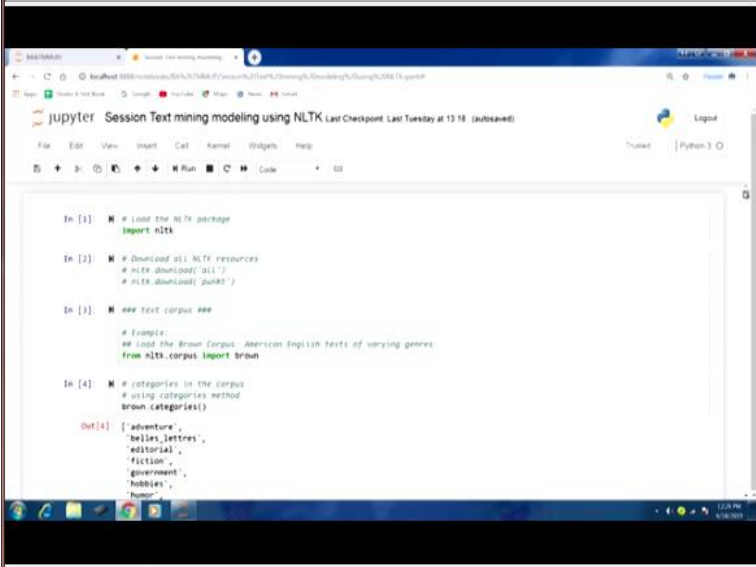


Business Analytics & Text Mining Modeling Using python
Prof. Gaurav Dixit
Department of Management Studies
Indian Institute of Technology Roorkee

Lecture-37
Text Collection And Transformation-Part II

Welcome to the course business analytics and text mining modeling using python. So, in previous lecture we had started about discussion on NLTK package that is typically used for a text mining modeling or you know text analytics and also in NLP natural language processing. So, what we will do we will go through we will do a small recap of few steps that we discussed in the previous lecture and then we will pick up from the point where we stopped in the previous one.

(Refer Slide Time: 00:54)



```
In [1]: # load the nltk package
import nltk

In [2]: # download all nltk resources
nltk.download('all')
# nltk.download('punkt')

In [3]: ### text corpus ###
# Example:
# load the Brown Corpus: American English texts of varying genres
from nltk.corpus import brown

In [4]: # categories in the corpus
# using categories method
brown.categories()

Out[4]: ['adventure',
         'beliefs_letters',
         'editorial',
         'fiction',
         'government',
         'hobbies',
         'humor']
```

So, let us start, so first thing as we discussed is you know nltk package, so first thing We need to import this into the patent environment. And nltk is typically install comes with the anaconda distribution, this aspect we discussed in the previous lecture. Then also you know certain NLTK sources we will be requiring as part of our discussion of this package and text mining modeling as well. So, this part we had done in the previous lecture itself you know commented those commands here.

(Video Starts: 01:26)

So let us move forward. So, next thing that we were able to cover in the previous lecture was the you know a brown corpus. And we went through some of the you know initial you know lines of code which we typically perform while trying to understand the corpus the kind of files that are there, kind of text that is there. So, let us quickly go through this part as well.

So, brown corpus part of NLTK corpus you know model, so we will import this model also. And then categories to typically if you just you know if you have a look at the kind of corpus you know this is a categories based you know text corpus. So, if you look at the categories in the corpus like we discussed in the previous lecture that you can think about a tree kind of structure and categories and files you know under those categories.

So, these are the categories we looked at the you know first 5 sentences also. So, you can see this is how, this is going to look like first 5 sentences, this is seems to be a list of a strings. We can always use join method that we learned before in our previous you know previous lectures. And we can have a nicer look off sentences there. Then you know if you are interested in you know text for a word you know particular category, so that also we can perform for that we can use the categories argument here.

So, we can run this also and have a look at the you know 5 sentences there and then if we understand in or finding out the files that are part of that corpus. So, for that we have this file IDs method that can be used. So, you can see this you know this particular method we can call on the corpus and we will have the list of files that are part of the corpus. So, you know if we are looking to find all the files or file IDs under a particular you know category in the corpus.

So, that also can be done in the file IDs method we can use this categories argument and we can find out the files that are part of that particular category, so you can see here again output. Now if you understand in a particular file let us say CA18 here. So, we can use the file IDs argument here in our brown sense, a sense method. We can specify the file ID and the text form that particular file it is going to be you know produced in the output.

