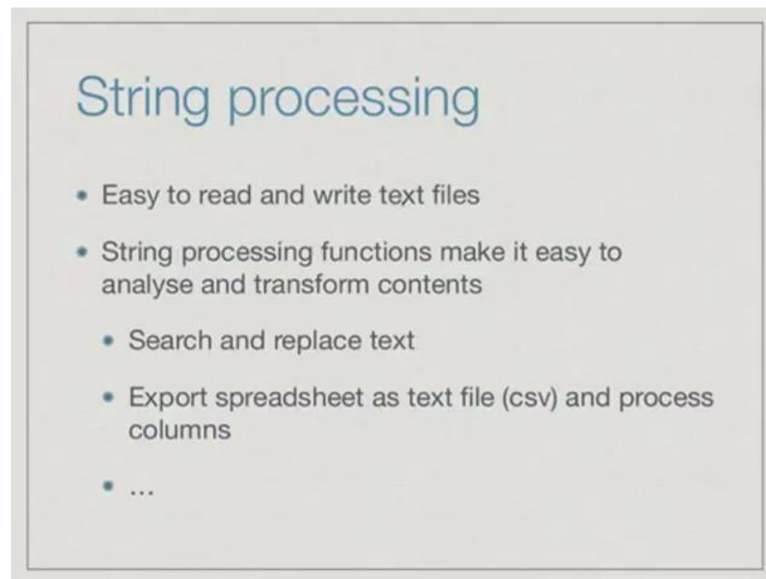
Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 05
Lecture - 04
String Processing

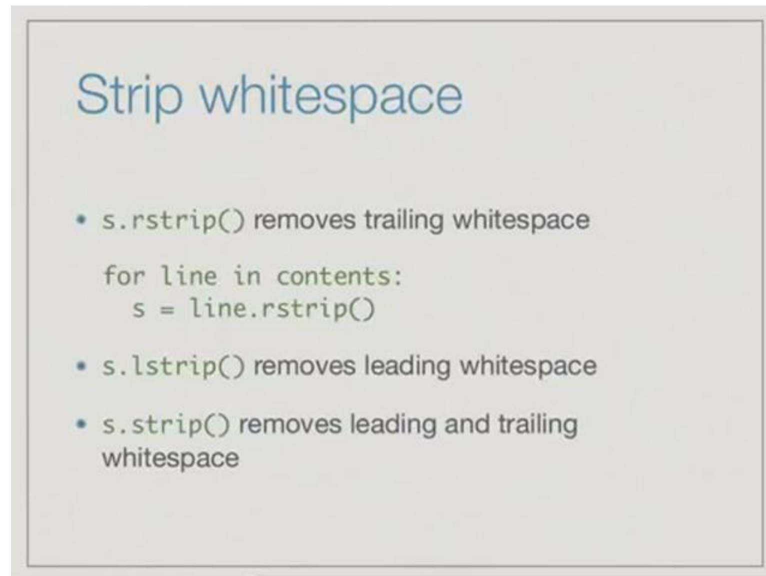
(Refer Slide Time: 00:02)



The last lecture we saw how to read and write text files. And reading and writing text invariably involves processing the strings that we read and write. And so, Python has a number of string processing functions that make it easier to modify this content.

So usually, you are reading and writing files in order to do something with these files, and to do something with this you can use built in string functions which are quite powerful. Among other things, what you can do with these string functions is for example, search for text or search and replace it. A typical use of string processing for a file is when we take something like a spread sheet and export it as text. There is something called a comma separated value format CSV, where the columns are output separated by commas as text. Now, what we can do with a string file is to read such a file line by line and in each line extract the columns from the text by reading between the commas. So, we will see all this in this lecture.

(Refer Slide Time: 01:04)



The first example of a string command that we already saw last time is the commands to strip white space right. We have rstrip, which we used for example to remove the trailing whitespace backslash n in our lines, and we had lstrip to remove leading whitespace, and we had strip which removes it on both directions. Let us see how this works.

(Refer Slide Time: 01:30)

```
>>> s = "        hello          "
>>> s
'    \thello    \t'
>>> t = s.rstrip()
>>> t
'    \thello'
>>> t = s.lstrip()
>>> t
'hello    \t'
>>> t = s.strip()
>>> t
'hello'
>>>
```

Let us create a string which as whitespace before and afterwards, so let us put some spaces may be a tab and then the word hello and then two tabs. We have a string which has whitespace and you can see the tabs are indicated by backslash t and blanks. Now, if

we want to just strip from the right we say t is equal to s dot rstrip. Remember this strip command strings are immutable right it won't change s it will just return a new string, if I say t is s dot r strip it will strip to the whitespace to the right and give me t, if I look at t it has everything up to hello but not that tab and the space afterwards.

