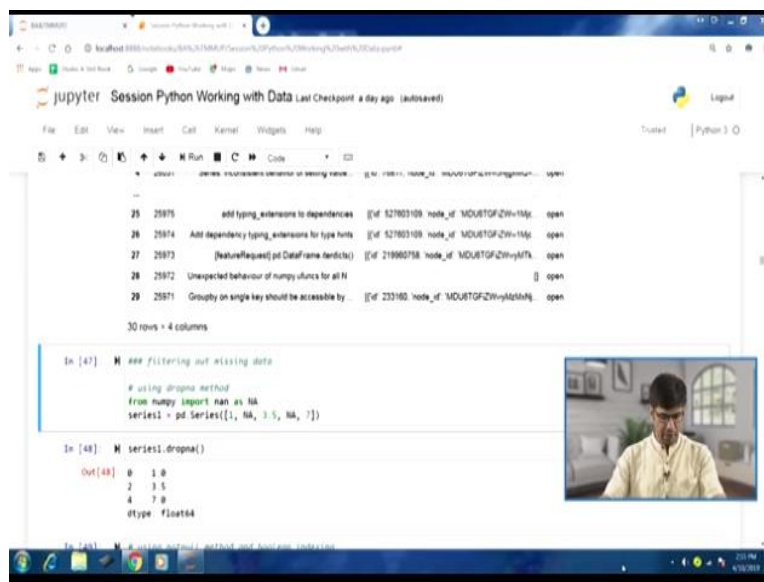


Business Analytics And Text Mining Modeling Using python
Prof. Gaurav Dixit
Department of Management Studies
Indian Institute of Technology Roorkee

Lecture-31
Python Working with Data–Part II

Welcome to the course business analytics and text mining modeling using python. So, in previous few lectures we have a started about discussion on how to work with data using python platform. And specifically we were able to cover how to work with CSV file, Microsoft, Excel files and web APIs as well. Now we will move to certain other aspects of working with data.

(Refer Slide Time: 00:48)



```
...
25 25975 add typing_extensions to dependencies [(of 52703109, node_id 'MDUETGFZMw=156p, open
26 25974 Add dependency typing_extensions for type hints [(of 52703109, node_id 'MDUETGFZMw=156p, open
27 25973 (featureRequest)(pd.DataFrame.ardicta) [(of 219980758, node_id 'MDUETGFZMw=156p, open
28 25972 Unexpected behaviour of numpy.ufuncs for all N [(of 233160, node_id 'MDUETGFZMw=156p, open
29 25971 Groupby on single key should be accessible by ... [(of 233160, node_id 'MDUETGFZMw=156p, open

30 rows x 4 columns

In [47]: # filtering out missing data
# using dropna method
from numpy import nan as NA
series1 = pd.Series([1, NA, 3.5, NA, 7])

In [48]: series1.dropna()
Out[48]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

So, let us start with filtering out missing data, so we have certain methods available in various packages in python which can be really useful in this. Certain examples, certain aspects we have covering in previous lectures as well. Now here in this data management context will go through some of the you know new you know aspects as well.

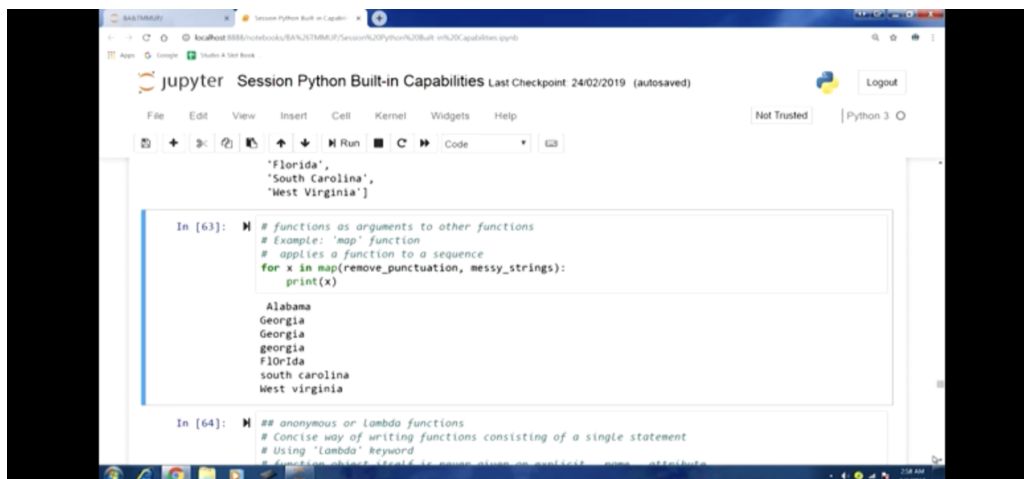
So, let us talk about filtering out missing data, so first thing using dropna method, so will need to you know import you know the required library modules here. So, first line is about that, now you can see there you know that we are specifying you know nan as NA and you can see this. So, later on wherever we are you know wherever nan is there it can be used as NA there. So, let us take this example of the series 1, so we have this where using the series function to create this

series object here, these 5 values, so let me run this.

(Video Starts: 01:52)

And if I want to drop all the NAs that are present in this series then I can call this dropna method here. And out of the 5 elements that we have in the series 1 object, 2 are NA elements, 2 are NAs, so those would be drop. So, if I run this and you can see in the output that only 3 elements with the indices 0 to 4 are there, so therefore the remaining 2 you know elements with NA values they have been dropped.

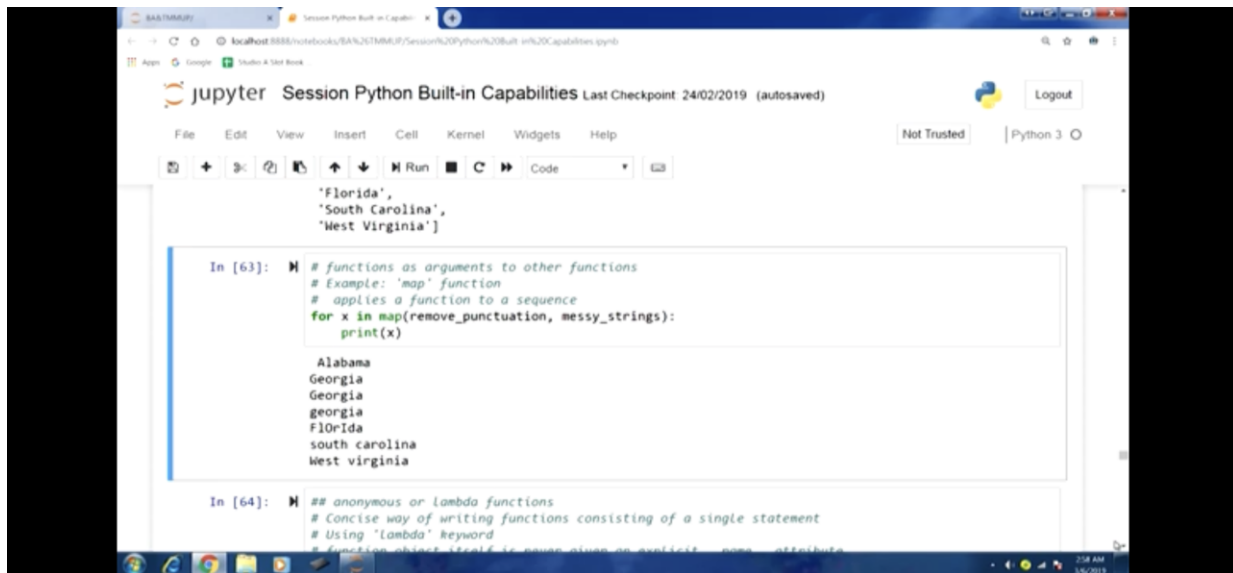
(Refer Slide Time:03:25)

A screenshot of a Jupyter Notebook interface. The top bar shows 'jupyter Session Python Built-in Capabilities Last Checkpoint: 24/02/2019 (autosaved)' and a 'Logout' button. Below the menu bar, there's a toolbar with icons for file operations, running, and code execution. The main area contains two code cells. The first cell, labeled 'In [63]:', contains a comment '# functions as arguments to other functions', an example of a 'map' function, and a loop that applies 'remove_punctuation' to a list of state names. The output of this cell shows the state names with punctuation removed. The second cell, labeled 'In [64]:', contains a comment about anonymous or lambda functions and a brief explanation of the 'lambda' keyword.