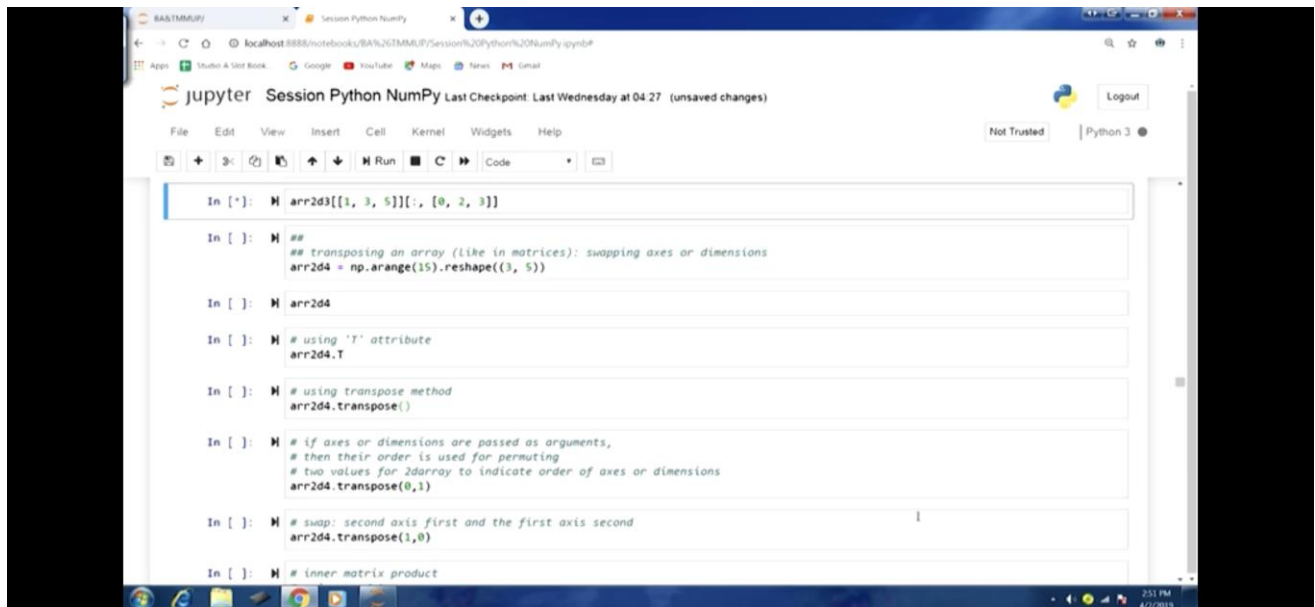
And then we can use the joint method to merge you know that those sentences. And let us have a look at 5 sentences from this file, so you can see this text how we are able to obtain this from a particular file in the corpus. Now we were able to discuss up to this much now we will pick up another corpus gutenber corpus and you know we will repeat some of the previous kind of you know lines of code and then we will move forward.

So, first thing let us load the Gutenberg corpus here, so from `nltk.corpus import gutenberg` here. So, we will have this then as we did for the brown corpus here also using the file IDs you know method we can have a list of the files that are there. So, if I run this, you can see can have a look at the file all text file, simple text format files. So, we talked about various file format for textual data. So, you can see in this particular corpus all these files that we can see they are in the simple text txt format.

Now let us move forward, now let us talk about raw method that is there, so this method can be used to return a single string for the file. So, all the textual content that is there in the file can be returned in a single string object, so let us pick up 1 file. So, we are picking here this `austen.austen-emma.txt` file out of the you know the list that we just saw in the previous output and we are using file IDs you know argument here for the raw method.

And we are passing on this file name and what it will do all the text content it will be you know defined in this single string object `emma`, so let me run this and we will have `emma`. So, let us have a look at the total number of characters that are part of this particular string object that we have just stated. Remember this is a string object is representing the whole content in this particular file `austen-emma.txt`.

(Refer Slide Time: 03:20)



```
In [*]: arr2d3[[1, 3, 5]][:, [0, 2, 3]]

In [ ]: ##
## transposing an array (like in matrices): swapping axes or dimensions
arr2d4 = np.arange(15).reshape((3, 5))

In [ ]: arr2d4

In [ ]: # using 'T' attribute
arr2d4.T

In [ ]: # using transpose method
arr2d4.transpose()

In [ ]: # if axes or dimensions are passed as arguments,
# then their order is used for permuting
# two values for 2darray to indicate order of axes or dimensions
arr2d4.transpose(0,1)

In [ ]: # swap: second axis first and the first axis second
arr2d4.transpose(1,0)

In [ ]: # inner matrix product
```

So, if I run this, so you can see that you know in the output 80 here, 887071 characters are part of this textual content that is there in this particular file here. Now we can have a look at because now this is a string object, so we can have a look at you know let us say first 200 characters in this particular file. So, emma we can use as we have learned we can use this brackets and in colon notation to actually access these characters here. So, if I run this you can see in the output 81 those 200 characters we are able to see, so you can see here.