Similarly, if I say t is s dot l strip it will remove the ones to the left now t will start with hello, but it will read have the whitespace at the end. Finally, if I say t is s dot strip then both sides are gone and I will just get the word that I want. This is useful because when you ask people to type things and forms for example, usually if they leave some blanks before and after, so if you want everything before the first blank to be lost and the last blank only keep the text in between then you can use the combination of lstrip, rstrip or just strip to extract the actual data that you want from the file.

(Refer Slide Time: 02:43)

Searching for text

s.find(pattern)

- Returns first position in s where pattern occurs, -1 if no occurrence of pattern

s.find(pattern,start,end)

- Search for pattern in slice s[start:end]

s.index(pattern), s.index(pattern,l,r)

- Like find, but raise ValueError if pattern not found

The next thing that may we want to do is to look text in a string. There is a basic command called find. So, if s a string and pattern is another string that I am looking for in s, s dot find pattern will return the first position in s which pattern occurs. And if pattern does not occur, so the positions will there obviously be between 0 and the length of s minus 1. We already wrote some our own implementation of this earlier. So, if it does not occur it will give you minus 1.

Sometimes you may not want to search entire string, so pattern takes an optional pair of argument start and end in which case instead of looking for the pattern from the entire

string it looks at a slice from start to end with the usual convention that this is the position from start to end minus 1. There is another version of this command called index. And the difference between find and index is what happens when the pattern is not found. In find when the pattern is not found you get a minus 1, in index when the pattern is not found you get a special type of error in this case a value error. So again let us just see how these things actually work.

(Refer Slide Time: 03:54)

```
>>> s = "brown fox grey dog brown fox"
>>> s.find("brown")
0
>>> s.find("brown",5,len(s))
19
>>> s.find("cat")
-1
>>> s.index("cat")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>>
```

So, we have a string here s which contains the word "brown fox grey dog brown fox." Now if I ask it to look for the first occurrence of the word "brown", then it will return the position 0 because it is right there at the beginning of string. If on the other hand I do not want this position, but I wanted to say starting from position 5 and going to length of s for example, then it will say 19 and if you count you will find that the second occurrence of brown is at position 19.

If on the other hand I look for something which is not there like "cat" then find will return minus 1, so minus 1 is not the error but the indication that the string was not found. The difference with index is that if I give index the same thing instead of a minus 1 it gives me a value error saying the substring does not occur right this is how find and index work.

(Refer Slide Time: 04:53)

The next natural thing after searching is searching and replacing. If I want to replace something I give it two strings what I am searching for and what I am replacing it with and it will return a copy of s with each occurrence of the first string replaced by the second string. Now this can be controlled in the following ways; supposing, I do not want to each occurrence, but I only want say the first occurrence or the first three occurrences.

So, I can give it an optional argument saying how many such occurrences starting from the beginning should be replaced. It says replace at most the first n copies and notice that like and strip and all that, here it's because changing this string replacing something by something else, is not that s is going to change because strings are immutable is going to return us the transform string. So let us look at an example.

(Refer Slide Time: 05:45)

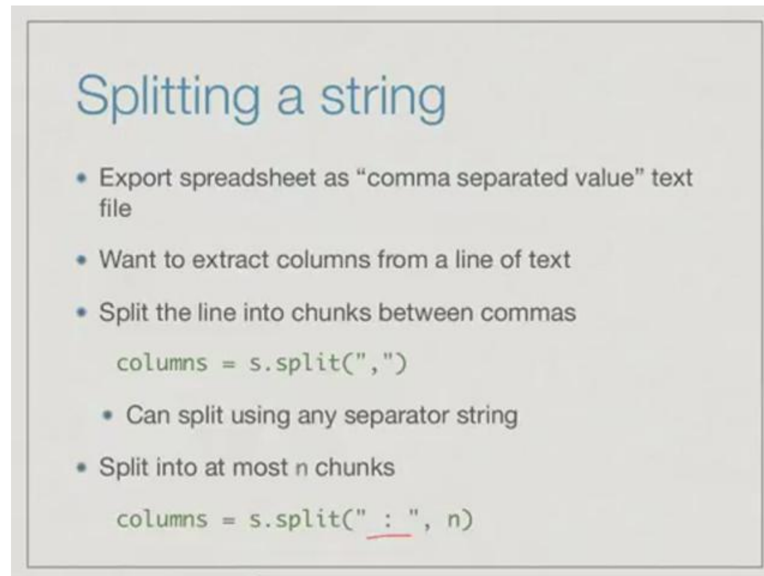
```
>>> s = "brown fox grey dog brown fox"
>>> s.replace("brown","black")
'black fox grey dog black fox'
>>> s.replace("brown","black",1)
'black fox grey dog brown fox'
>>> t = "ababa"
>>> t.replace("aba","DD")
'DDba'
>>> t = "abaaba"
>>> t.replace("aba","DD")
'DDDD'
>>>
```