```
'Florida',  
'South Carolina',  
'West Virginia']  
  
In [63]: # functions as arguments to other functions  
# Example: 'map' function  
# applies a function to a sequence  
for x in map(remove_punctuation, messy_strings):  
    print(x)  
  
Alabama  
Georgia  
Georgia  
Florida  
south carolina  
West virginia  
  
In [64]: ## anonymous or lambda functions  
# Concise way of writing functions consisting of a single statement  
# Using 'lambda' keyword  
# function object itself is passed along as explicit argument
```

Now there could be another approach to achieve the same thing dropping NAs, in this approach we can use the notnull method and combine it with the boolean indexing. So, you can see in this next line of code in the series 1 within bracket we are passing an expression which is actually call to this method notnull. So, series 1 dot notnull, so you will get a boolean output here which will indicate you know which will indicate which of the elements are notnull those would be true.

(Refer Slide Time: 07:30)



The screenshot shows a Jupyter Notebook titled "Session Python Built-in Capabilities". The top bar indicates the last checkpoint was on 24/02/2019. 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 main area contains two code cells. The first cell, labeled "In [63]:", contains a list of state names: 'Florida', 'South Carolina', and 'West Virginia'. The second cell, labeled "In [64]:", contains a code snippet that defines a function to remove punctuation and then applies it to the list of state names using the map function. The output of the second cell shows the state names with punctuation removed: Alabama, Georgia, Georgia, Florida, south carolina, and west virginia.

```
'Florida',  
'South Carolina',  
'West Virginia']  
  
In [63]:  
# functions as arguments to other functions  
# Example: 'map' function  
# applies a function to a sequence  
for x in map(remove_punctuation, messy_strings):  
    print(x)  
  
