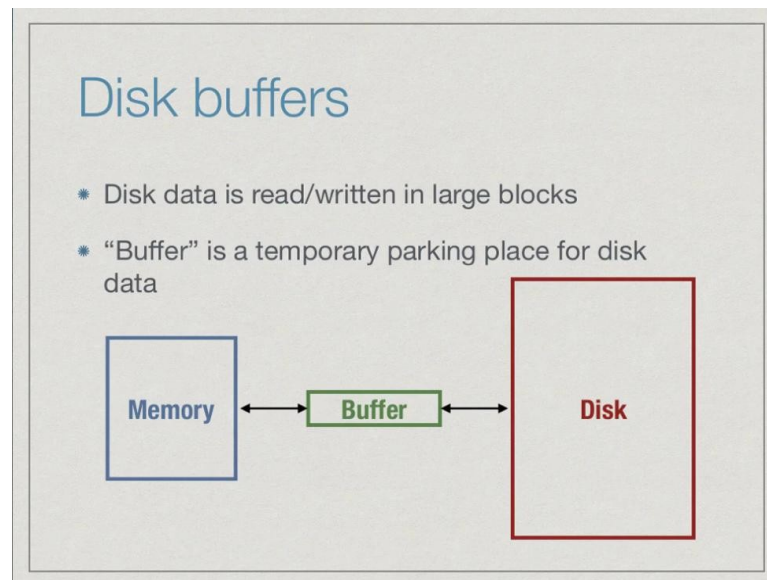
To get around this most systems will read and write data in large blocks. Imagine that you have a large storage facility in which you store things in big cartons. Now, when you go and fetch something you bring a carton at a time even if you are only looking for, say one book in that carton, you do not go and fetch one book out of the carton from the storage facility, you bring the whole carton and then when you want put things back again you assemble them in a carton and put them back.

In the same way, the way that data flows back and forth between memory and disk is in chunks called blocks. So, even if you want to read only one value or only one line it will actually a fetch large volume of data from the disk and store it in what is called a buffer and then you read whatever you need from the buffer. Similarly, when you want to write to the disk you assemble your data in the buffer when the buffer is enough quantity to be written on the disk then one chunk of data or block is written back on the disk.
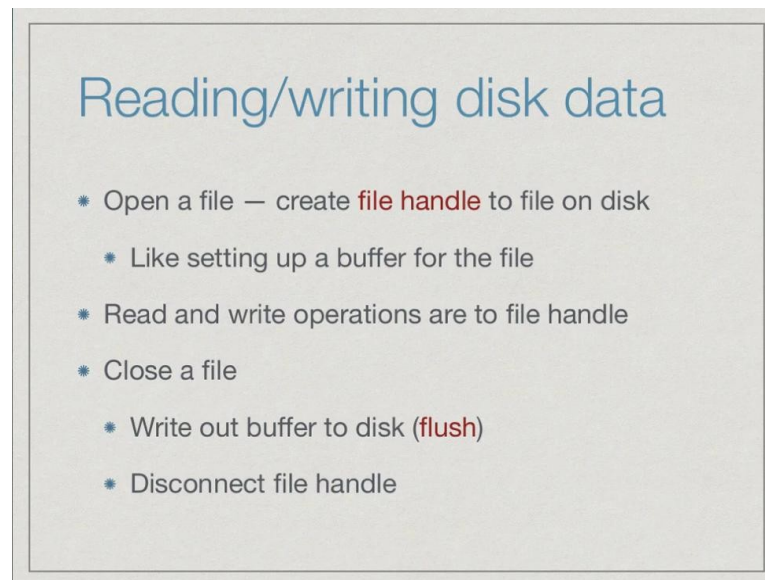
When we read and write from a disk the first thing we need to do is connect to this buffer. This is called opening a file. So, when we open a file we create something called a file handle and you can imagine that this is like getting access to a buffer from which data from that file can read into memory or written back.