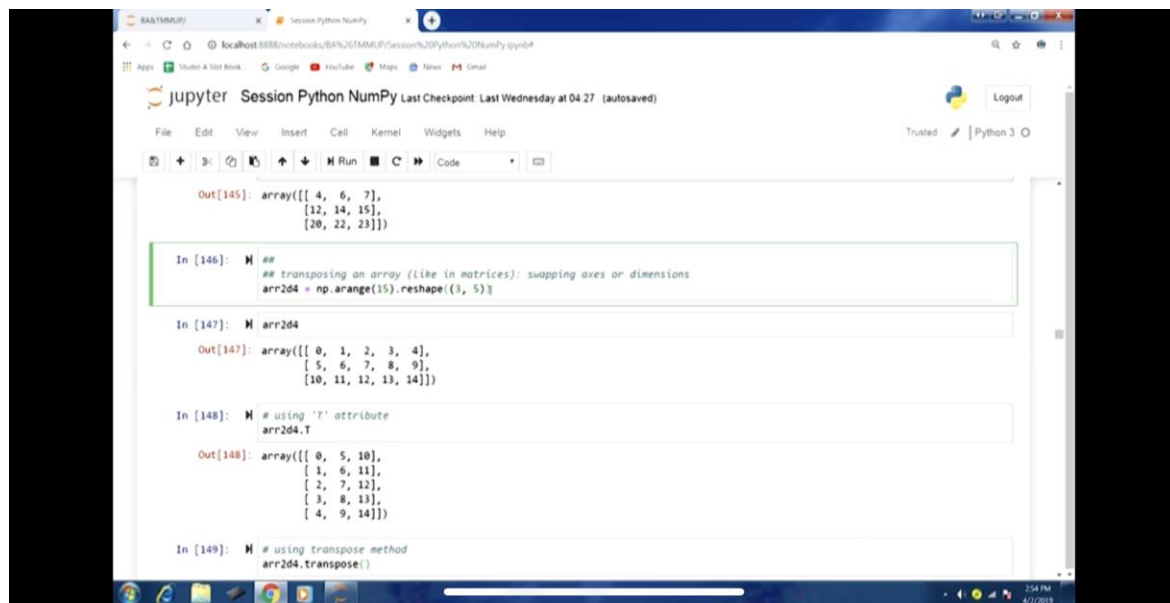
Now if you want to have a nicer output here, so you can always use this print function, so we can pass on this is string emma and brackets 0 to 200 to print function and we will be able to print a nicer output here you can see. Now this is more like a text format kind of outputs, so slightly better to understand the kind of content that is there. Now let us move to the another important aspect here.

So in previous lecture we talked about the tokenization as one of the important steps, we talked about certain other steps and we also discussed that order of education of those steps depend on the problem on the data. So, so will be discussing as some of these steps one by one and you know orders are typically as we said is you know going to be depending you know the kind of modeling that we are going to do.

So, we will first pick up the sentence tokenization, so typically as we discussed 2 types of tokenization are done sentence and word tokenization. So, we will start with the sentence tokenization, so this will do using different nltk interfaces, so different ways we can actually perform this sentence tokenization. So, we will take some of those examples here, so let us pick up first approach, so in this approach we use this sent-tokenize function to perform tokenization

So typically this is the default and a recommended sentence tokenization, so if you do not have any other peculiarity in your problem or data set that you are dealing with the text corpus that you are dealing with. Then you know you can always go with this recommended sentence tokenizer, so this is a particular tokenizer advantages with this one is this is a pre trained tokenizer on several language model not just English and it can work really well on many you know popular languages.

(Refer Slide Time:06:23)

A screenshot of a Jupyter Notebook interface. The browser address bar shows 'localhost:8888/notebooks/BA%20MBA/P/Session%20Python%20NumPy.ipynb#'. The Jupyter header includes 'jupyter Session Python NumPy' and a 'Logout' button. The notebook has a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for running, saving, and other actions. The code area shows four input cells. The first cell (In [145]) contains a NumPy array: `array([[4, 6, 7], [12, 14, 15], [20, 22, 23]])`. The second cell (In [146]) contains a comment and a line of code: `arr204 = np.arange(15).reshape((3, 5))`. The third cell (In [147]) contains a comment and a line of code: `arr204`. The fourth cell (In [148]) contains a comment and a line of code: `arr204.T`. The output of the fourth cell is a 5x4 array: `array([[0, 5, 10], [1, 6, 11], [2, 7, 12], [3, 8, 13], [4, 9, 14]])`. The fifth cell (In [149]) contains a comment and a line of code: `arr204.transpose()`. The output of the fifth cell is a 5x4 array: `array([[0, 5, 10], [1, 6, 11], [2, 7, 12], [3, 8, 13], [4, 9, 14]])`. The status bar at the bottom shows '2:41 PM 4/2/2019'.

Now one thing is important to understand here these that you know many of these libraries they are pretrained on European language English and other European language and other you know language coming from western countries. So, they might not work so well for Indian languages,

so let us move forward. So, `nltk.sent_tokenize`, so `emma` is the you know string object that we have there, so we will pass on this one and using this will get our sentence token in this.

So let me run this. Now let us have a look at the number of sentences here, so we can use the `length` function here and `emma_sense`. And you can see we have 7493 sentences here, let us have a look at the first 5 sentences, so again we can use `emma_sense` and we can use this you know brackets, colon notation, `0:5` and we will have the output here. Now here if you look at the output you know a few things you can notice here that the tokenizer the just now that we have use to produce these sentence tokens.

So, essentially as we talked about, we are you know about sentence tokenization is that breaking the text into sentences and in using word tokenization into words. So, till now what we are doing is we are breaking the textual content into sentences here, so if you look at the output that is there in this output 85 typically, the tokenizer it will not just use periods to limit sentences.