Alabama  
Georgia  
Georgia  
Florida  
south carolina  
west virginia  
  
In [64]:  
## anonymous or lambda functions  
# Concise way of writing functions consisting of a single statement  
# Using 'lambda' keyword  
# function object itself is never given an explicit name attribute
```

So, only those would be kept in the output, so if I run this we would be able to achieve the same output. So, if we compare output 183 and 184 they are same, so these are 2 ways either dropna method or using notnull method and Boolean indexing to achieve this thing you know dropping NAs. Now let us take the scenario when we are dealing the data frame, so let us take create this data frame dfi and let us let me run this.

So, again here again in this case also in this data frame we have lots of nam and we can use dropna method again to drop you know those values. So, if I dfi dot dropna and you can see I am left with just 1 row because most of the other rows they had nam. So, therefore all of them have been you know drop, so when we call in this fashion dropna any row have been nam is going to be you know dropped.

Now we can also have few variations of dropna method where we can specify that drop only those rows where all the elements are NAs. So, for that we can use this how argument, so in the dropna method we can pass on how argument the keyword argument it is a keyword argument and we can specify all. So, only the rows which have all the NAs only those are going to be dropped.

So, if we look at the output we had just a 1 row with the index 2 that has been dropped because it had all the NAMs. Now just like dropping rows we can also drop columns, so for that we will have to specify axis as 1. So, let us you know create let us create another you know column here df 5 and 3 so will column fill with all the values will be 8, so let us create this column.

And now we can go ahead and drop na and axis 1, so all the columns with the na values they would be dropped. So, in this case will be left with just 1 column which we have just now added, so if we look at the output 190. Then you can see that just 1 column has been you know is left there, now let us add another column, now this one is the na column here.

(Refer Slide Time: 10:09)

The screenshot shows a Jupyter Notebook window titled "Session Python Built-in Capabilities". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running code, and other notebook functions. The code area contains two input cells:

```
In [64]: ## anonymous or lambda functions
# Concise way of writing functions consisting of a single statement
# Using 'lambda' keyword
# function object itself is never given an explicit __name__ attribute
# convenient when functions as arguments are used (data processing context)
# becomes feasible to write and apply a custom operator
def short_f(x):
    return x * 10

ret2 = lambda x: x * 10

In [69]: # Example:
def apply_short_f(alist, short_f):
    return [short_f(x) for x in alist]
```

Below the code, the output of the first cell is visible, showing a list of US states: "utah", "georgia", "Florida", "south carolina", and "West virginia".

So, if I assign df14=na then will have another column here, now if I want to drop you know this column here. So, in this case we are demonstrating the column which is having all the nam values. So, again we can use the how argument and specify all and therefore the column the only column with all the you know na values only they would be dropped. So, we can call this method in this fashion if I run this you can see that column with the column index 4 is gone there.

Now let us move to the few other aspects, now filtering, filtering out rows with less than a specified number of columns without na values. So, this kind of you know filtering is also

possible, so especially in the kind of data where time is quite important. And we might have this kind of data frame df6, so let me first create this data frame.

And you can see in the output 193 we have 3 columns 0, 1, 2 and in the column 1 and 2 column with index 1 and 2 lot more nans. So, if you want to specify if you want to filter the row here, so focus is on rows we would like to filter rows with less than a specified number of columns without na values. So, if in this case let us take you know let us take the rows where at least you know the columns without na values should be at least 2.

So, in that case how do I implement this for this we have this thresh argument, so in the dropna method we can specify thresh=2. So, in this output 193 let us see which of the rows are going to be dropped. So, if you compare you can see the row number 0 and row number 1 there if we these are the rows where we have 2 nans. So, therefore will be left with just 1 you know will be just left with the 1 column without nan.

So, therefore those rows has been dropped because out of you know only 2 columns as per the thresh argument we have a specified, 2 columns should not have nan values. So, 2 columns should not nan values, so therefore the first and second row would be dropped. Now let us move to the next aspect, this is about filling in the missing data, so for this we can use fillna method.

So, this method can really use to replace NAs with zeros similar exercise we have done before also. So, here in this context we will do it again, so you can see where calling for the data frame df6.fill na and specifying the value there 0 which is going to replace any NAs that are there in the df6. So, if I run this you can see in the column 1 and column 2 initial few values where we had nans, now they have been replaced with 0s.

Now instead of replacing with just 1 value we can also have a dict object which can indicate you know different replacement value for different column. So, let us say column 1 we would like to replace nans with 0.5 and column 2 will like to replace nans with 0. So, we can specify that using

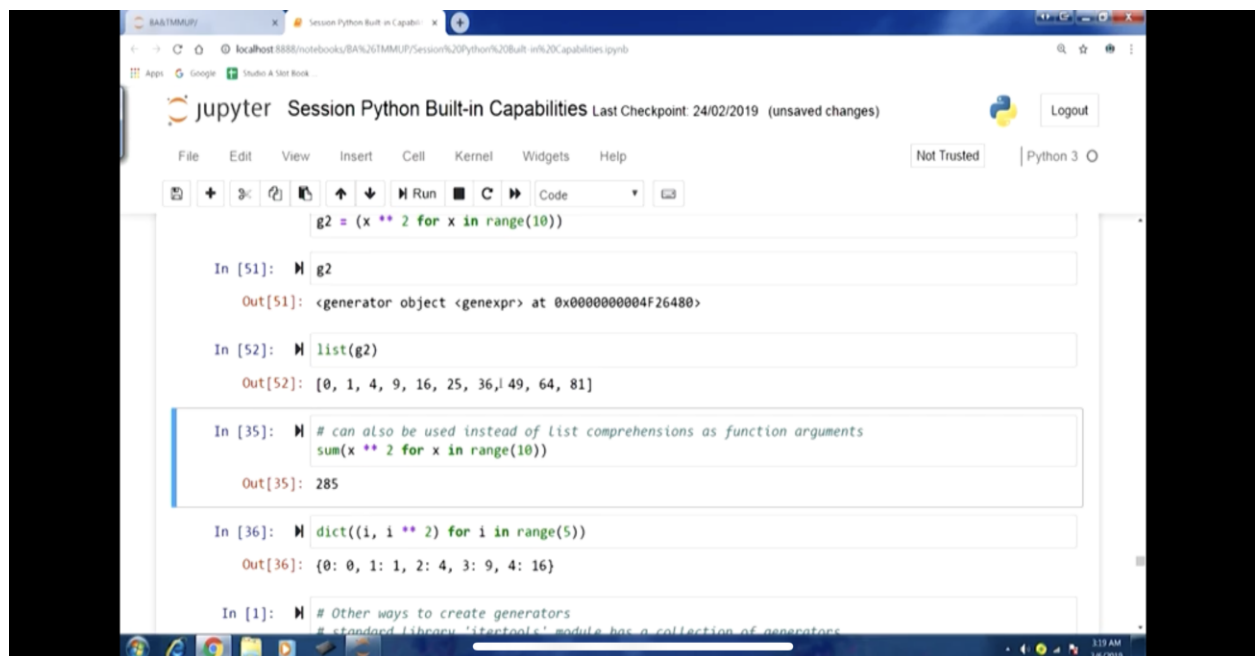
a dict object, so if you look at this line of code `df6.fillna` and within that parenthesis we have a dict object for column 1.

We are we have been indicated the value 0.5 for column 2, the value is 0, so if I run this and if you look at the output here 196 column 1 and column 2. So, column 1 the you know the first 4 rows we had nans and all of them have been replaced with 0.5 and you know column with index we had 2 nans at the top of this column and they have been replace with 0.

So, in this fashion where different columns using a dict object we can you know replaced with different values. Now if you want to do inplace replacement in the data frame itself, so for that we have `inplace` argument. So, that can be use, so let us have a look at this particular line of code here on the left hand side I have underscore because this is `inplace`.

This is to indicate that this is a `inplace` replacement you know `inplace` modification that we are going to perform, then we are calling `df6.fillna` and we would like to you know replace all the nas with 0 and `inplace` is `true`. So, the changes could have reflected in the `df6` data frame itself, so if I run this if you look have a look at the `df6` and now in this case `df6` itself has been modified, so this is `inplace` replacement.

(Refer Slide Time: 17:07)



