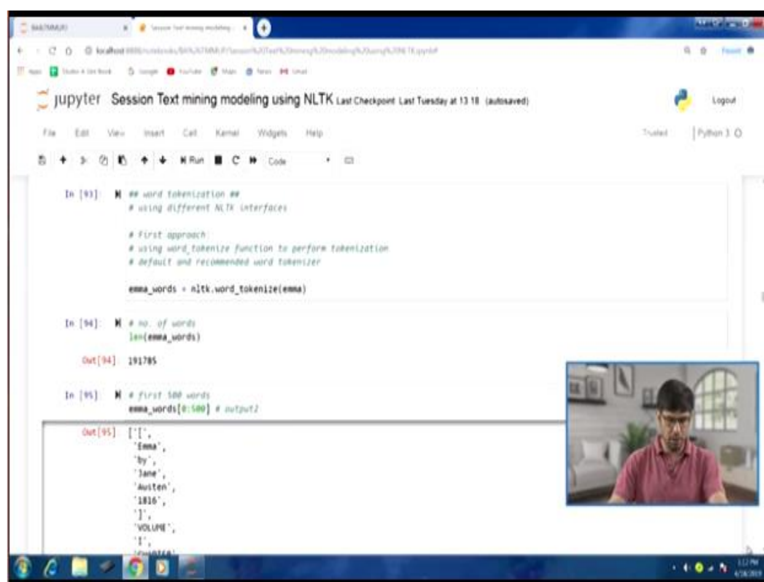
Business Analytics And Text Mining Modeling Using python
Prof. Gaurav Dixit
Department of Management Studies
Indian Institute of Technology Roorkee

Lecture-38
Text Mining And Modeling-Part I

Welcome to the course business analytics and text mining modeling using python. So, in previous lecture we had started our discussion on word tokenization. So, word we will do, we will do a small recap of 2, 3 approaches that we were able to cover in the previous lecture and then we will move forward to other aspects of tokenization and you know in other steps as well.

(Refer Slide Time: 00:53)



So, word tokenization we talked about first approach that is using the default and recommended word tokenizer that is using the word underscore tokenize function.

(Video Starts: 00:53)

So, we looked at you know this particular function and you know used emma text that we have been using for to demonstrate these tokenization examples. We can have a look at output here again we also you know discussed another approach second approach using tree bank word tokenizer.

So, internally this is placed on regular expansion, so here however this is typically use to convert sentence into word tokens. So, we took the example of Emma underscore sense and now we use that to create you know tokens here, so you can see tree wang_wt or tokenize and we were able to create these tokens you can see few aspects how we could be different from the default tokenizer that also be discussed.

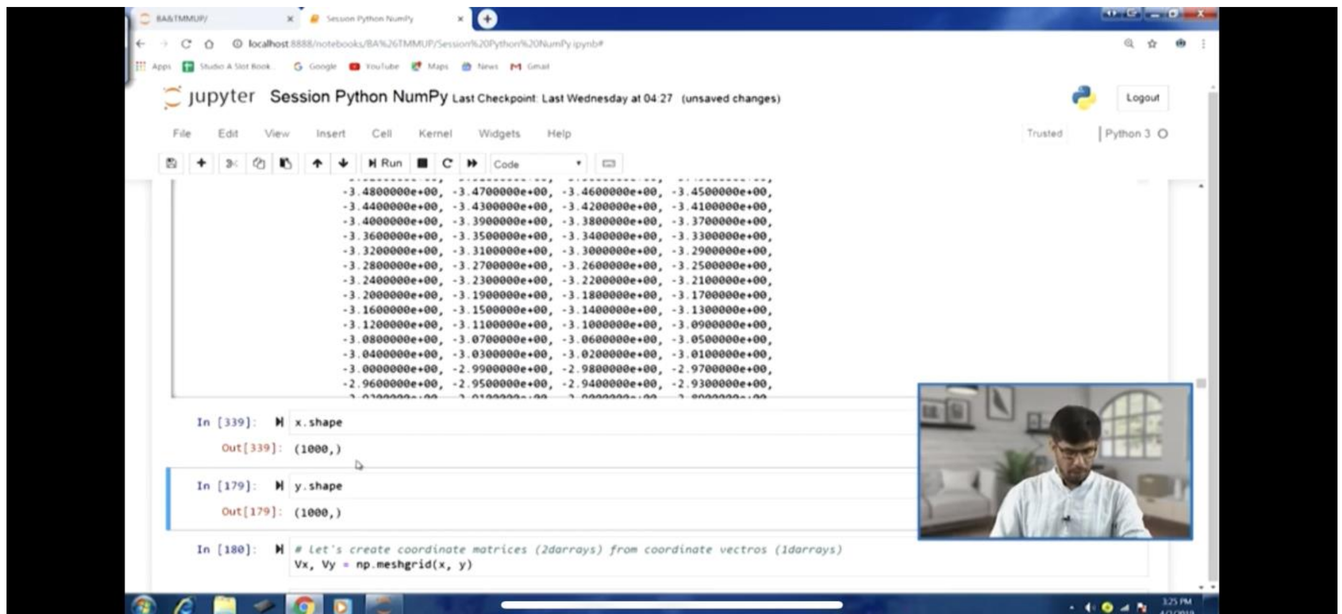
Then we move to the next approach that is third approach where we use the regular expressions to redefine our patterns and those regular expression base pattern can then we use to create regular expression object, rejects objects and then we can call tokenize method there and perform our word tokenization. So, we looked at this aspect as well and we saw that how we were able to get rid of any kind of punctuation tokens which are punctuation marks which have also become tokens in the previous 2 approaches.