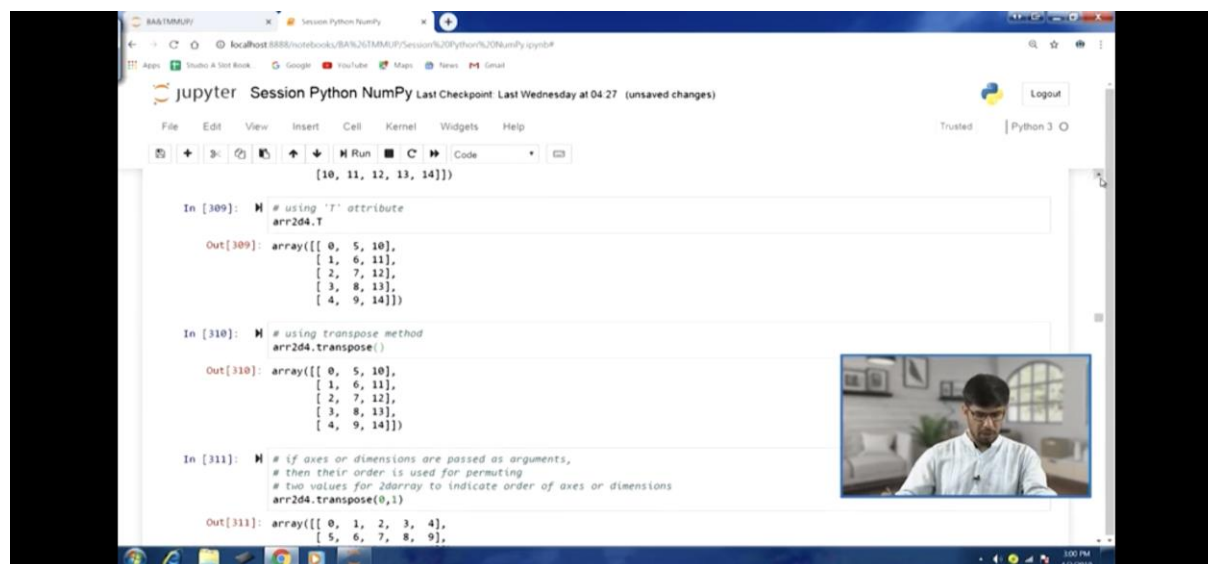
It will also consider other punctuation and the capitalization of words also, because capitalization of words you know any new sentences supposed to be starting with a you know capitalized word first character of the first word in a sentence is going to be in the capitalized form. So, that can also be used by this tokenizer and not just the periods other punctuation marks can also be used to tokenize sentences.

So, if you look at this first sentence it starts by starts like `emma by, jane, austen` and you can see and now you can see the second sentence starts with `she was the youngest of the 2 daughters` and the third sentence `her mother\nhad died`. So, in full sentence 16 years had and then fixed sentence between you know them. So, if you really look at here then you know different you know if you look at the full output then you will find the many places the capitalization another things have been use you can see.

All these sentences the first character of the first word in all these sentences in the you know is capitalized there. So, that is also there and dot is also there at the end of each of these sentences

mostly. So, you can see that all these things are being used by tokenizer could produce these you know sentence tokens. So, this is first approach. Now let us talk about the second approach here.

(Refer Slide Time: 11:51)

A screenshot of a Jupyter Notebook interface. The browser address bar shows a local host URL. The notebook title is "Session Python NumPy". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for running, saving, and other actions. The code area shows three input cells. The first cell (In [309]) uses the 'T' attribute to transpose a 2D array 'arr2d4', resulting in a 4x5 array. The second cell (In [310]) uses the 'transpose()' method on 'arr2d4', resulting in the same 4x5 array. The third cell (In [311]) uses 'arr2d4.transpose(0,1)' to permute the axes, resulting in a 5x4 array. A small video inset in the bottom right corner shows a person speaking. The system tray at the bottom indicates the time is 1:00 PM on 6/2/2019.

```
[10, 11, 12, 13, 14]]

In [309]: M # using 'T' attribute
arr2d4.T

Out[309]: array([[ 0,  5, 10],
 [ 1,  6, 11],
 [ 2,  7, 12],
 [ 3,  8, 13],
 [ 4,  9, 14]])

In [310]: M # using transpose method
arr2d4.transpose()

Out[310]: array([[ 0,  5, 10],
 [ 1,  6, 11],
 [ 2,  7, 12],
 [ 3,  8, 13],
 [ 4,  9, 14]])

In [311]: M # if axes or dimensions are passed as arguments,
# then their order is used for permuting
# two values for 2darray to indicate order of axes or dimensions
arr2d4.transpose(0,1)

Out[311]: array([[ 0,  1,  2,  3,  4],
 [ 5,  6,  7,  8,  9]])
```

So in this approach, we use this punkt sentence tokenizer class to actually perform tokenization. So, the previous function that we just used send_tokenize function that is also an instance of this class punkt_send_tokenizer. So, instead of using this you know pre-trained or other well trained sent_tokenize a send_tokenize function. We can have we can create another instance of this class and maybe use that.

So this could be another approach. So, for this you know will you know initialize another instance here. So, punkt_st we use this is for we are working with for sentence tokenization nltk.tokenize and this class comes sentence tokenizer. So, we will have this instance here, so let me run this and then you know we can call this instance punkt_st and we can use this tokenizer in the function here pass on emma.