```
g2 = (x ** 2 for x in range(10))

In [51]: g2
Out[51]: <generator object <genexpr> at 0x000000004F26480>

In [52]: list(g2)
Out[52]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [35]: # can also be used instead of list comprehensions as function arguments
sum(x ** 2 for x in range(10))
Out[35]: 285

In [36]: dict((i, i ** 2) for i in range(5))
Out[36]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

In [1]: # Other ways to create generators
# standard library "itertools" module has a collection of generators
```

Similar examples we have demonstrated in previous lectures as well. Now let us talk about another aspect which is interpolation method. So, apart from filling na values with you know by specifying a certain value or using a dict object to for different column, different values. So, that kind of arrangement we have another kind of arrangement here.

We can use interpolation method like ffill and fill these you know fill some of these values. So, let us have a look at this data frame df7 and let me run some of these lines and you can have a look at the data frame here. So, if you look at column with index 1 and column with index 2 here, so bottom values they are filled with nans. So, we can actually use the method ffill to actually fill these value.

So, ffill is something that we have used before, so it is a kind of interpolation method. So, in the fillna method itself we can we have an argument method for actually you know using these kind of methods to fill values, fillna values. So, in this case we are specifying ffill, so if I run this you can have a look at the you know column with index 1 and column with index 2.

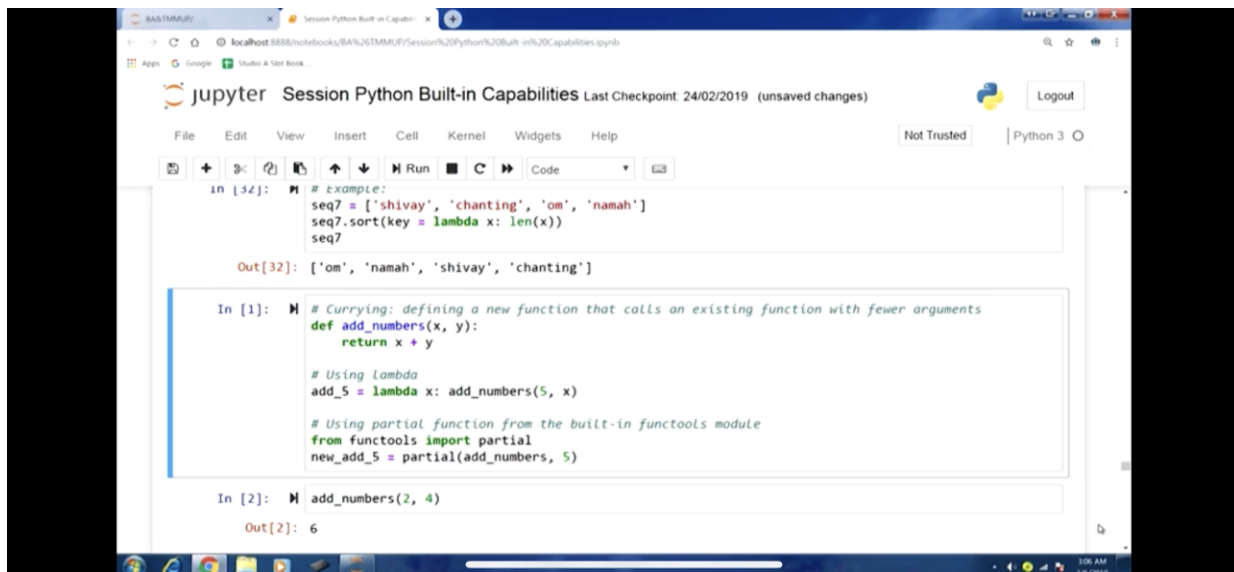
The bottom values where we had nans they have been replaced with the last value that we had. So, the you know so that kind of interpolation has been applied in this particular case. Now we can also limit this interpolation that we just here, for that we can use another argument which is limit and we can specify the number of times this interpolation using ffill can actually be used.

So, we can specify this, so if I run this again you can see that only twice this interpolation is happened in column with the index 1 and column with the index 2. And therefore in the column with index 1 where we had 4 nans only 2 nans have been interpolated with these values and 2 are left as nans itself. Now more common method for you know filling you know na values specially from the statistical point of view on you know data mining point of view .

We can replace these nan values with a mean values, so for that you know in the fillna method we can pass on the mean value here as a replacement for na. So, in the parenthesis you can see we can pass on df7.mean, so the mean of the data frame that is mean past here. So, if I run this

all the nans they have been replaced with the mean for that particular column.

(Refer Slide Time: 21:04)



The screenshot shows a Jupyter Notebook window titled "Session Python Built-in Capabilities". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and code execution. The notebook contains three code cells:

```
In [32]: # example:
seq7 = ['shivay', 'chanting', 'om', 'namah']
seq7.sort(key = lambda x: len(x))
seq7

Out[32]: ['om', 'namah', 'shivay', 'chanting']
```

```
In [1]: # Currying: defining a new function that calls an existing function with fewer arguments
def add_numbers(x, y):
    return x + y

# Using lambda
add_5 = lambda x: add_numbers(5, x)

# Using partial function from the built-in functools module
from functools import partial
new_add_5 = partial(add_numbers, 5)

In [2]: add_numbers(2, 4)

Out[2]: 6
```

So, column 1 a different mean, column 2 a different mean, so in this fashion those nans are going to be replaced. Now let us talk about another data management aspect which is about the removing duplicates. So, many rows are going to have you know duplicate you know they would be duplicate of some other rows. So, how do we get rid of you know just keep 1 instance of those rows and remove other duplicates.

So, let us take an example here, so we will take this data frame df8 and you can have a look at this there are 2 columns k1 and k2 here and we have same in observation in this case. So, to remove you know duplicate rows you can first have a look at the output to 03. If you look at the row number row with the index 5 and row with the index 6, so for k1 column and for the k2 column we can see 2, 4, 2, 4.

So, these rows are actually being repeated here, so one of the row is a duplicate row here. So, we can you know remove this row using you know duplicate in method here, so we can call this df8.duplicated. So, what this will do, this will you know find duplicate rows and it will return a Boolean value whether it is duplicate or not. So, if I run duplicated on df8 you can see in the row

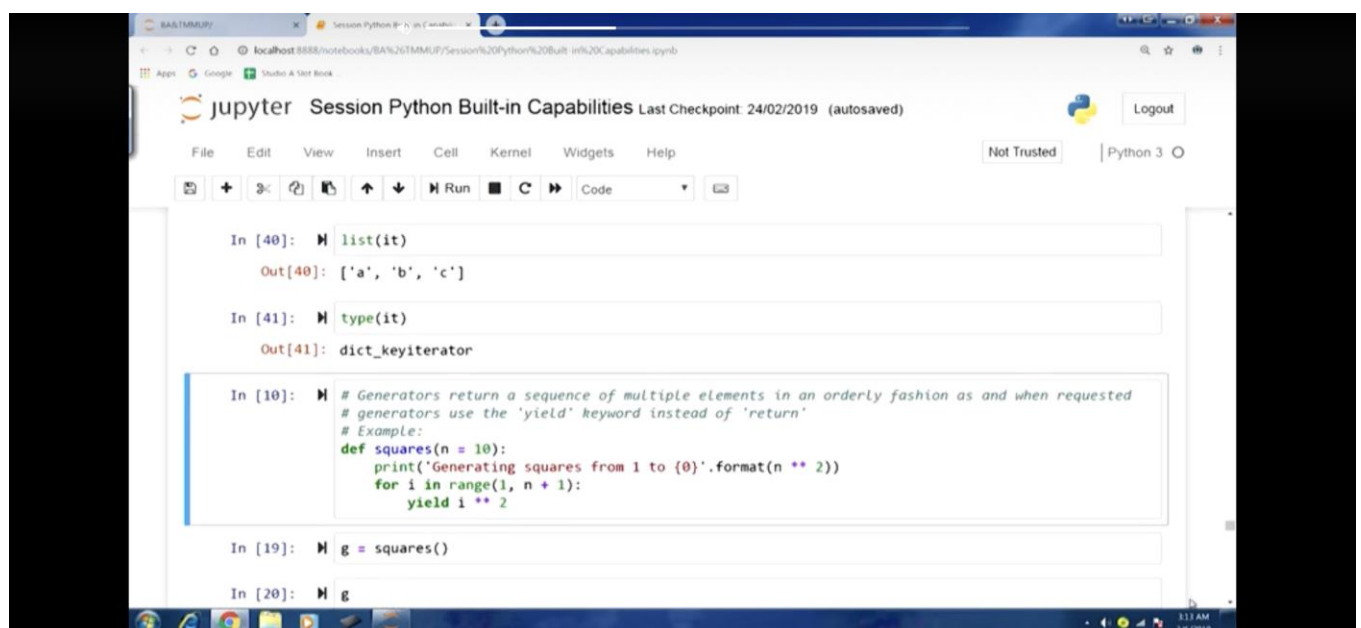
with index6 it is returning as true.

So, therefore this particular last row is actually a duplicate row, now how do I drop this duplicate row for that I can use `drop_duplicates` you know method here. So, we can call `df8.drop_duplicates` here and if I run this you can see the row within index6 is actually gone. So, that duplicate, so the last instance of this you know duplicate row is actually gone.

And the first instance of the that particular row is being you know kept in the data frame. Now sometimes we might be required to drop duplicate rows based on one particular columns instead of all the columns. So, one thing is that all the you know for certain rows all the columns have same values, so that is one thing. Sometimes you would just like to focus on 1 column and if any values repeating there.

We would like to consider that also as a duplicate instance of rows and therefore we would like to you know drop those rows. So, how do we perform this, so for that let us first add another column here and `df8` data frame. So, you can have a look at this output `df8` to 07 we have added another column `v1`. And now what we are going to do now we will drop duplicate rows based on column `k1`, the first column.

(Refer Slide Time: 22:27)



The screenshot shows a Jupyter Notebook window titled "Session Python Built-in Capabilities". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and code execution. The notebook content consists of several code cells:

- Cell 40: `list(it)` with output `['a', 'b', 'c']`.
- Cell 41: `type(it)` with output `dict_keyiterator`.
- Cell 10: A code cell containing a docstring and a generator function:

```
# Generators return a sequence of multiple elements in an orderly fashion as and when requested
# generators use the 'yield' keyword instead of 'return'
# Example:
def squares(n = 10):
    print('Generating squares from 1 to {}'.format(n ** 2))
    for i in range(1, n + 1):
        yield i ** 2
```
- Cell 19: `g = squares()`
- Cell 20: `g`

The bottom of the window shows a Windows taskbar with the system clock at 1:11 AM on 1/4/2019.

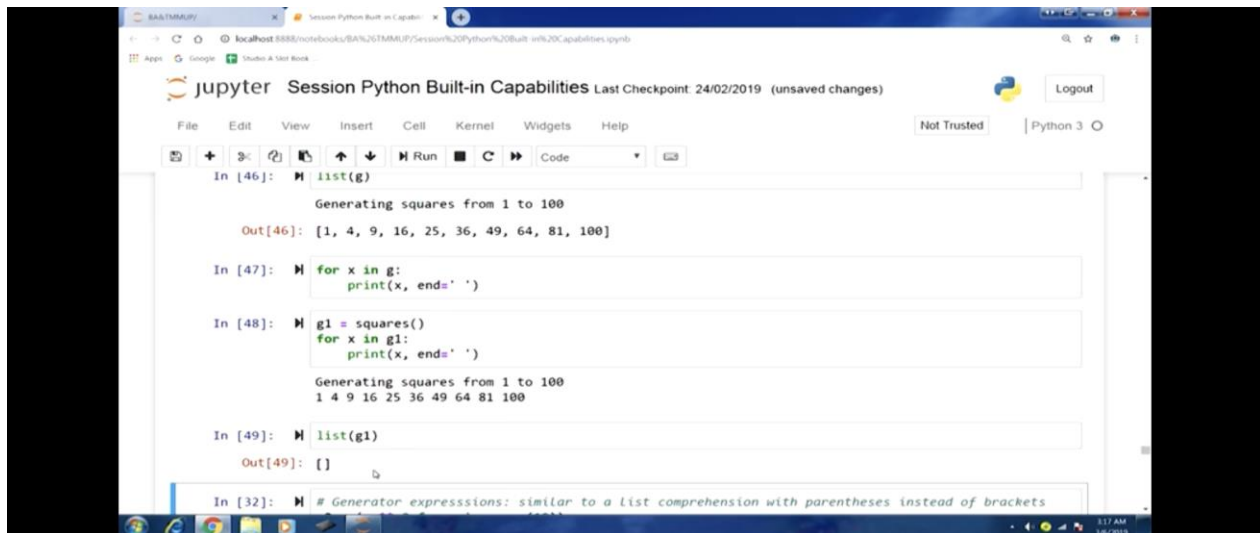
And here you can see we have 2 distinct values here 1 and 2 and they are being repeated many times. So, if I you know plan to drop you know duplicate rows based on this particular column will be just left with just because they are just 2 distinct values will be left with just you know 2 rows there. So, if I call `df8.drop_duplicates` and within parenthesis I am passing the column which is to be used for dropping the duplicate rows.

If I run this you can see in the output to 08, we have left with just 2 rows because in the k1 we had just 2 distinct values. So, therefore first instance of you look at the values here compare the output to 07 and 208 you can see the and you can have a look at the index also. So, the row index 0 and row index 1 has been have been kept and other rows have been removed here.

Because the duplication was based on 1 particular column rather than all the columns. Now if you want to change the behavior of retaining a particular instance in case of duplicate rows. So, that also we can do using keep arguments, so in this keep argument we can specify certain values use to indicate which instance we would like to keep it here.

So, for example if we would like to keep the last instance instead of the first one which is the default behavior, we can indicate the same in the keep argument. So, if you look at the code here `df8.drop_duplicates` and we are using k1, k2 here. And will so one thing is we talked about dropping based on 1 column we can also drop based on 2 columns, so here k1 and k2 we are using these 2 columns.

(Refer Slide Time: 27:07)



The screenshot shows a Jupyter Notebook window titled "Session Python Built-in Capabilities". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running code, and viewing output. The notebook contains several code cells:

- Cell [46]: `list(g)`
Output: "Generating squares from 1 to 100" followed by the list `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`.
- Cell [47]: `for x in g: print(x, end=' ')`
- Cell [48]: `g1 = squares(); for x in g1: print(x, end=' ')`
Output: "Generating squares from 1 to 100" followed by the list `1 4 9 16 25 36 49 64 81 100`.
- Cell [49]: `list(g1)`
Output: `[]`.
- Cell [32]: A comment: `# Generator expressions: similar to a List comprehension with parentheses instead of brackets`.

And then whatever duplicates we are able to find will keep the last instance instead of the first instance there. So, if I run this and you can have a look at the values here, if you look at 0, 1, 2, 3, 4. So, row with the index 5 is actually gone and row with the index 6 is you know kept there. So, 5 and 6 they were duplicate rows and you know so and duplicate based on 2 columns k1 and k2.

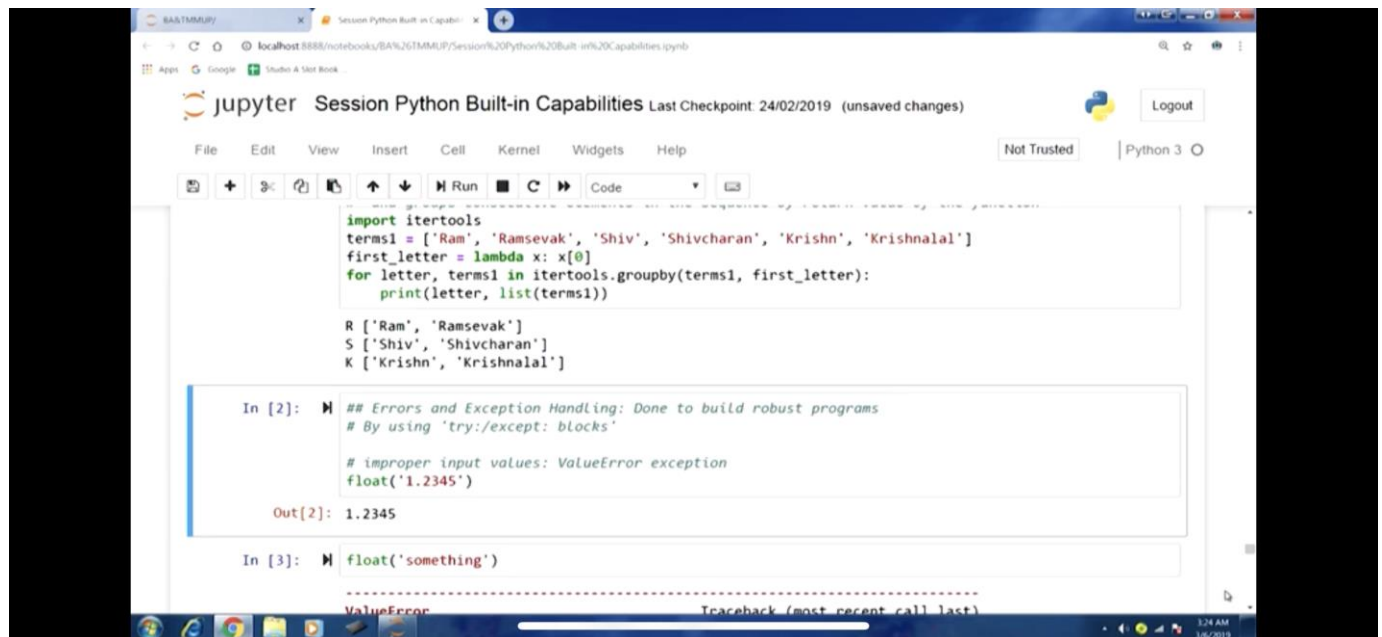
So, both had these value 2 and 4, so these were duplicate rows but we have kept the last instance instead of the first instance. So, index row with the index 5 is gone instead of index 6. So, to demonstrate to compare the results you can see here drop you know duplicates. If I drop using these 2 columns you can see the output here, so the fall behavior row index 5 is kept and the previous output as you saw that row index 6, row with the index 6 is kept over there.

Now let us move onto the another aspect that we typically you know have to deal with while working with data is to perform transformation certain data transformation based on mapping certain mappings. So, let us take few examples here, so let us focus on data transformation based on values. So, in this case first example that we are taking is about mapping of a certain distinct agro food products to agricultural crop.

So, if you look at the data frame that we are trying to you know will that will create here . In this we have food we are going to have food column and with these values atta, maida, atta, sugar,

gudh, atta, sugar, daliya, besan. So, if you look at few of the values are repeating and then we have ounces of those food item. So, let me first create this data frame you can have a look at these 2 columns food and ounces.

(Refer Slide Time: 29:10)