So, we looked at how using regular expression we got nicer you know tokens there then we also discuss another you know regular expression approach in the sense we change the pattern and mainly to identify gaps. And you we saw an output that slight difference in terms of tokens, so we did not have any tokens with having just punctuation marks. But few tokens were, you know different in the sense because we were trying to identified gaps there.

We also looked at start and in dices when we are using this rejects approach. And how they can be used to compile the tokens that we are producing there, so these approaches we are able to cover. Now let us talk about the fourth approach that is there for word tokenization, so we can use derived classes of different types of word tokenization as well. So, for example word punk tokenizer class, we can use an instance of this class to perform our word tokenization.

So, independent alphabetic and non alphabetic tokens, for example, we can obtain using an instance of this class. So, we can use word punk tokenizer function here to create the instance of this particular class, word punct_wt. So, let me run this and then we can call the tokenize function using this object word punct_wt.tokenize and let me run this will have the word tokens. Let us have a look at the first 500 you know tokens here.

(Refer Slide Time: 02:34)



So, you can see the output is quite similar to previous output that we have had and in this particular case will have the you know alphabetic and non-alphabetic tokens independent separate. So, you can see again in that sense the output is quite similar to the previous outputs here you can see in our opening bracket, comma, they have also become tokens. So, let us move forward now we will take another you know derived class here, wide space tokenize your class. So, we will take an instance of this wide space tokenizer class.

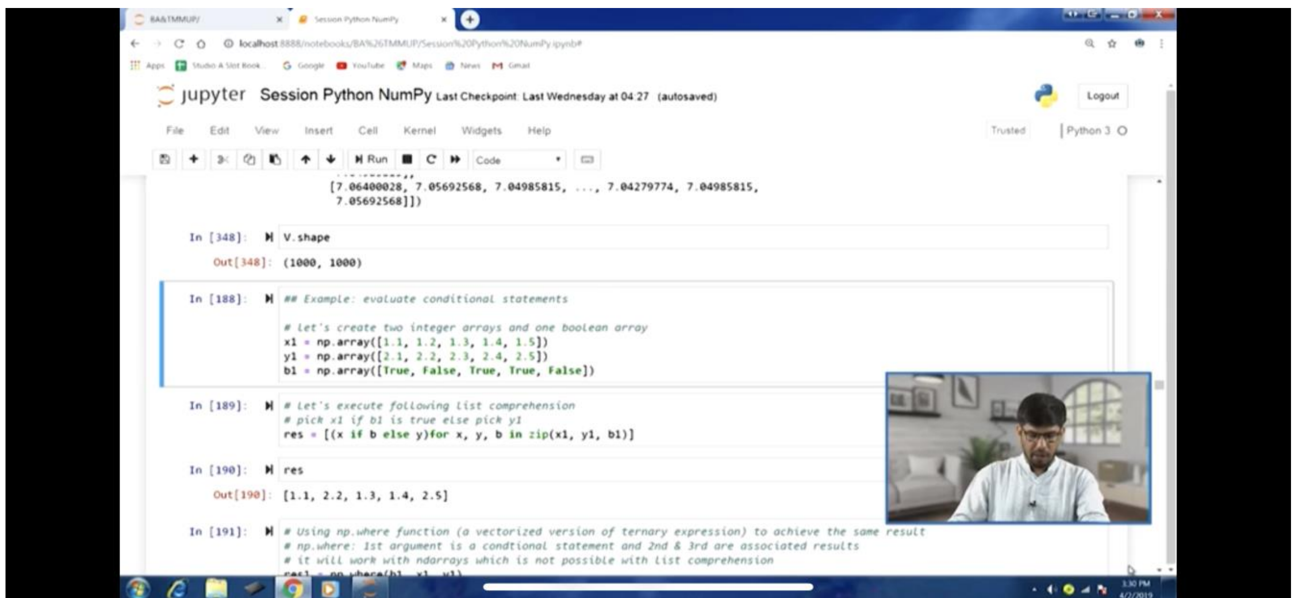
So this is based on you know again based on wide spaces, so we will use the white space tokenizer function here to create an object instance. So, wide space underscore wt while calling nltk.wide space tokenizer will have this object, will use this object to call tokenize you know method on Emma and we will have our set of tokens. Let us have a look at the tokens here, so you can see again the output is similar.