Once again let us see our old example; s is "brown fox grey dog brown fox" and now supposing I want to replace "brown" by "black", then I get 'black fox grey dog black fox.' If I say only want 1 to be replaced then I get 'black fox grey dog' where the second brown is left unchanged. Now you may ask what happens if I have this pattern it does not neatly split up if I have the different copies of brown overlaps. Supposing, I have some stupid string like "abaaba" and now I say replace all "aba" by say "DD".

Now the question is, will it find two aba s or 1 aba, because there is an aba starting at position 0, there is also an aba in the second half of the string starting at position 2. The question is will it mark both of these and replace them by DD, well, it does not because it does it sequentially so it first takes the first aba, replaces it by DD, at this point the second aba has been destroyed. So it will not find it.

Whereas, if I had for instance two copies of this disjoint then it would have correctly found this and given me DD followed by DD. So, there is no problem about overlapping strings it just does it from right to left and it makes sure that the overlap string is first written, so it will not the second copy will not get transformed.

(Refer Slide Time: 07:17)



The next thing that we want to look at is splitting a string. Now when we take a spreadsheet and write it out as text, usually what happens is that we will have an output which looks like this. The first column would be written followed by comma then second column, so if we had three columns then the first column set 6 second column set 7 and the third was string hello, then we write it out a text as you will get 6, 7 and "hello". Actually "hello" is a bit of problem because it has double quotes let us not use hello let us use something simpler. So let us just say that we had three numbers 6, 7 and 8 for example.

Now, what we want to do is we want to extract this information. So, we want to extract the individual 6, 7 and 8 that we had as three values. So what we need to do is look for this text between the strings, so we want to split the column into lines into chunks between the commas and this is done using the split command. So, split takes a string s and takes a character or actually could be any string and it splits the columns it gives you a list of values that come by taking the parts between the commas. So, up to the first comma is a first thing. So columns is just a name that we have used, it could be any list. The first item of the list will be up to the first comma then between the first and second comma and so on and finally after the last comma.

Comma in this case is not a very special thing you can split using any separator string. And again just like in replace we could control how many times we replaced, here we

can also control how many splits you make. So, you can say split according to this string notice that this could be any string so here we are splitting it according to space colon space. But we are saying do not make more than n chunks, if we have more than n columns or whatever chunks which come like this beyond a certain point we will just lump it as one remaining string and keep it with us. So again let us see how this works.

(Refer Slide Time: 9:22)

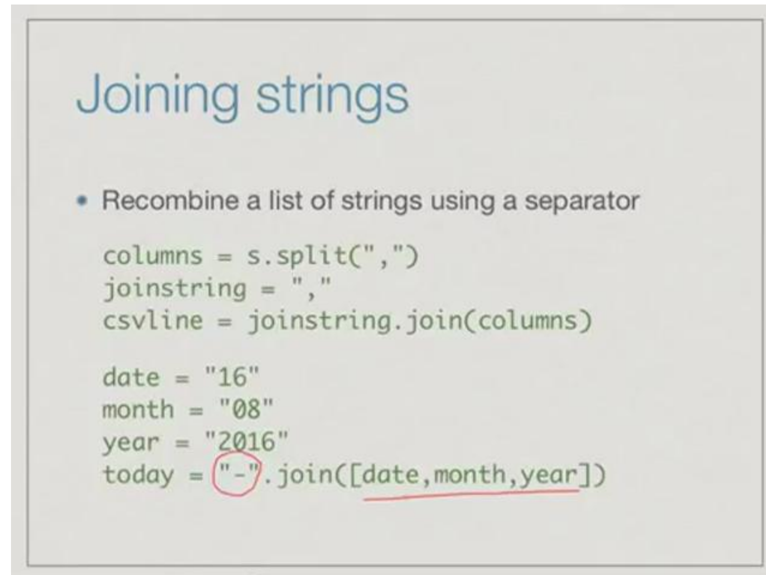
```
>>> csvline = "6,7,8"
>>> csvline.split(",")
['6', '7', '8']
>>> csvline.split(",",1)
['6', '7,8']
>>> csvline = "6#?7#?8"
>>> csvline.split("#?")
['6', '7', '8']
>>>
```