```
import itertools
terms1 = ['Ram', 'Ramsevak', 'Shiv', 'Shivcharan', 'Krishn', 'Krishnalal']
first_letter = lambda x: x[0]
for letter, terms1 in itertools.groupby(terms1, first_letter):
    print(letter, list(terms1))

R ['Ram', 'Ramsevak']
S ['Shiv', 'Shivcharan']
K ['Krishn', 'Krishnalal']

In [2]: ## Errors and Exception Handling: Done to build robust programs
# By using 'try:/except: blocks'

# improper input values: ValueError exception
float('1.2345')

Out[2]: 1.2345

In [3]: float('something')

ValueError                                Traceback (most recent call last)
```

Now we have another object dict object here dict 1 where we have created a mapping key value combination where each of these values that we just talked about. There mapping is done with their agricultural crop source, so each of these agro products they have been mapped with their agricultural crops here. So, atta is actually you know is processed from wheat and maida is also comes from wheat and sugar from sugarcane, gudh also sugarcane, daliya wheat and besan chana dal.

So, this kind of mapping we have here, so if I run this, so what we have now 1 data frame where we have 2 columns and values. And then we have a dict object where we have the mapping, so can we transform the data frame based on the mappings that we have. And in this transformation what we are actually going to do is we are going to add a column in the data frame which will indicate the crop.

So, for any value we would be adding a column with the crop, so instead of manually entering the actual crop source. We can use this mapping to actually automate this process here which can be really useful in a large data set context. So, here what we are doing you can have a look at this code here we are using a map method data frame map method to extract mapping from the dict object.

So, df9 crop this is so in this fashion when we indicate will be adding a new column and df9 food, so we are using the values of this column food column. And then we are applying mapping based on dict1, so if I run this will end up with 1 additional column crop and the appropriate you know crop source for the values in the food column has been added there.

So, in this fashion we are able to transform a data set based on certain mappings. Now let us talk about certain other transformation for example replacing values using the replace method. So, to demonstrate this will have the series 2 object here, so let me first define the series, so you can see you know we have certain higher negative values in the higher in the magnitude sense.

So, some of these values can be sentinel values for missing data just like na and nans. Sometimes the analyst and researcher they might also use some of these values to indicate the missing data. So, we can replace these values these sentinel values na and nas which is actually the regular sentinel marker for missing value.

So, we can use replace method series2.replace and we can replace all the instances of -999 with np.9 that is nan. So, if I run this and if you look at the series2 output here all replaces where we had -999, so those have replaced here. So, we want to do multiple we want to replace multiple values like this so there also we can use replace method. So, you can see next line of course series to.replace and we are passing a list of these values you would like to replace with nan.

So, if I run this you can see more instance of nan because the value has also been replaced. Now we can also just like other example that we have demonstrated, we can have different replacement for each value here. So, in this case in the replace method also we can have about 2 argument. First one is the list of values sentinel values that you would like to replace.

And then the second list is you know about the values which are going to be used as a replacement here as a substitute here. So, if I run this code you would see that -999 has been replaced with nan and -1000 has been replaced with -1, so different replacement for each value can also be implemented using replace method itself .

The same thing we can also achieve using a dict you know kind of mapping that we have used the similar approach in other examples also here you can see here also we are having this kind of example, so same output. Now let us talk about the renaming index label, so we have used you know various examples in this context also.

Now in the data context we will again demonstrate this, so let us take this data frame. So, in this data frame we have a full columns with the names 1, 2, 3, 4 and we have 3 rows with the cities Delhi, Mumbai, Bangalore and values in this data frame. So, what we are going to do is we are going to work with the index labels that we have, so we are going to transform some of these index label.

So, let us have a function f1 lamda function where which can actually be used to actually create the upper case version of these index names. So, let me first define this function f1 and what we can do is we can use the index attribute for this data frame df10.index and then apply this map method call this map method and apply this function f1. So, that we are able to change the index name as per the functionality that we are define in function f1.

So, if I call this you can have a look at the output 2 to 1 here you can see Delhi, Mumbai and Bangalore just 3 characters have been kept and all and the upper cases. So, in this fashion we have done slicing of those you know index name because some of these names were slightly containing more characters. So, we have just kept the first 3 characters using slicing column 3 and then use the upper cases of those characters.

So, in this fashion the index will actually be changed so let us take another example, so let me first change the index that we just talked about. So, you can see in the output 2 to 2 the index row

index names have changed. Now in the next example what will do is will demonstrate rename methods, so we will take this df10 that you can see here.

And next thing we will do we will call this rename method which can also be use to change these index names. So, in this if you look at we are calling this method df10 this data frame object. And the first argument we are that is for the index and there we are applying this str.title you know function here. So, it will actually change the row index as per the title former that is typically there where the first character is alphabetical and others are in small lower cases.

First one in the upper cases and then for the column index name we are calling this str.upper function, so it will convert all the characters into the upper cases. So, if I run this you can see in the output 2 to 4 all the index name whether row and column they have been changed in one go in this fashion. So, we can also use dict mapping with the rename method.

So, again let us take the example of tf10 this is the current names for you know index names for df10. So, you can see again we are calling rename method and for the index we can specifying dict object for dell we are mapping this as INDRAPRASTH and for columns with 3 colon with the colon name 3 we are replacing this with a teen.

So, let me run this and let us have a look at, so at the output 2 to 6 you can compare this with 2 to 5 and you can see the changes there. So, dict object dict mapping can also used to perform this kind of renaming there. Now we can also go for in phrase modification using rename method for that we have this inplace argument. So, again df10 and in the rename method you can have a look index colons the same values like previous example and we have used inplace argument. Now whatever changes are done they would be reflected in df10 itself, so you can see in the output here that all those changes have been reflected.

(Video Ends: 30:32)

So, with this we would like to stop this lecture and in the next one we will start our discussion on work in with data and will specifically will talk about winning of continuous variable which is important task in the analytics you know in general thank you.

Keywords: Dataframes, strings, concatenation, python, data mining modelling, analytics,
Prediction