

**Applied Accelerated Artificial Intelligence**  
**Prof. Bharatkumar Sharma**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Palakkad**

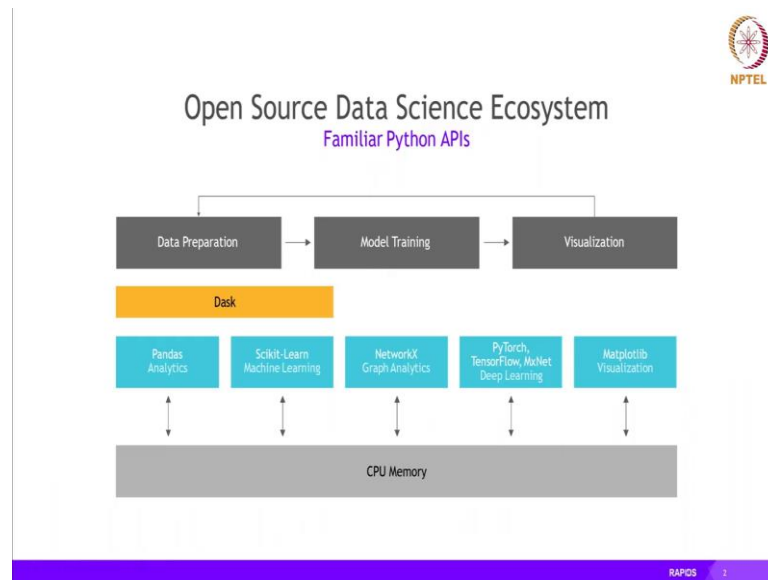
**Lecture - 46**  
**Accelerated Data Analytics Part 3**

(Refer Slide Time: 00:14)



Welcome everyone to the next lecture on Accelerated Data Analytics. In the last lecture we went through some of the motivation of using accelerated data analytics, we went through the description of the job profiles for a data engineer, data scientist. What is the cycle, what are the activities that they do, we looked at the different stacks traditionally used. So, what we are going to do today is we are going to quickly go through some of the modules of RAPIDS and we will show you a demo of how to write the code.

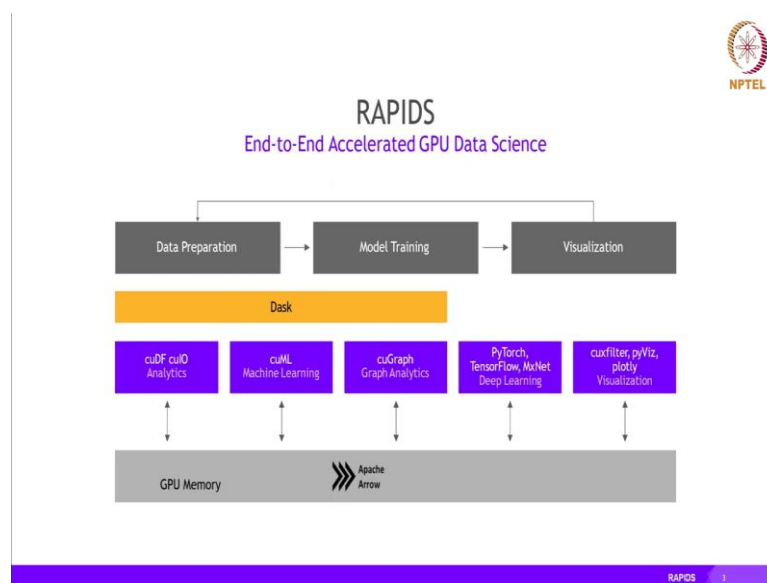
(Refer Slide Time: 01:06)



So, let us get started so maybe a quick recap of the software stack of open source data science platform. We talked about the most popular ones being used on the CPU side of it. Pandas primarily used for analytics, scikit learn or sklearn being used for machine learning algorithms.

And then we have other libraries or modules for networkX which is primarily for graph analytics deep learning. And then Matplotlib for visualization and we also talked about Dask, Dask basically helps us in making our computation go distributed. So, basically the responsibility is towards that side.

(Refer Slide Time: 01:55)



So, we also talked about the equivalent wrapped stack which is the accelerated GPU science stack. If you see we are replacing the bottom most layer with the help of use of apache arrow. So, that everything remains in memory in the GPU and the other advantages that we saw last time also removing the marshalling and unmarshalling serialization and deserialization across different modules and all.