And we will have our tokenization performed here, so we will have emma_sents2. Now here also we can again have a look at the output for a first 5 sentences, so emma_sents2 0 to 5 within brackets 0 power 5 and you can see the output here. You can also compare the output here you

would see that main output is similar to the output 1. That is using the default you know sentence tokenizer function there might be slight differences here.

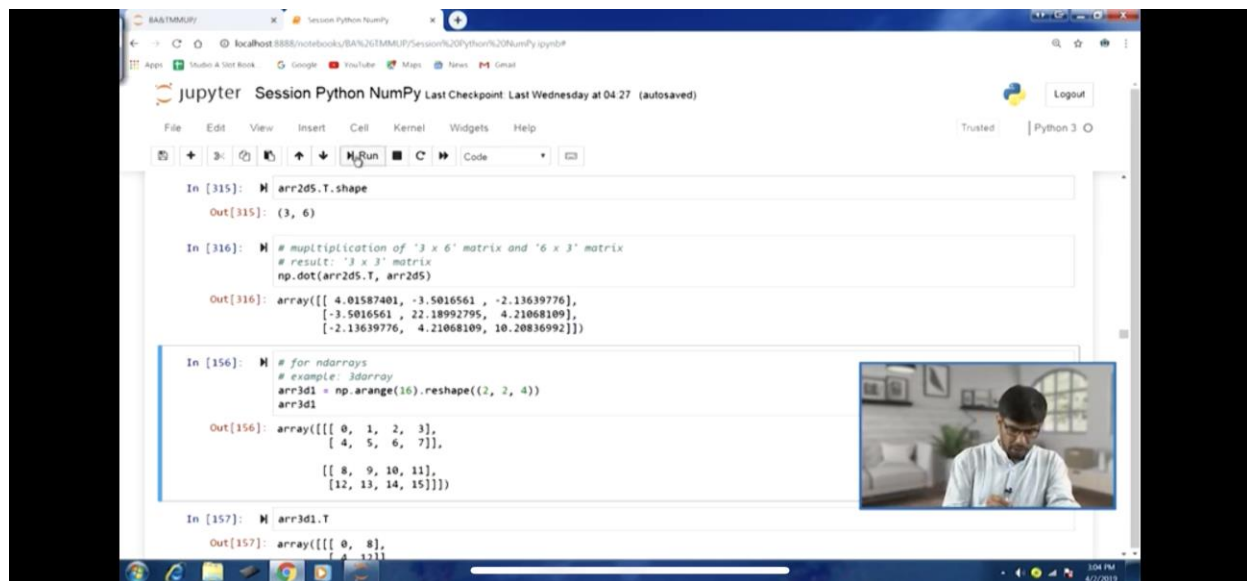
If you just compare you can see this that sentence starting of sentence for few sentences slightly different, I can see 16 years is whole sentence there and output 85 and between them this first sentence few come here then you know slightly different thing 16 years there then wood houses family. So, the fifth sentence actually starting from you know slightly different there wood houses family.

So maybe some punctuation mark has been used differently. Because the other one the other tokenizer was pretrained and so it might work differently. So, there you can see Mr. Woodhouse family here so you know from that capitalization maybe the default you know tokenize function tokenize method has picked up and perform the sentence tokenization.

So, you can see output is more or less similar but there are certain slight differences that you can you know notice here. And the differences is also mainly as I said is because the default you know tokenizer is actually pretrained, so it is you know it is working differently than the class and if you take the class and if we take the class data instance and use the method there.

So, let us talk about the next applause that is third applause for sentence tokenization here. So, in this we can use an instance of regextokenizer class. So, this is based on regular expression. So, we have a touched bit upon regular expression in previous lectures. So here we would be specifying a particular you know regular expression and we will use that to you know find out patterns and that is going to be used to create you know a sentence tokens here.

(Refer Slide Time: 15:25)



```
jupyter Session Python NumPy Last Checkpoint: Last Wednesday at 04:27 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run
In [315]: arr2d5.T.shape
Out[315]: (3, 6)

In [316]: # multiplication of '3 x 6' matrix and '6 x 3' matrix
# result: '3 x 3' matrix
np.dot(arr2d5.T, arr2d5)

Out[316]: array([[ 4.01587401, -3.5016561, -2.13639776],
 [-3.5016561, 22.18992795,  4.21068109],
 [-2.13639776,  4.21068109, 10.20836992]])

In [356]: # for ndarrays
# example: 3darray
arr3d1 = np.arange(16).reshape((2, 2, 4))
arr3d1

Out[356]: array([[[[ 0,  1,  2,  3],
 [ 4,  5,  6,  7]],
 [[ 8,  9, 10, 11],
 [12, 13, 14, 15]]]])

In [357]: arr3d1.T

Out[357]: array([[[[ 0,  8],
 [ 4, 12]]],
 [[ 1,  9],
 [ 5, 13]]],
 [[ 2, 10],
 [ 6, 14]]],
 [[ 3, 11],
 [ 7, 15]]])
```

So, we are going to be using an instance of regex tokenizer class and a specific regular expression based patterns are going to be used, you can have a look at this pattern here. So, this pattern is you know mainly to capture you know any sentence and different components there could be different components of a sentence. So, you can see within parentheses we are trying to capture you know some of those aspects here also.