Suppose this is our line of text which I will call CSV line it is a sequence of values separated by commas notice it is a string. Now if I say CSV line dot split using comma as a separator and then I get a list of values the string 6, the string 7, the string 8. Remember this is exactly like what we said about input it does not get you the values in the form that you want you then have to convert them using int or these are still strings. So, it just takes a long string and splits it into a list of smaller strings. Now here there are three elements so if I say for example I only want position 0, 1, 2. So, if I say I only wanted to do it once then I get the first 6, but then 7 and 8 does not get split because it only splits once.

Now, if I change this to something more fancy like say hash comp question mark. So now I have a different separator it's not a single character, but hash question mark then I can say split according to hash question mark and this will give me the same thing. You can split according to any string it's just a uniform string. There are more fancy things you can do with regular expression and all that, but we won't be covering that for now.

As long as you have a fixed string which separates your thing you can split according to that fixed string.

(Refer Slide Time: 10:49)



So, the inverse operation of split would be to join strings. Supposing, we have a collection of strings and I want to combine it in to a single string separate each of them by a given separator. So as an example, supposing we take s which is some CSV output and we split it into columns on comma, and then we can take join string and set it to the value comma and then use that to join the columns.

Now this is a bit confusing, so join is a function which is associated with a string. In this case a string in concerned is a comma. So it says, more or less you are saying comma dot join columns which is use comma to join columns. So, you have just given it a name here join string is equal to comma and then CSV line is join string dot join columns.

So what this says is, use comma to separate so if at the end of this I had got like last time 6, 7 and 8, then this will now put them back as 6 comma 7 comma 8 into a single string. Here is another example, here we have a date 16 a month 08 and a year 2016 given as strings and I want to string it together into a date like we normally use with hyphens.

Here instead of giving an intermediate name to the hyphens and then saying hyphens dot join I directly use this string itself, just want to illustrate that you can directly use this joining string itself as a constant string and say use this to join this list of values. All you

need to make sure is what you have inside the join in the argument is a list of strings and what you applied to is the string which will be used to join them. Let us just check that this works the way we actually intended to do.

(Refer Slide Time: 12:38)

```
>>> date = "16"
>>> month = "08"
>>> year = "2016"
>>> "-".join([date,month,year])
'16-08-2016'
>>>
```

Let us directly do the second example. Supposing, we say date is 16, remember these are all strings month is 08, year is 2016, and now I want to say what is the effect of joining these three things using dash as separator and I get 16 dash 08 dash 2016.

(Refer Slide Time: 13:09)

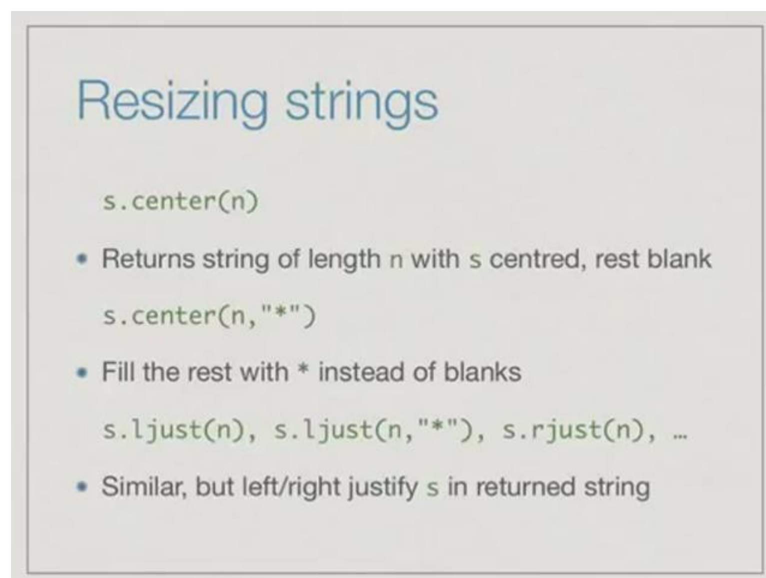
Converting case

- Convert lower case to upper case, ...
- `s.capitalize()` — return new string with first letter uppercase, rest lower
- `s.lower()` — convert all uppercase to lowercase
- `s.upper()` — convert all lowercase to uppercase
- `s.title()`, `s.swapcase()`, ...