Now, having opened this file handle everything we do with the file is actually done with respect to this file handle. So, we do not directly try to read and write from the disk, instead we read and write from the buffer that we have opened using this file handle and finally, when we are done with our processing we need to make sure that all the data that we have written goes back. So, this is done by closing the file.

So, closing the file has two effects; the first effect is to make sure that all changes that we intended to make to the file. Any data we want to write to the file is actually taken out to the buffer and put on to the disk and this technically called flushing the buffer. So, closing a file flushes the output buffer make sure that all rights go back to the file and do not get lost and the second thing it does is that it in some sense makes this buffer go away. So, it disconnects the file handle that we just set up. Now, this file is no longer connected to us if we want to read or write it again we have to again open it.
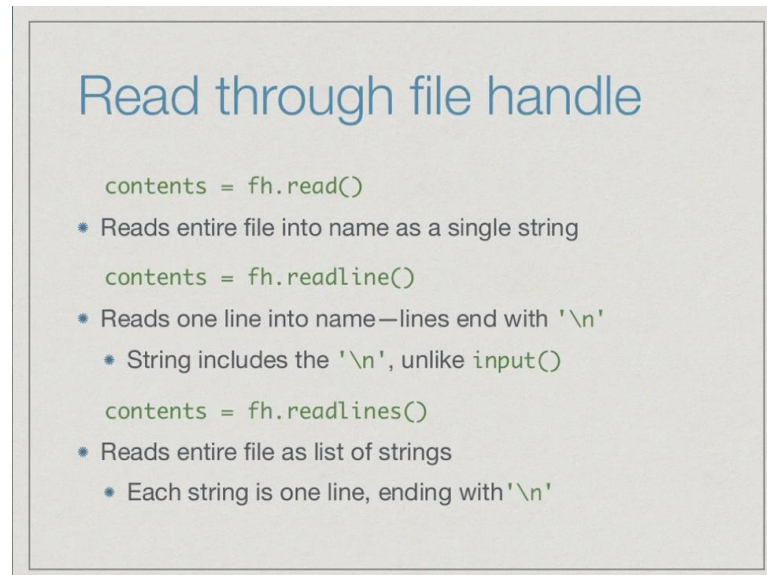
The command to open a file is just 'open'. The first argument that you give open is the actual file name on your disk. Now, this will depend a little bit on what system you are using, but usually it has a first part and an extension. This commands, for instance, to open the file gcd dot py. Now implicitly, if you just give a file name it will look for it in the current folder or directory where you running the script. So, you can give a file name which belongs to the different part of your directory hierarchy by giving a path and how you describe the path will depend on whether you are working on Windows or Unix, what operating system you are using.

Now, you see there is a second argument there, which is letter 'r'. This tells us how we want to open the file. So, you can imagine that if you are making changes to a file by both reading it and writing it, this can create confusion. So, what we have to do is decide in advance whether we are going to read from a file or write to it, we cannot do both. There is no way we can simultaneously read from a file and modify it while it is open. So, read is signified by 'r'.

Now, write comes in two flavors, we might want to create a new file from scratch. In this case we use a letter 'w'. So, 'w' stands for write out a new file, we have to be bit careful about this because if we write out a file which already exits then opening it with more 'w' will just overwrite the contents that we already had. The other thing which might be useful to do is to take a file that already exists and add something to it. This is called

append. So, we have two writing modes; 'w' for write and 'a' for append. What append will do is it will take a file which already exists and add the new stuff the writing at the end of the file.

(Refer Slide Time: 05:02)

Once we have a file open, let us see how to read. So, we invoke the read command through the file handle. This is like some of the other function that you saw with strings and so on, where we attach the function to the object. So, fh is the file handle we opened, we want to read from it. So, we say fh dot read, what fh dot read does is it swallows the entire contents the file as a single string and returns it and then we can assign it to any name, here we use the name contents. So, contents is now assigned the entire data which is in the file handle pointed by fh in one string.