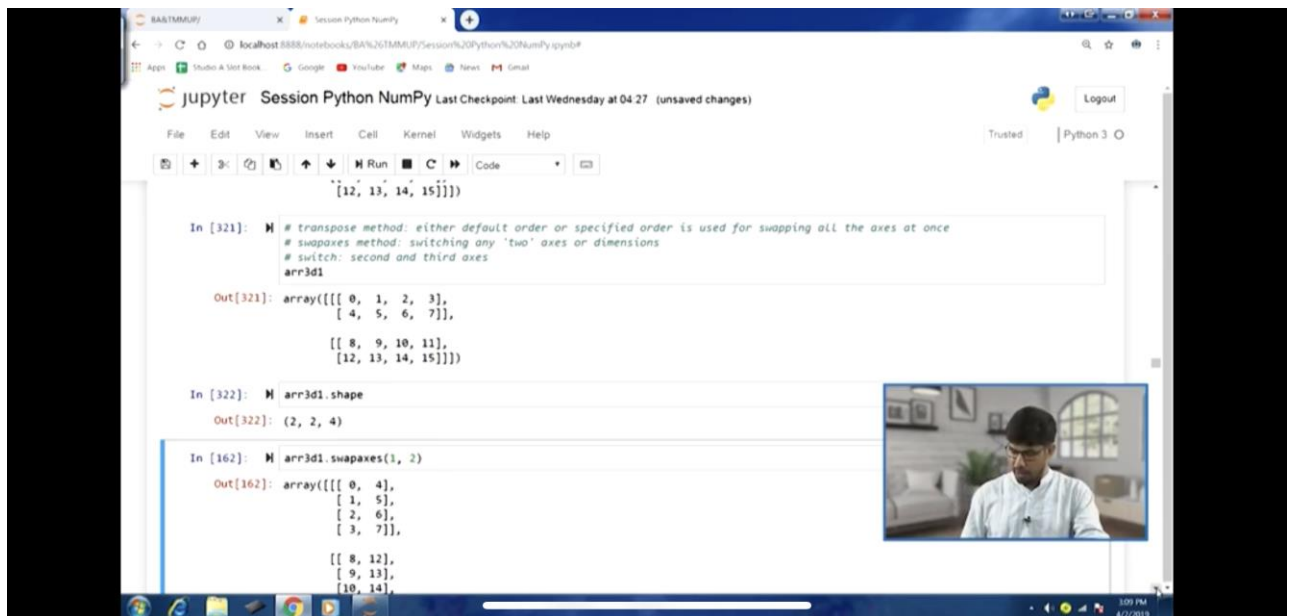
Overall this pattern is about you know identifying a you know sentences there, so it this pattern would be able to you know select, able to match most of the you know sentences using this you know pattern. So, let me run this, we will have this pattern 1, now we can use this regexp pattern for building the tokenizer and here you would see in the function that we are going to call here tokenize.regexp regextokenizer.

Here we have 2 augments pattern and gaps, so pattern we have already defined so we will use the pattern 1 and gaps is true. So, if this gaps is true then we will this particular you know method is going this particular function is going to use this to find gaps between the tokens. So, let me create this instance of this regexp object here. So, we got a class, we got an instance of this class regexp object here.

Now we can use this `regexp_st` that we have just created and call this `tokenize` method here. We are we will pass on `emma` the same you know text here and we will have the tokenization in `emma_sents3`. So, let me run this and let us have a look at first 5 sentences. Now you can again compare the output now you can see that first sentence you know starting you know if you look at the you know character by character .

Overall the output is similar to the you know output 1 and previous output as well. However, if you look character by character slight differences are going to be there. So you can see there `emma` started you know with brackets to that thing is there still. If you look at you know second sentence you can see before in the second sentence before `cv` we have the `shalsh` also here but in the previous output we did not have that.

(Refer Slide Time: 20:27)



```
[12, 13, 14, 15]]])

In [321]: # transpose method: either default order or specified order is used for swapping all the axes at once
# swapaxes method: switching any "two" axes or dimensions
# switch: second and third axes
arr3d1

Out[321]: array([[[ 0, 1, 2, 3],
                  [ 4, 5, 6, 7]],
                [[ 8, 9, 10, 11],
                 [12, 13, 14, 15]]])

In [322]: # arr3d1.shape
Out[322]: (2, 2, 4)

In [162]: # arr3d1.swapaxes(1, 2)
Out[162]: array([[[ 0, 4],
                  [ 1, 5],
                  [ 2, 6],
                  [ 3, 7]],
                [[ 8, 12],
                 [ 9, 13],
                 [10, 14],
```

And we go back further and in the default tokenizer output also it started with `c`. So, we can see that slight differences here because of the approach certain output certain changes certain differences would always be visible there, you can see in next line also her mother before her mother there is a space hanging out there and the 16 years again that is prefixed by slashen and `so`.

So, because of the we are using a pattern to match the sentences, so these kind of slight changes are going to be there in this however this is another approach to create sentence tokens. Now let us move forward, now a next thing that we are going to talk about is word tokenization. So, for word tokenization as well will be using a number of analytic interfaces, so we will be talking about a few approaches here.