So there are many other interesting things you can do with strings for example, you can manipulate upper case and lower case. If you say capitalize, what it will do is it will convert the first letter to upper case and keep the rest as lower case, if you say s dot lower it will convert all upper case to lower case, if you say s dot upper it will convert all lower case to upper case and so on.

There are other fancy things like s dot title. So, title will capitalize each word. This is how it normally appears say in the title of a book or a movie. S dot swap case will invert lower case to upper case and upper case to lower case and so on. So there are whole collection of functions in the string thing which deal with upper case, lower case and how to transform between these.

(Refer Slide Time: 13:52)



The other thing that you can do with strings is to resize them to fit what you want. So if you want to have a string which is positioned as a column of certain width then we can say that center it in a block of size n. So what this will do is it will return a new string which is of length n with s centered in it.

Now by centering what we mean is that on either side there will be blanks instead of blanks you can put anything you want like, stars or minuses. You can give a character which will be used to fill up the empty space on either side rather than a blank. Now you may not want it centered or you may not want to the left or the right, so you can for example left justify during ljust or rjustify during rjust and again you can give an

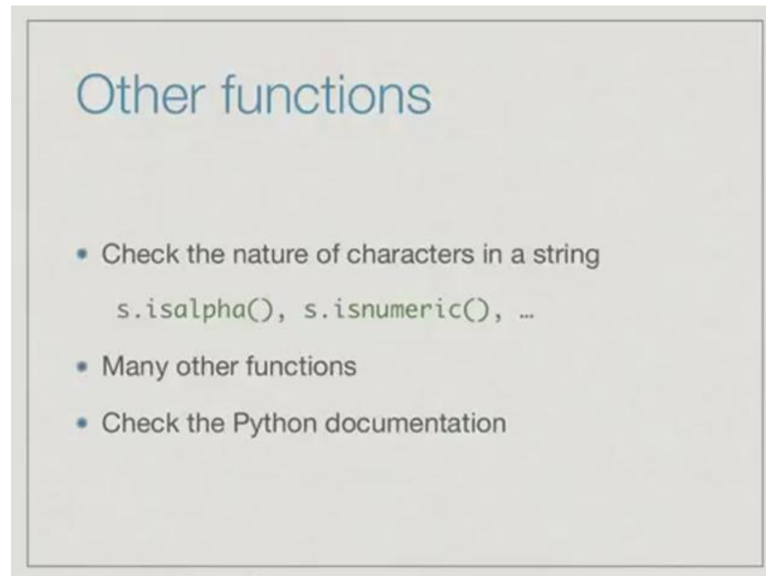
optional character and so on. S, we can just check one or two of these just to see how they work.

(Refer Slide Time: 14:46)

```
>>> s = 'hello'
>>> s.center(50)
'                      hello                       '
>>> s.center(50,'-')
'----------------------hello-----------------------'
>>> s.ljst(50,'-')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'ljst'
>>> s.ljust(50,'-')
'hello---------------------------------------------'
>>> s.rjust(50,'-')
'---------------------------------------------hello'
>>> 
```

Suppose, we take a short string like 'hello' and now we center it in a large block of say 50. We say s dot center 50, then this gives us hello with a lot of blank spaces on either side. Now we can replace those blank spaces by anything we want, so say minus sign then we will get a string of a minus signs or hyphens before that. Now we can also say that I want the thing left justified in this not a center. So if I do that then I will get hello at the beginning and a bunch of minus signs, similarly with rjust and so on.

(Refer Slide Time: 15:23)



Some of the other types of functions which we find associated to strings are to check properties of strings. Does s consists only of the letters a to z and capital a to capital z. So that is what s dot is alpha says is it an alphabetic string, if it is true it means it is, if it is not it has at least one non alphabetic character. Similarly is it entirely digits, is numeric will tell us if it is entirely digits. So, there is a huge number of string functions and there is no point going through all of them in this thing, we will if we need them as we go along we will use them and explain them.

But you can look at the Python documentation look under string functions and you will find a whole host of useful utilities which allow you to easily manipulate strings. And this is one of the reasons that Python is a popular language because you can do this kind of easy text processing. So you can use it to quickly transform data from one format to another and to you know change the way it looks or to resize it and so on. String functions are an extremely important part of Pythons utility as a glue language for transforming things from one format to another.