But you would see that the tokens slight differences, that in the sense that we are not seeing our punctuation marks has been taken as tokens in this one as well, because this is based on the wide spaces. So, different approaches to create word tokenization and you know slightly varying output, so depending on our requirement for you know depending on our problem and the kind of test purpose we might have, we might go for any of those approaches there.

Now let us move to the next aspect here, so that is about removing specific characters. Now some of these word tokenization approaches that we talked about, they had the you know a special character also as tokens. So, what we can do about removing them, so this particular removing a special character thing is as you would understand in the programming word it can even be done before tokenization or after tokenization.

Because essentially this is a programming you know problem in the sense only the input format is going to be changing. So, when we have done the tokenization will be you know removing these special characters which have now become tokens. And if we have not done tokenization then we will be removing these specific characters from the text itself. So, both are possible in the programming context, so we will discuss both of these scenarios here.

(Refer Slide Time: 07:17)



So, for this removing specific characters we would be using these libraries, so we will import them import RE and string and you know let us first take the case of after tokenization. So, in the first approach will take a regular expression pattern and in this pattern we are going to use this string dot punctuation attribute and this attribute actually consist of all possible combination all possible spill characters or symbol.

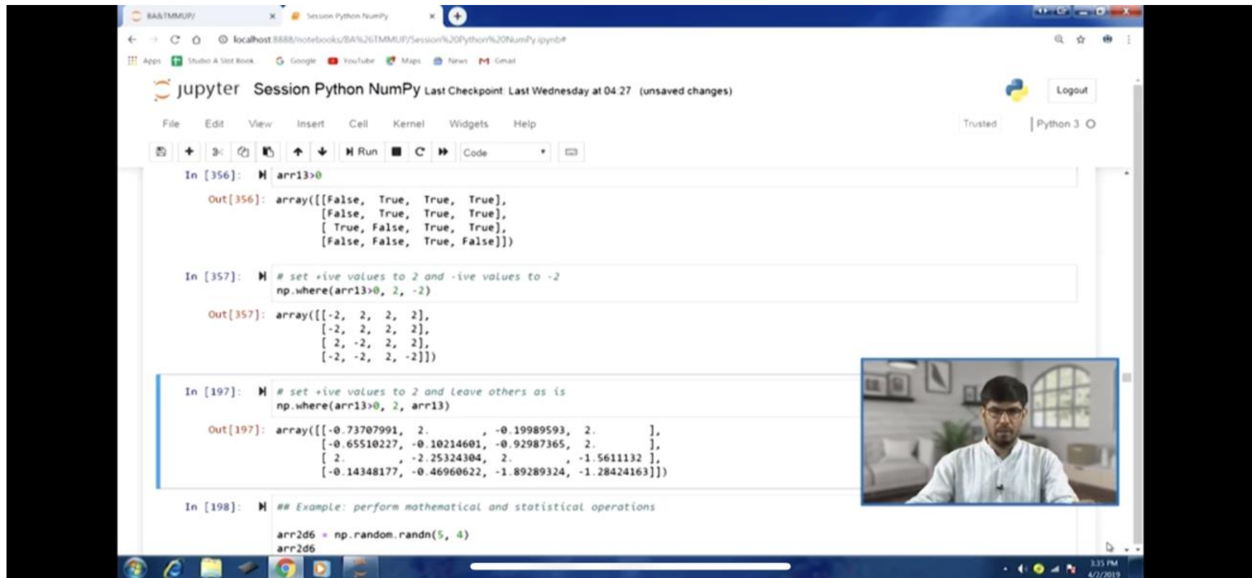
Because the our idea is to remove special characters and symbols. So this particular attribute will help us in the sense it has you know adjustable list of all those characters. So, our regular expression pattern will take you know will use this particular attribute , then if you look at the pattern that we are trying to define here we are using the lead or compile you know function here to compile this pattern and in the parenthesis you see that we are using you know string dot punctuation that there we are escaping this using re.escape for function here.