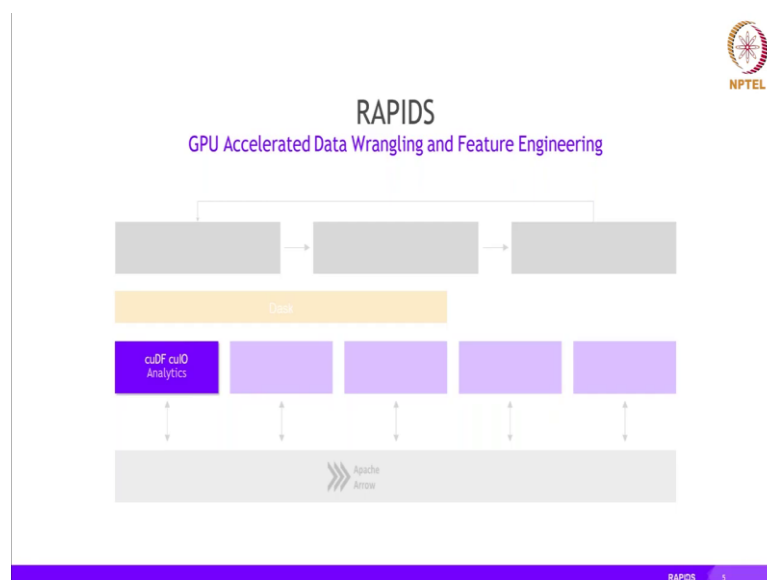
Today we are going to concentrate on cuDF and cuIO which is a replacement of Pandas equivalent it is like Pandas equivalent and we will be going through a demo of how you could utilize cuDF and cuIO. If possible if time permits we will also go through cuML show you a demo of also what machine learning algorithms exist. And how we can utilize them using cuML, which is cudaML and tomorrow we are going to go more into the task point of view.

(Refer Slide Time: 03:04)

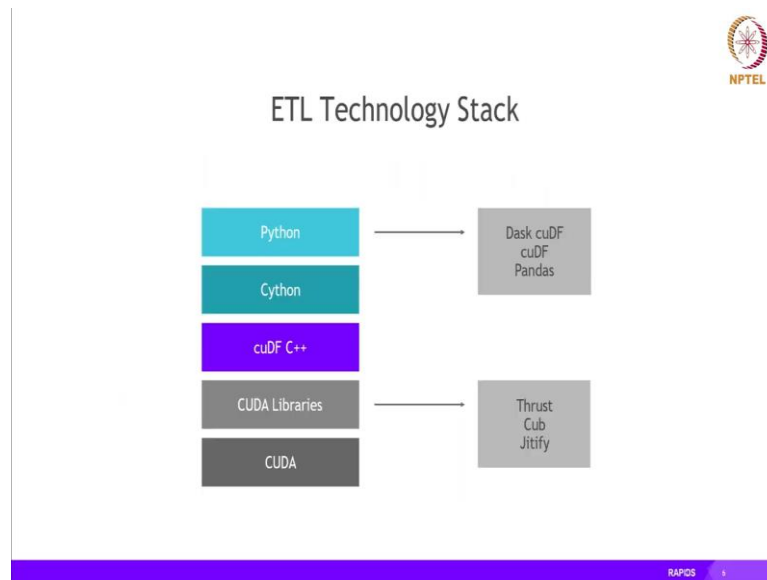


So, let us get started with cuDF if you remember last time you said how the acronym cu comes into the picture cu basically stands for CUDA which is the GPU architecture Compute Unified Device Architecture and DF is the Data Frame. So, we are talk we are basically talking about having data frames which are accelerated on the GPU using CUDA.

(Refer Slide Time: 03:28)



(Refer Slide Time: 03:31)



So, from a point of view of the ETL technology stack, which is the extract transform load part of it. This is how the overall wrappers are generally written inside RAPIDS first of all as you can see the bottom most layer is our architecture of GPU which is CUDA Compute Unified Device Architecture. Above that we have certain libraries which are part of the CUDA ecosystem like thrust is a library which is like C++ STL equivalent which is standard template library equivalent in GPUs.

We have other kinds of libraries also which exist in the CUDA ecosystem which are primarily C, C++ or C based. Over that we have the RAPIDS ecosystem basically develops has developed a layer of cuDF functionality which is written in C++. So, all those calls of the libraries of CUDA which are also in C++ are basically done via this cuDF, C++ calls.

But what a Python users basically they end up writing code in Python and for you for the binding or for the calling of C++ the internal usage goes via Cython. So, Cython is the one which is used for converting your Python poles to the C++ calls behind the scenes.

(Refer Slide Time: 05:10)

ETL - the Backbone of Data Science  
libcuDF is...

**CUDA C++ LIBRARY**

- ▶ Table (dataframe) and column types and algorithms
- ▶ CUDA kernels for sorting, join, groupby, reductions, partitioning, elementwise operations, etc.
- ▶ Optimized GPU implementations for strings, timestamps, numeric types (more coming)
- ▶ Primitives for scalable distributed ETL

```
std::unique_ptr<table>  
gather(table view const& input,  
       column_view const& gather_map, ...)  
{  
    // return a new table containing  
    // rows from input indexed by  
    // gather_map  
}
```

NVIDIA C++ RAPIDS


NPTEL

So, the first component in this pipeline in the ETL pipeline which is the backbone of the data sciences libcuDF. So, as I told you cuDF libcuDF is basically a C++ library and it is primarily meant for creating data frames which is what nothing but a structured data type. Right like in form of table which has rows and columns right behind the scene for all of the APIs, which are provided.