Now, we can also consume a file, we are typically dealing with text files. So, text file usually consists of lines; think of python code, for example, we have lines after lines after lines. A natural unit is a bunch of texts which is ended with new line character. If you remember this is what the input command does, the input command waits for you type something and then you press return which is a new line and whatever you type up to the return is then transmitted by input as a string to the name that you assigned to the input.

So, readline is like that, but the difference between the readline and input is that, when you read a line you get the last new line character along with the input string. When you

say input you only get the characters which come before the last new line the new line is not included, but in readline you do get the new line character.
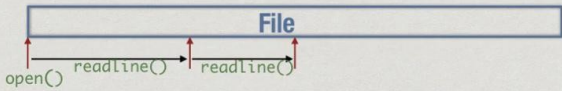
So, you have to remember that you have the extra character floating around at the end of your string. So, this is conventionally the noted backslash n. The backslash n is a notation which denotes a single character even though looks two characters. This is supposed to be the new line character. Now, the actual new line character differs on operating systems from one to the other, but in python if we use backslash n and it will correctly translated in all the systems that you are using.

The third way that you can read from a file is to read all the lines one by one into a list of strings. So, instead of readline, if I say readlines then it reads the entire the files as a list of srings. Each string is one item in the list and remember again each of these lines has the backslash n included. So, read, readline and readlines, none of them will actually remove the backslash n. They will remain faithfully as part of your input.

In other words, if you are going to transfer this from one file to another, you do not want to worry reinserting the backslash n because this is already there. So, you can use this input output directly, but on the other hand, if you want to do some manipulation of the string then you must remember this backslash n is there and you must deal with it appropriately.

(Refer Slide Time: 07:49)

Reading files

* Reading is a sequential operation
    * When file is opened, point to position 0, the start
    * Each successive `readline()` moves forward
* `fh.seek(n)` — moves pointer to position `n`
* `block = fh.read(12)` — read a fixed number of characters

Reading files is inherently a sequential operation. Now, of course, if we use the basic command read, it reads the entire content. So obviously, it reads from beginning to the end, but if you are reading one line at a time then the way it works is that when we open the file we are initially at the beginning of the file. So, you can imagine a pointer like this red arrow which tells us where we are going to read next. So, initially when we open we are going to read from the beginning, now each readline takes us forward. If I do a readline at this point it will take me up to the next backslash n.
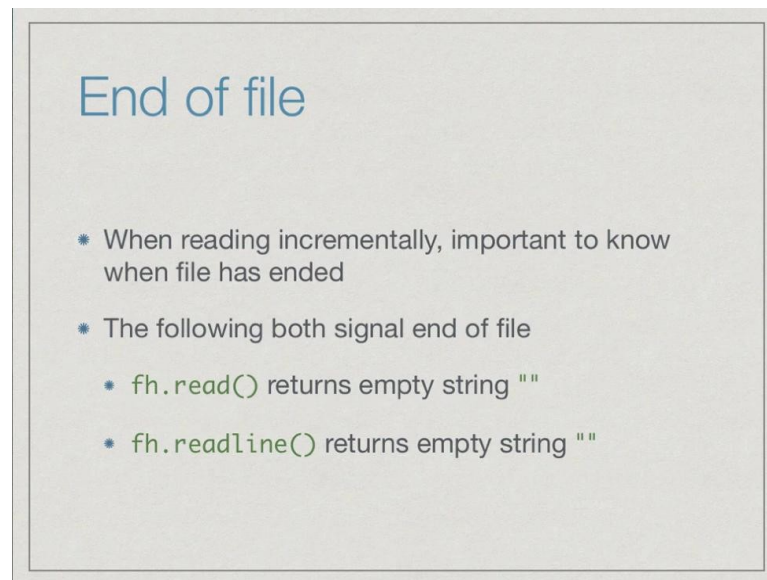
Remember a line is a quantity which is delimited by backslash n. So, we could have a line which has 100 characters, next line could have 3 characters and so on. It is from one backslash n to the next is what a line, so this is not a fixed link. So, we will move forward reading one character at a time until we have backslash n, then everything up to the backslash n will be returned as the effect to a string return by the readline and pointer move to the next character. Now, we do another readline possibly of different line again the point to move forward. So, in this way we go from beginning to the end.