So, that those special characters are skipped out and then we are calling format here that is mainly to copy the patterns string that would be created and will replace you know copy this string into the replacement field denoted by you know braces there. So, using this will have the pattern and then will be compiling this, so in one go we would be doing all these and we will have our pattern.

Now once this is done, once we have the pattern object here, we can use a list comparison which is going to be based on some method and will find all matching tokens as per the patterns. We will apply this pattern on our text or that we have, so the text right now we are dealing with the after tokenization scenario. So, we will have our you know tokens, so we will apply you know these pattern on these tokens and all the matching tokens you know they will be able to find out them and then they will replace the with the empty string.

So, the same thing is you know here in this particular code you can have a look, here if you look at the this list comprehension we are calling pattern 4.sub and empty you know string, token. And this token is coming from emma_words that we have already computed word tokenization we have already perform. So, for each token that is there in emma words will be applying this pattern and will be replacing these tokens wherever there special character or symbols are matching with the mpt string.

(Refer Slide Time: 12:19)



Then we are calling or we are doing all this within the filter function, so because in the filter function the first argument is none, so we are passing none as the you know function to be applied here. So, essentially you know all the mt tokens are going to be removed here, if you are interested in finding out more about the filter function, you can always go to the help section and we can go to the python difference.

And we can have a look at what this filter function is about because you know some of these functions are quite useful in text pausing. So, sometimes it is better that we have a look at them using our help reference, so that we have better idea about the arguments and the kind of processing that will be done by those functions. And also the you know written into value that is going to be there, so filter.

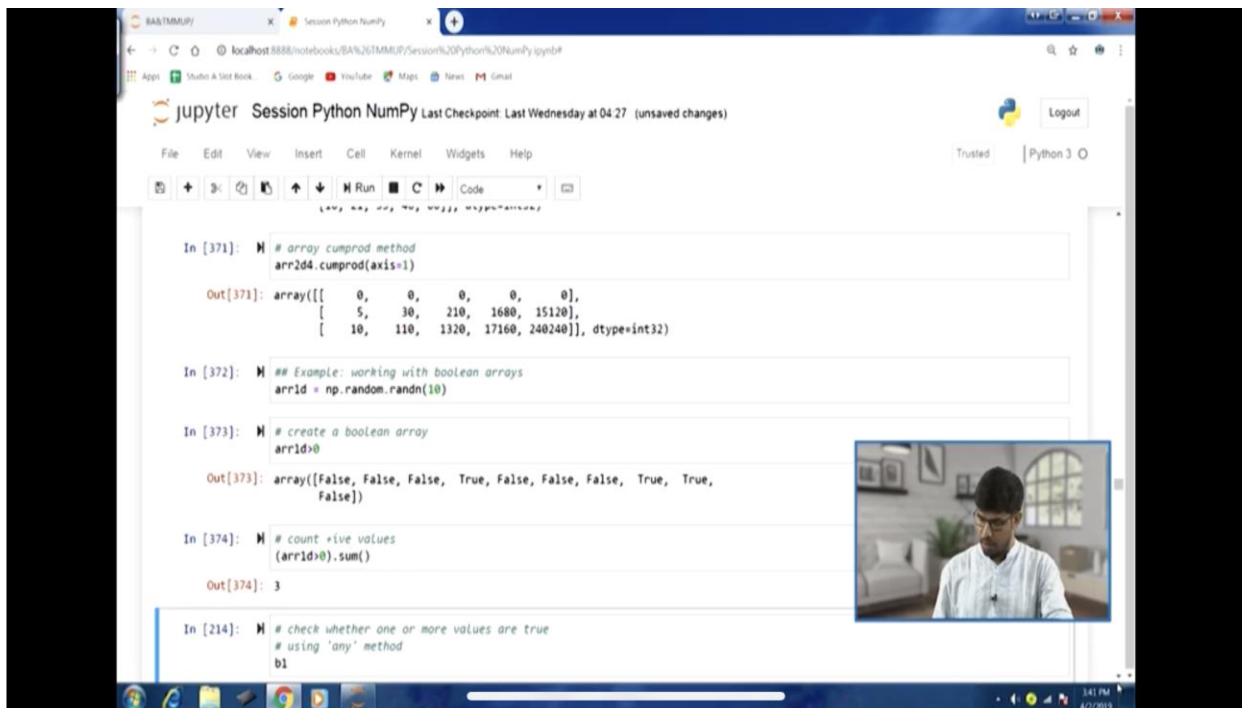
So, you can see in this filter function you can see the it is going to construct an iterator from those elements of the iterable and the second element that is there. And primarily you can focus on this part you can see it function is not the identity function is assumed and that is all elements iterable that are false or removed. So, this is essentially word we want you know we are we have a specified none and therefore identity function is going to be there.