So, let us start with the first approach, so in this also for word tokenization also we have a default and recommended word tokenizer, so that is `wild underscore tokenize` function, so that is typically used to perform word tokenization. So, for what tokenization as well we can call this function `word_tokenize` and we can pass on this string object `emma` here and we will have the word tokens `emma_words`.

So, let me run this default tokenization for words and let us have a look at the number of words here we can use the `length` function here `length` and within parentheses `emma_words`. And you can see in the output 94 that 1 we have 1, 91,785 words here, so those many words are there in this word tokenization output that we just perform. If you understand looking at let us say first 500 words.

So let me run this `emma_words` on within brackets `0:500` and we will have you can see the output now, this is word tokenization So, you can see different words we are able to see in the sentence tokenization we had the you know sentences there, now this time we have the words here. So, you can see even the you know first you know opening bracket is also being treated as a word token.

Then the next one is `emma`, then `by`, then `Jane`, then `austen 1816`, then again closing bracket here. So, in this fashion you can see after `woodhouse`, we have `comma` here. So, `comma` is also being you know treated as a token here then again `comma` and all those instances are insensible `comma` is going to be repeating. So, you can see the kind of output that we are able to obtain by using the default you know in a word tokenizer.

So, even the different special characters and punctuation marks they are also being treated as a you know tokens here. Now let us talk about the second approach, so here for the second approach, we can use this treebank word tokenizer class to perform tokenization. So, this particular class is actually based on the pen tree bank and it internally it is uses various regular expansions to tokenize the text.

And in this you know 1 primary assumption is that the sentence tokenization has already been performed, so actually it is going to be typically used to tokenize the sentences into words. So, let us take for an example let us take the first sentence that he had obtained by applying the you know default sentence tokenizer. So, we will take emma_sense 0, so let me run this will create an instance of the treebank tokenizer, tree bank word tokenizer and also this is the sentence that will be you know that we can take for applying this.

Let us have a look at emma_sense1 also this is the second sentence that we have. Now we can use the tokenize method here also, we can call Treebank_wt the instance the class instant that we just created and call the tokenized function here emma_sense0 and if I run this will have it will written word tokens. So, you can see , now we can have a look at the let us say first 500 words here, so if I run this I can see here now similar to output 2.

So, you can see in this also output 99 also first token is opening bracket then emma, then by, then Jane, then Austen, 1816 and then closing brackets, you can also see is commas there also. So, similar kind of output that we obtain the differences that this is an instance of a different you know treebank word tokenizer class and you know it is typically use to convert sentences into word tokens.

So, few more things that this tokenizer this splits and separates into independent tokens some of these things periods of reading at the end of sentences. So, in the default tokenizer that might not be there in case you will also find you know periods appearing as tokenizer you know if you just, scroll down. In this output also you might find you can see semi colons we can already see colon dot is also there.

So, periods as independent you know tokenizer you know we can obtain using this tokenizer commas and single quotes when followed wide spaces. So, some of these tokens are common with the previous output as well, but this particular tokenizer will do it also, most punctuation characters that might not be the case in the default you know tokenize your words with a standard characters. So, they can also be the tokenizers, standard contractions, for example, don't to do and n't.

So, do not might be you know broken into these 2 words and these 2 tokens might be there. So, slightly different behavior, so mostly output is going to be same but few differences might be there. Now let us talk about the third approach, now this is an instance of `regex exp tokenizer` class. And here we use a specific regular expression based patterns and so let us take an example `pattern2` identify tokens.

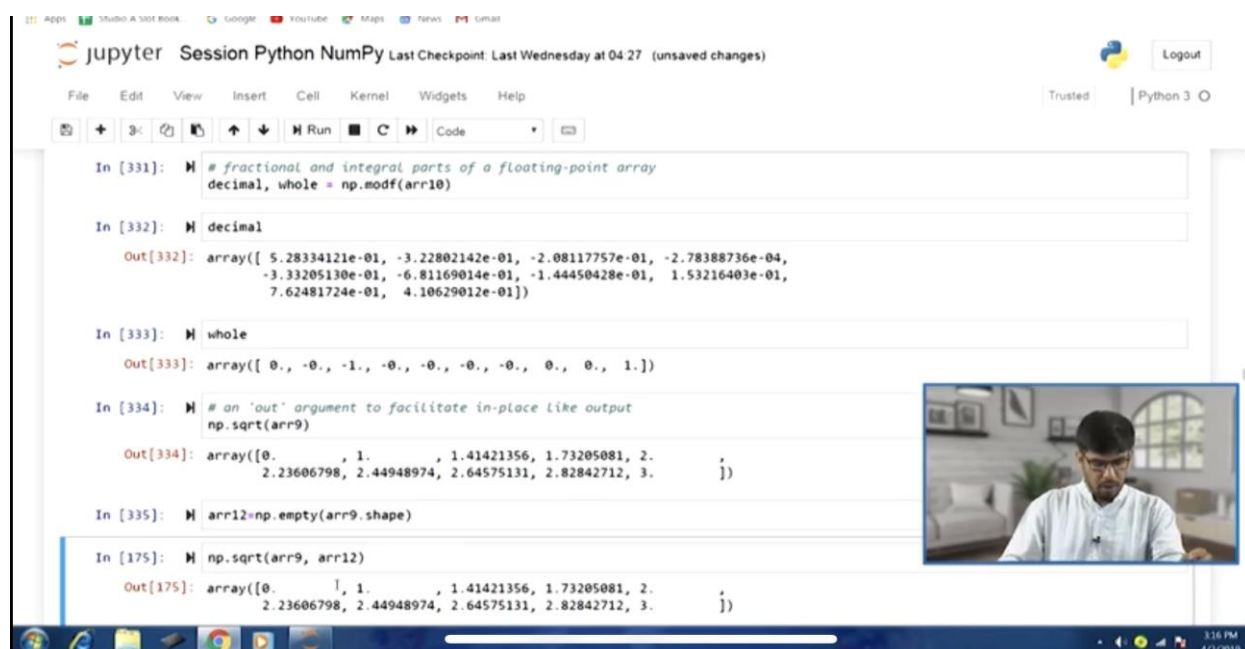
So, here we are defining this pattern to you know `backslash w+`, so this is something that we are going to use to identify words from the text that we have. So, let us first create you know this `pattern 2` object here, so let me run this and then using this `pattern2` and you can see `gaps` is false and will be creating this `regex_wt` object. So, let me run this and from this will be getting this `emma_words3`.