In case we want to actually divert from the strategy there is a command seek, which takes a position, an integer n, and moves directly to the position n regardless of where you are. This is one way to move back or to jump around in a file other than by reading consecutively line by line.

Finally, we can modify the read statement to not to read the entire file, but to read a fix number of characters. Now, this may be useful if your character actually your file actually consists of fix blocks of data. So, you might have say, for example, pan numbers which are typically 10 characters long and you might have just stored them as one long sequence of text without any new lines knowing that every pan number is 10 characters. So, if we say fh dot read 10, it will read the next pan number and keep going and this will save you some space in the long run. So, there are situation where you might exploit this where you read a fix number of characters.

(Refer Slide Time: 09:56)

End of file

* When reading incrementally, important to know when file has ended

* The following both signal end of file
  * `fh.read()` returns empty string `""`
  * `fh.readline()` returns empty string `""`

When we are reading a file incrementally, it is useful to know when the file is over because we may not know in advance how long files is or how many lines of file is. So, if you are reading a file line by line then we may want to know when the file has ended. So, there are two situations where we will know this. So, one is if we try to read using the read command and we get nothing back, we get an empty string that means the file is over, we have reached end of file.

Similarly, if we try to read a line and we get empty string it means we reached the end of file. So, read or readline if they return empty string its means that we have reached the end of the file. Remember, we are going sequential from beginning to the end. So, we reached the end of the file and there is nothing further to read in this file.

Having read from a file then the other thing that we would like to do is to write to a file. So, here is how you write to a file just like you have read a command you have a write command, but now unlike read which implicitly takes something from the file and gives to you, here you have to provide it something to put in the file. So, write takes an argument which is a string. When you say, write s says take the string s and write it to a file.
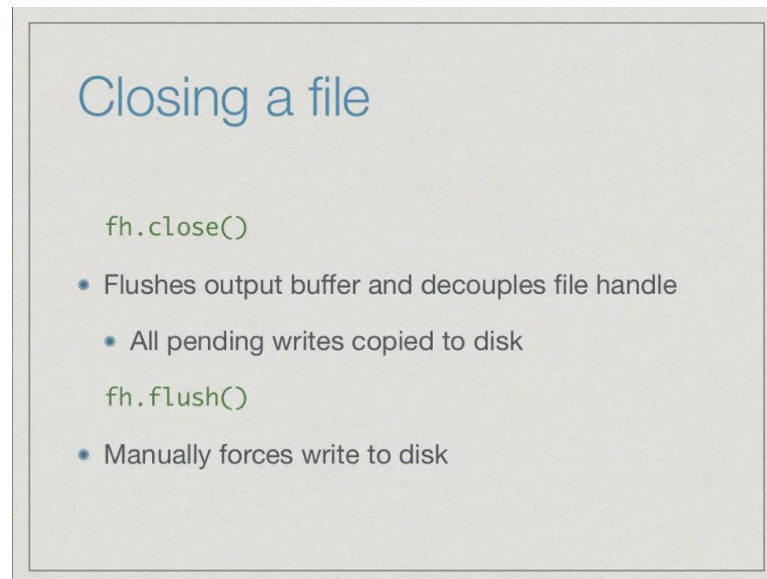
Now, there are two things; one is this s may or may not have a backslash n, it may have more than one backslash n. So, is nothing tells you this is one line part of a line more than a line you have to write s according to the way you want it to be written on the file, if you want it to be in one line you should make sure it ends with a backslash n.