We would like to you know remove the you know empty tokens there, so that is what we want. So, this is what we can achieve using this particular function. So, let us go back and you know and then output of you know output of this particular function would be is going to be an iteratable object. So, therefore we are using you know list you know function here to create a list, so that we can have a you know output there.

So, let me run this and let us have a look at the first 500 tokens here, so you can see now that you know if you look at the tokens for you would not find any special characters as independent tokens here. So, it start with emma by jane, and you can see we can keep on going like this, we can keep on scrolling and you would see that all these tokens are meaningful English language words.

So, you can see, so in this fashion we are able to actually remove all these special characters. Now, we can also you know have a look at the emma words without you know when we did not have we did not remove the special characters, you can see opening bracket, closing bracket, comma and all those things, they are clearly visible here. So, all these things are gone, so all these characters are removed.

(Refer Slide Time: 17:45)



So I can see here in the output dot is also there, so where is comma is also there. So, various punctuation mark they were treated semicolon is also there, they were treated as tokens, so now, all those are gone. So, even after tokenization you know we can remove these special characters. So, that is one scenario. Now second scenario is could be before tokenization. So can we remove these special character before tokenization. So, let us take talk about the first approach that we can adopt.

So, in this again we can use the regular expression pattern and let us focus on extracting alphanumeric characters because essentially the meaningful tokens would be comprising of you know alphanumeric characters. So, we can you know define a pattern to actually perform that, so we would be adding characters to be retained here, so pattern 5 you can have a look, you have a smaller a to z case the capital A to Z case and 0 to 9 and the space you know.

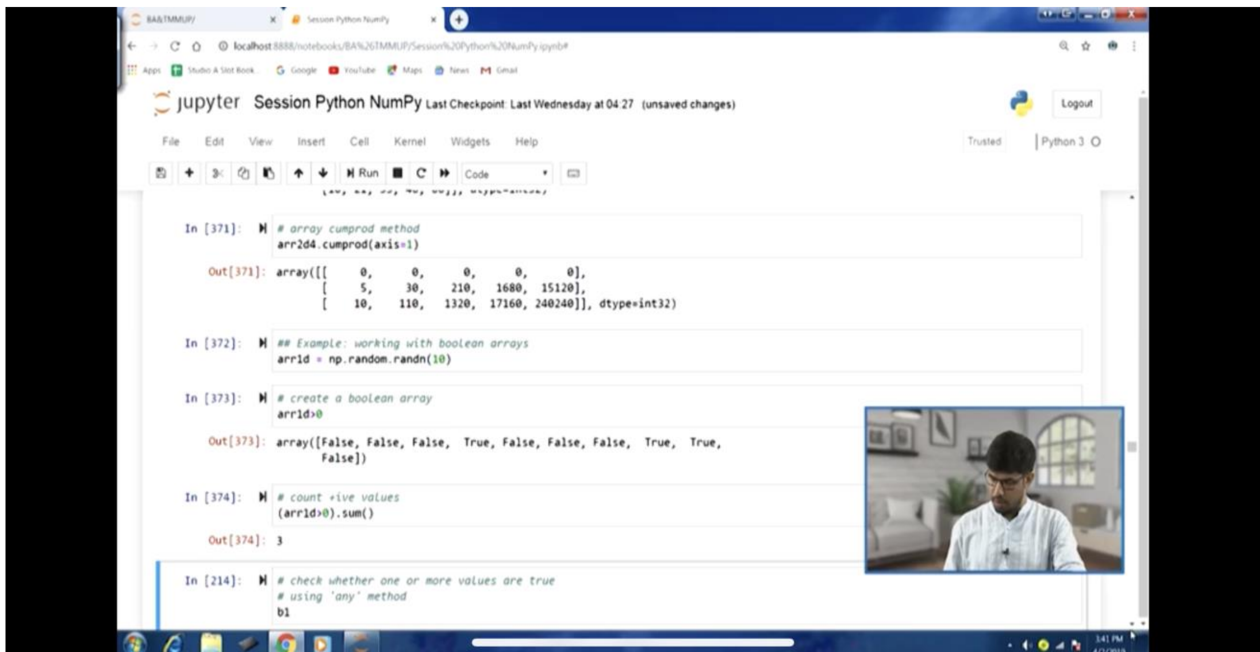
That is something that we would like to have in the pattern, so let us define the pattern here. Now remember this scenario is before tokenization that means the input is actually in the text form we do not have tokens here. So, we will be working these patterns are going to be used on you know text content directly before you know tokenization. So, first thing is that we can use a strip method to remove white spaces, so there are any wide spaces we like to get rid of them.