So we will tokenize will call `regex_wt.tokenize` and will have passed this `emma` and this will written as the tokens using the particular `regex` object and therefore the pattern. So, if I run this and let us have a look at first 500 tokens here, so you can see here output, so output is similar but slight differences, you can notice immediately. For example the earlier 2 outputs we had opening brackets also as a token, in this case that is gone.

Because of the you know regular excavation that we have used, so we did not consider that as a token, so that is gone here, you can also see certain other punctuation marks which were you know treated as tokens in the previous2 approaches, now they are also gone. So, depending on, so these are different you know approaches that can be really useful depending on the problem that we are dealing with and the kind of text corpus we might have.

So, you can clearly see that in this particular example you know typically meaningful words are there as tokens. Now we can have another pattern, pattern 3 where this one we can use to identify gaps. So, again let me define this pattern object pattern 3. And we will have another rejects object here `regex_wt1` and we will call this you know this class function `regex_tokenizer` and `pattern3` and `gaps2` in this case, I run this we can use now this one to tokenize `emma`.

(Refer Slide Time: 27:41)



```

In [331]: # fractional and integral parts of a floating-point array
          decimal, whole = np.modf(arr10)

In [332]: decimal

Out[332]: array([ 5.28334121e-01, -3.22802142e-01, -2.08117757e-01, -2.78388736e-04,
                 -3.33205130e-01, -6.81169014e-01, -1.44450428e-01,  1.53216403e-01,
                 7.62481724e-01,  4.10629012e-01])

In [333]: whole

Out[333]: array([ 0., -0., -1., -0., -0., -0., -0.,  0.,  0.,  1.])

In [334]: # an 'out' argument to facilitate in-place like output
          np.sqrt(arr9)

Out[334]: array([0.,  1.,  1.41421356,  1.73205081,  2.,
                 2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.])

In [335]: arr12=np.empty(arr9.shape)

In [175]: np.sqrt(arr9, arr12)

Out[175]: array([0.,  1.,  1.41421356,  1.73205081,  2.,
                 2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.])

```

And let me run this we will have `emma` words, let us have a look at the first 500 you know let us say tokens here. So, this is a if you look at this output and compared with the previous one, now you can see earlier in the previous one we did not have opening brackets as the token, now here opening bracket has been combined with `emma` here. So, that is because of the you know the pattern you know that the change the pattern.

And therefore the output in terms of what tokens that we are getting that is also you know changed. So, this is another approach, so let us move forward, now in this we can also using the previous output that we had `regex_wt1`, we can also get a start and end in dices here of each token. And so for this we can use this `span_tokenize` method and of the we can you know call

on this regex object `regex_wt1.span_tokenize` and pass on `emma` and we will have the start and end indices of each of those tokens.

So, let us say `clever`, start and end indices and end index value per `c` and start index value for `r`, so in this fashion for all of those tokens will get those start and end indices. So, let me run this and let us have a look at those start and end indices you can see 0 5 6 8, so in a sense you are you know it will get the kind of idea the kind of number of characters that are part of those tokens. And you can use this also to construct the tokens here again or even modify the tokens here again.

So, we can use this list to comprehension here `emma[start:end]` and for a start comma end in what in end indices. So, in this fashion will be able to produce the you know kind of similar output here that we had. So, using a start and end indices also you can clearly see that we have been able to produce the same output. Because it is coming from the same approach only the different functions, combination of functions and code that we are using here, so you can see.

Previous output that we had here the tokens you can see first token I might had the opening brackets also. So, in this case that we have using start and end indices here also you can get this similar.

(Video Ends: 30:52)

So, we will like to stop here and in the next lecture we will continue to discuss about word tokenization using an another approach fourth approach, so let us stop, thank you.

Keywords: Natural Language Processing, expression, string, sentence tokenization, object, array.