And this write actually returns the number of characters written. Now, this may seem like a strange thing to do, why should it tell you because you know from the length of s what is number of character is written, but this is useful if, for instance, the disk is full. If you try to write a long string and find out only part of the string was written and this is a indication that there was a problem with the write. So, it is useful sometimes to know how many characters actually got written out of the characters that tried to write.

The other thing which writes in bulk to a file is called writelines. So, this takes list of strings and writes them one by one into the file. Now though it says write lines these may not actually be lines. So, its bit misleading the name if you want to them in lines you

must make sure that you have each of them terminated by backslash n. If they are not then they will just cascade to form a long line thing. So, though it says writelines it should be more like write a list of strings, that should, that is a more appropriate name for this function, it just takes a list of strings and writes it to the file pointed to by the file handle.

(Refer Slide Time: 12:40)

And finally, as we said once we are done with a file, we have to close it and make sure the buffers that are associated with the file, especially if you are writing to a file that they are flushed. So, fh dot close, will close the file handle fh and all pending writes at this point are copied out to the disk. It also now means that fh is no longer associated with the file we are dealing with. So, after this if we try to invoke operation on fh it is like having undefined name in python.

Now, sometimes there are situations where we might want to flush the buffer without closing the file. We might just want to make sure that all writes up to this point have been actually reflected on the disk. So, there is a command flush which does this. In case we say flush, it just say if there are any pending writes then please put them all on to the disk, do not wait for the risk drives to accumulate until the buffer is full and then write as you normally would to the disk.

(Refer Slide Time: 13:42)

Here is a typical thing that you would like to do in python, which is to process it line by line. The natural way to do this is to read the lines into a list and then process the list using for. So, you say content is fh dot readlines and then for each line and contents you do something with it. You can actually do this in a more compact way, you can get do away with the name contents and just read directly every line return by the function fh dot readlines. So, this is the equivalent formulation of the same loop.
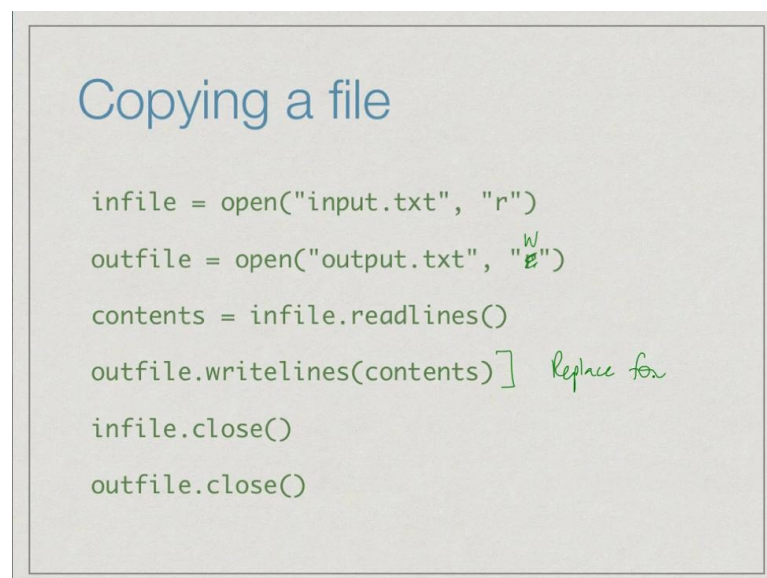
(Refer Slide Time: 14:25)

As an example, for how to use this line by line processing, let us imagine that we want to copy the contents of a file input dot txt to a file output dot txt.

So, the first thing we need to do is to make sure that we open it correctly. We should actually open outfile with mode 'w' and in file mode 'r'. This tells that I am going to read from infile and write to outfile. Now, for each line in returned by readlines on infile, remember that when I get line from readline the backslash n is already there, if I do not do anything to the backslash n, I can write it out the exactly the same way. So, for each line that I read from the list infile dot readlines I just write it to outfile and finally, I close both the files. This is one way to copy one file from input to output.