Then second thing is we can use this list to comprehension and here also we are calling this using this sub method to find all matching you know tokens as per the you know pattern. So, let us say emma_sense you remember that we had you know use this now we had called his sense method and we had obtained sentences from emma. So, for every sentence in that particular output emma underscore sense object we can run this we can call this sub method.

And in this first thing we are stripping the sentence on white spaces and then we are applying pattern. And wherever as per our pattern those characters are not there, so those are going to be removed. So, if I run this and let us have a look at the 500 characters here, so you can see that the output here as you can see emma and we had the brackets there all those are gone many other things you can see. All text is looking much nicer in the sense the punctuation and the special characters you would not see many of them.

So, you can see many of those things are gone and we have a nicer looking output here. So, this is another approach that can be used and you can compare this with the emma_sense output, so we have this here. So, you can see all you know brackets opening brackets closing brackets slash and you know all those things are gone dot, so all those things period character and all those things are gone here. So, you can compare the output here, double dash all those things you would not be able to see now.

(Refer Slide Time: 23:24)



Now, let us talk about another approach that can be used second approach that can be use to you know remove these special characters. So, in this approach, again will create a rejects pattern, now however we would like to you know if we decide to retain certain special characters. So, this approach could be more suitable in those scenarios, so for example we want to we would like to do it in apostrophes and sentence period.

Because the apostrophes might give you know meaning to certain you know words, so therefore maybe would like to prefer to retain them and the sentence periods also. Because it could you know for further processing it could be really useful you know delimiter. So, maybe we would

like to retain them, so in the next pattern that we are going to define we will just add characters to be removed.

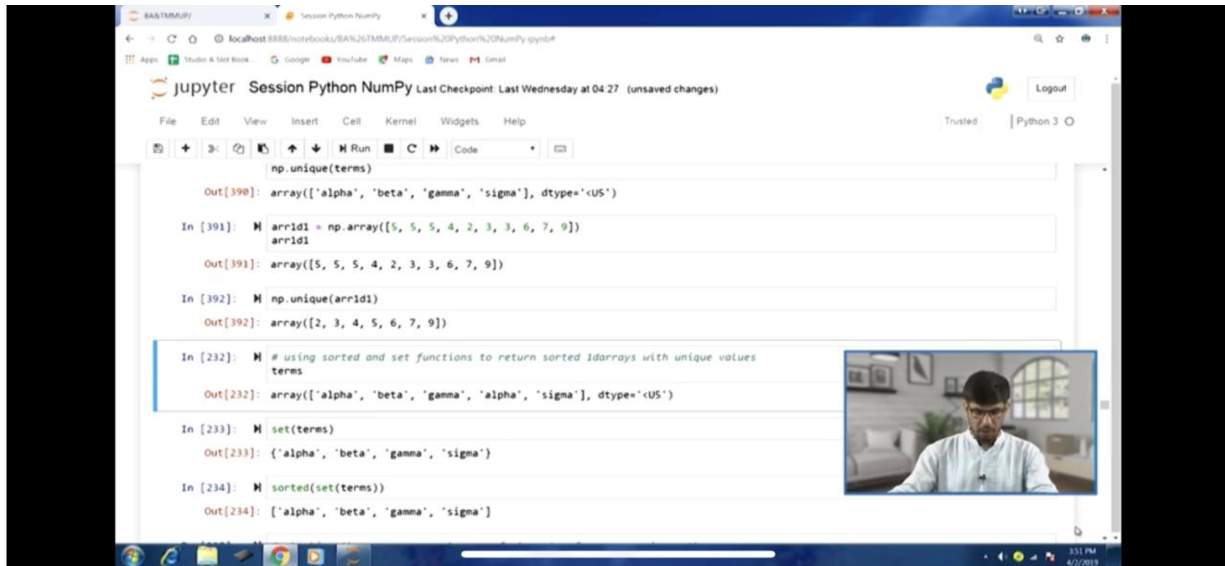
So, you can see that what kind of characters we are you know combining, putting in this pattern defining in this pattern you can see question mark or dollar or ampersand asterisk or percentage or you know at right or you know different till day and so r symbol for r. So, all these characters are there that now we would like to remove, so let me define this pattern and then how we are going to perform.

So, similar kind of list comprehension we are going to use here, so you can see for sentence in Emma underscore signs that we already have. So, for each of the sentence in that particular texts, so you could remember that emma_signs is the non tokenized untokenized you know text that we had. So, from that we are extracting sentence for each sentence we are first stripping than matching the pattern and the pattern is found, then we are replacing that with the empty tokens. So, if I run this let us have a look at the output.

So, if I scroll back and have a look at the output, now you would see that slashes are back and many other things you know dot are back and you would also see that you know many other things which are not you know part of pattern that we have. So again double dash is also back and you would also see that apostrophes are also back, so if you are not able to spot some of those you know.

For example emma situation you can see the real levels in the e.emma is apostrophes are back now which were gone earlier. So, you can see different kind of you know regular expression and different kind of output we have gone. So, removing a special characters we can define our own regular expression base pattern and use that to actually process the textual content test whatever we have.

(Refer Slide Time: 27:36)



So, it can be done before tokenization also and after tokenization also because essentially we are processing text. We have enough you know programming constructs available in python to actually perform this quite easily. So, this was about removing a special character characters now we will move on to the next aspect that is that we that we might have to perform you know.

So, this is about expanding contractions, so what we mean by these contraction, so as you would understand that in English language formally you know we you know we typically write like you know, will not, is not. But sometimes when you know informal language we might write isn't or wouldn't or can't, so you know if there are in your text both kind you know you know both kind of these contractions and expansion, both kind of forms are present.

Then it might get problem for your you know further steps tokenization and many other step that we will be discussing coming lectures. So, it would be better if we are able to you know get the same form and we talk about contraction and expansion then we will prefer the expanded form here. So, that is you know as per the formal writing, so we would like to replace is not to is not and won't to will not.

So, for this we need to have a mapping for this, so, that we are able to whenever we are you know trying to find out when where are we are trying to you know we are using regular

expression to find out these contraction. We should have a mapping, so that we are able to replace these contraction with their corresponding expansion. So, for this we are defining slightly or exhaustive list of contraction map here.

So, you can see aren't or not, can’t, cannot so in this fashion you can see this is a edit object we have and we have key value combination and key we have the contraction and the value we have the expansion. So, we can use this you know to provide the mapping and later on we will see we will use this mapping. This is quite an exhaustive list is small through and so we can use these kind of list actually do certain kind of text passing where we require we prefer full form for text analytics rather than the contractions there.

So, next thing is that will write a function to expand these contractions in the text, so let us define a function here. So, function is def the keyword and expanded underscore contractions and we have 2 arguments for this function send 2 parameter sent and mapping here. So, sent is for the sentence which will be processing and where will be finding all the contractions and then replacing it those with the expansions.

So, mapping is the mapping that we just that I just showed you that is going to use the dict object that I just showed you, that is going to be use to map out the contraction with the expansion. So, we will be referring to this mapping and any sentence that is there will be processing that. So the our dysfunction is going to be using these 2 arguments here. Now first thing we need a pattern for matching contractions with their expansion.

(Refer Slide Time: 30:02)

So let us have a look at this pattern. So, here we are calling the re.compile compile function to actually compile this you know pattern object here regular expression here. So, you can see here also we have the you know first argument you know we have the curly bases. So, that will be constructing this regular expression replacing in this limited space using the braces. So, we are calling format here you know format method here and you can see this particular you know join we have use before also.

So, in this joint we are calling mapping.keys and using in a format we are going to create you know all the keys and they are mapping because essentially we would be you know matching, you know pattern with the text that is there where these contractions are present. And we have also you know define certain flags, for example ignore case and all and for more details on format and you know other you know methods that we are using you can always refer the help section there.

So, with this we have define our pattern, now the next thing is we are again defining another function expand_map, so what it will do. It will take a contraction and you know map it out to when expansion, so it will written the expansion, so it will take contraction as the input and

output would be expansion. So, within this we are calling a group method for the match object that we learned you know during our discussion of regular expression to access subgroups.

So, let us say we have found a match certain occurrence based on our pattern, so that you know if the is you know stored in a match object. So, subgroups you know we can you know access using the group method here. So, let us say we have got the contraction.group 0, so first sub group will get first you know match will get, then next thing that we are doing is we are retaining, we are you know storing the first character.

Because you know we would like to retain the correct case of the word, so for that purpose we would like to store the match 0 here. Then next thing is the core thing of this function find out the expansion that means based on the given contraction that we are passing will like to find out the you know expansions. Here we are using internally expression, so mapping.get we have the mapping, so we are using the get method to based on the match will get the value part the key part is going to be passed on which is the contraction will get the value part.