(Refer Slide Time: 15:25)


```
Copying a file

infile = open("input.txt", "r")

outfile = open("output.txt", "w")    [W]

contents = infile.readlines()

outfile.writelines(contents)    ] Replace for

infile.close()

outfile.close()
```

Of course, we can do it even in one shot because there is a command called writes lines, which takes the list of strings and writes them in one shot. So, instead of going line by line through the list readlines we can take the entire list contents and just output it directly through writelines, this is an alternative way where I have replaced. This is basically replacing the for. So, instead of saying for each line in infiles I can just write it directly out.

One of the things we are talking about is this new line character which is a bit of annoyance. If we want to get with a new line character, remember this is only a string and the new line character is going to be a last character in this string. So, one way to get is just to take slice of the string up to, but not including the last character. Now, remember that we when we count backwards minus 1 is the last character. If we will take the slice from 0 up to minus 1 then it will correctly exclude the last character from the string.

So, s is equal to line colon minus 1, will take the line and the strip of the last character which is typically backslash n that we get, when we do readlines. Now, in general we may have other spaces. So, remember when you write out text very often we cannot see the spaces the end of the line because they are invisible to us. These are what are called white space. So spaces, tabs, new lines, these are characters which do not display on the screen, especially spaces and tabs and there at a end of line, we do not know the line ends with the last character we see there are spaces after words.

So, r strip is a string command which actually takes a string and removes the trailing white space, all the white spaces are at end of the line. In particular there is only a backslash n and it will strip to a backslash n. It also strips to other jump there is some spaces and tabs before the backslash n and return that. So, s equal to line dot r strip that is strip line from the right of white space. This is an equivalent thing to the previous line

except it is more general because strips all the white space not just by the last backslash n, but all the white spaces end of the line.

We can also strip from the left using l strip or we can strip on both sides if we just say strip without any characterization l or r. These are the string manipulation functions and we will look at some more of them, but this is just useful one which has come up immediately in the context of file handling. So, before we go ahead let us try and look at some examples of all these things that we have seen so far.

(Refer Slide Time: 18:05)

We have created a file called input dot txt which consist of a line, the quick brown fox jumps over the lazy dog. Now, let us open the python interpreter and try to read lines from this file and print it out. So, we can say, for instance, that f is equal to open input dot txt in read. Now, I have opened the file and now I can say, for instance, for line in f dot readlines print line. Now, you will see something interesting happening here, you will see that we have now a blank line between every line our file.

Now, why is there blank lines between every line in our file that is because when we readlines we get a backslash n character from the line itself. So, the quick brown, the first line end with the backslash n, fox end with backslash n and then over and above that if you remember the print statement adds a backslash n of its own. So, actually print is putting out to blank lines for each of these.

Now, let us try and do this again, supposing I repeat this thing and now I do this again, now nothing happens the reason nothing happens is because we had this sequential reading of the file. So, the first time we did f dot readlines, it read one line at a time and now we are actually pointing to the end of the file.

(Refer Slide Time: 19:49)

```
>>> text = fh.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fh' is not defined
>>> text = f.read()
>>> text
''
>>> text = f.readline()
>>> text
''
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>> f = open("input.txt","r")
>>> for line in f.readlines():
...    print(line,end="")
...
The quick brown
fox
jumps over the lazy
dog.
>>>
```

If for instance, at this point we were to say text equal to fh dot read, sorry f dot read, then text will be empty string. This is the indication that we have actually reached the end of the file. Similarly, if we try to say readline again text will be empty string. So, remember we said that if read or readlines returns the empty string then we have reached the end of the file. The only way we can undo this is to start again by closing the files. So, what we say is f dot close. This closes of the file.