So, if it is true then we will use that if it is false then you know it might be present in the lower case, so in the else part we have covered that scenario. So, we will have the expansion here and then you know we are again you know reassembling our expansion here using first underscore character +expansion starting from you know character with the index 1. So, second character, so this is to retain the correct case of the word as we discussed and then we will written the expansion.

So, in this fashion any contraction we would be able to, map it to its appropriate expansion using as we discussed using the particular pattern that we are going to compile. Now after this once this is done then with the expand and underscore contraction we are going to call this expand_map. So, you can as you can see the arguments we had is sent is also being passed to expand_contracts.

And so therefore we are going to process the sentence here itself and this you know function itself. So, the next thing is we are calling the sub method and first argument is this expand_map

function. So, this function is going to be applied and you know on sentence, so **any any** any occurrences that are there, so we will use this sum method replace all contractions with their expansion for a sentence.

So function expand_map that we just discussed will be call for every non overlapping occurrence of the pattern. So, any occurrence of the pattern that we get using the pattern So, for that expand_map will be call and for that means will for any contraction will have the expansion and that is going to be use to replace the occurrences in the sentence that we have passed.

So, in this fashion finally will be able to written the expanded sentence where all the contractions would have been replaced with their appropriate expansions. So, with this let me run this, so that we have access to this user defined function. Now once we have done the next thing, now the next thing that we require us to use this function. So, first will take this test is a string here.

So in this you know double quotes I am using this string that young man was not fast enough and he could not win the race. So, in this case as you can see in the tester string that we have wasn’t is there which is a contraction and then couldn’t is there is a contraction. If you go back to the mapping here let us see whether we have covered these 2 wasn’t and couldn’t, if you go back to the mapping.

So, you can see that you know here let me scroll up bit more you can see wasn’t is covered here, wasn’t was not and if we scroll back couldn’t might also be I think it is covered there, you can see couldn’t could not. So, **t**hese 2 mappings, these 2 contractions are there in our you know our map that we had defined the dict object that we had defined. Now for this you know test string example that we have.

I think we would be able to replace these contractions with their appropriate expansion. So, the first line we are defining test and the next thing we are calling this our user defined function

that we just discussed expand_contractions. And we are passing on the first argument which is sentence, so in this case test and our contraction mapping that we are already defined.

So, if I done this in the output one third you can see you can compare it with the test string, the young man was not fast enough and he could not win the race. So, you can see the contractions you know they have been appropriately in this case they have been appropriately you know replaced with the expansions that we have there. Now let us do the same exercise with our corpus here.

So, emma is the corpus that we had used, so you know in the text that we have for emma we might not be sure whether we have the these contracts and present there. Because this is you know established corpus and it seems that former lighting mechanism has been used that is why to demonstrate this we had used this you know test string as an example. Because in the emma you know file that we have it is part of the within the corpus.

A former lighting mechanism has been used to maybe we might not have any contractions to be replaced with their appropriate expansions. But anyway you can always run this. So, for to demonstrate on a corpus level will use this you can see the code is same, so within the list comprehension. Now slight changes that in the corpus will have a number of sentences, test string there was just 1.

So, now we require a list comprehension here, so you can see we are calling expand_contraction, contractions you will find function in the transformation part of the you know list expression, within that sentence comma contraction on a scroll map, contraction and scroll map we already defined. Sentence is coming from you know for sentence in emma_f, so emma_f is the you know previous output that we had produced where you know we had you know cleaned you know few things there.

We had use the you know filters action, filter function there and we had clean this uuh text. So, we are going to use that particular you know output and will going to apply this expand underscore contraction function here. So, if I run this will have emma_fe which will actually

you know have the will have you know replace the all contractions with their appropriate expansions.

So if I have a look at first 500, now if we look at the output how about the only problem is that we do not know whether any you know contractions were presenting this is such a long text any you know contractions were present in any part of this text.

(Video Ends: 32:52)

However, you can always check it back, so you can always write your own code to first find out whether contractions are you know present or for that matter without special characters are present in a text before applying some of these things. So, both these things that we discussed removing a special characters and you know replacing the contractions with appropriate expansions whether they are present in the first place.

So, if they are present maybe only then we should be you know running that part of the code. So with this you would like to stop here in the next lecture will continue our discussion on this part and on some of the you know coming steps there, thank you.

Keywords: tokenization, contraction, text mining and modelling, extraction, errors.