Now, if you try to do f dot read then we will get an error saying that this is not being defined. So, we do not have f with us anymore. So, again we have to say f is open input dot txt r and now we can say while for line in f dot readlines for each line. Supposing, we now use that trick that we had last time which is to say end equal to empty string that says do not insert anything after each print statement. Now, if you do this you see we get back to exactly the input files as it is without the extra blank lines because print is no longer creating these extra lines.
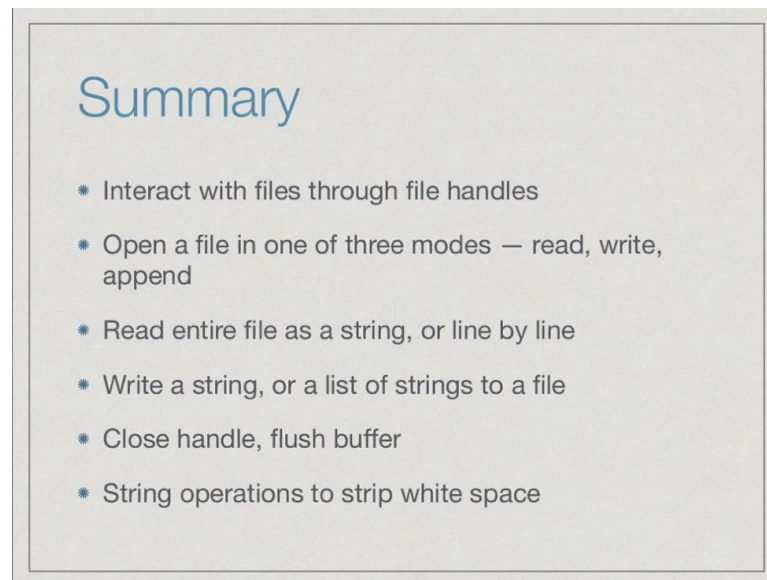
```
>>> f = open("input.txt","r")
>>> g = open("output.txt","w")
>>> for line in f.readlines():
...     g.write(line)
...
16
4
20
5
>>> f.close()
>>> g.close()
>>> ^D
madhavan@dolphinair:...016-jul/week5/python/files$ more output.txt
The quick brown
fox
jumps over the lazy
dog.
madhavan@dolphinair:...016-jul/week5/python/files$ ▌
```

Let us say we want to copy input dot txt to a output dot txt, we say f is equal to open input dot txt r as before and we say g is equal to open output dot txt w and now we say for line in f dot readlines, g dot write line.

Now, notice you get the sequence of numbers why do we get a sequence of numbers that is because each time we write something it returns a number of character written and it will turn out, if you look at the lines quick brown fox, etcetera, for example, the second line is just fox, fox has three letters, but if you include the backslash n its wrote 4 letters. That is why quick brown was 15 letters plus a backslash n, fox was a 3 plus backslash n. So, this is line by line. Now, if I correctly close these files then come out of this, then output dot txt is exactly the same as input dot txt.

(Refer Slide Time: 22:26)

Summary

* Interact with files through file handles
* Open a file in one of three modes — read, write, append
* Read entire file as a string, or line by line
* Write a string, or a list of strings to a file
* Close handle, flush buffer
* String operations to strip white space

To summarize what we have seen is that, if you want to interact with files we do it through file handles, which actually corresponds to the buffers that we use to interact between the memory and the file on the disk. We can open a file in one of three modes; read, write and append. We did not actually do an example with the append, but we do append what it will do, keep writing beyond where the file already existed, otherwise write will erase the file and start from the beginning.

We saw that read, readline and readlines, using this we can read the entire file in one shot of the string or read it line by line. Similarly, we can either write a string or we write a list of strings too. So, we have a write command in a writelines and writelines are more correctly to be interpreters write list of strings.

Finally, we can close the handle when we have done and in between that we can flush the buffer by using flush command and we also saw that there are some string operations to strip white space and this can be useful to remove these trailing backslash n which come whenever you are processing text files.