There are CUDA kernels or CUDA functions which are running in parallel on the GPU. Like if you call a sorting function join group by reduction whatever those functions are they are basically exposed as C++ functions.

You can see here like here we are calling the gather function. This gather function is the one which will do kind of a reduction operation on the columnar data that you have. So, there are various other kinds of function is as well and there are options to also scale them across multiple GPUs.

(Refer Slide Time: 06:26)



## ETL - the Backbone of Data Science

cuDF is...

### PYTHON LIBRARY

- ▶ A Python library for manipulating GPU DataFrames following the Pandas API
- ▶ Python interface to CUDA C++ library with additional functionality
- ▶ Creating GPU DataFrames from Numpy arrays, Pandas DataFrames, and PyArrow Tables
- ▶ JIT compilation of User-Defined Functions (UDFs) using Numba

```
In [2]: #Read in the data. Notice how it decompresses as it reads the data into memory.
pdf = cudf.read_csv('rapids-data/ames-10k.csv')

In [3]: #Taking a look at the data. We use "to_pandas()" to get the pretty printing.
pdf.head().to_pandas()
```

User ID	Product ID	Gender	Age	Occupation	City	Category	Buy In Current City	Years	Marital Status	Product Co
0	1000001	P00000042	F	0+	10	A	2		0	3
1	1000001	P00000042	F	0+	17	A	2		0	1
2	1000001	P00000042	F	0+	10	A	2		0	12
3	1000001	P00000042	F	0+	10	A	2		0	12
4	1000002	P00000042	M	25+	16	C	4+		0	8

```
In [4]: #Removing the first character of the years in city string to get rid of plus sign, and converting
to int
pdf['city_years'] = pdf['Buy_In_Current_City_Years'].str.get(1).astype(int)

In [7]: #Here we can see how we can control what the value of our domain with the replace method and turn
strings to ints
pdf['City_Category'] = pdf['City_Category'].str.replace('A', '1')
pdf['City_Category'] = pdf['City_Category'].str.replace('B', '2')
pdf['City_Category'] = pdf['City_Category'].str.replace('C', '3')
pdf['City_Category'] = pdf['City_Category'].astype(int)
```

RAPIDS 1

So, as I told you that there is a libcuDF is basically C++ libraries which primarily calls behind the scene CUDA libraries like crust. And all while cuDF is basically a Python wrapper over all of this there is a Python library for manipulating GPU data frames and it is primarily meant to look very similar to Pandas API.

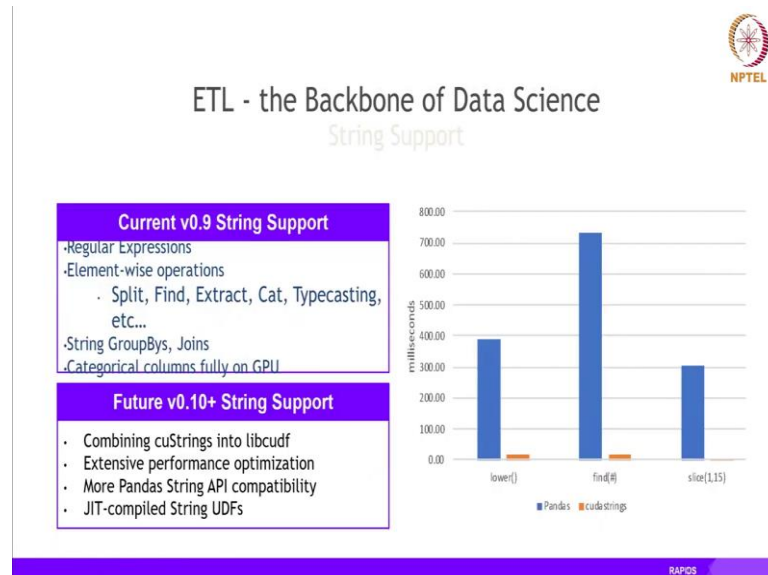
So, the idea is that if a user has already a sequential code using Pandas. It becomes very easy for them to just fit to this module and import cuDF instead of importing Pandas and rest all API should ideally look the same it also has APIs where you can import the data from Pandas or convert your GPU cuDF based data frame to Pandas equivalent.

And behind the scene it calls the CUDA C++ library which we just saw some time back. And we can also define suppose as I told you that many of these functionalities should ideally run on the GPU and there are C++ functions which are written behind the scenes already for join for string operations and all, but what if you want a function which is kind of not present in the standard list right. What if you want a user defined function which does not exist and you want to do that on your corner database.

So, cuDF basically provides JIT compilation just in time compilation of user defined function using another package called as Numba. In fact, we will be looking at how to use this to write your own accelerated functions to be run on the GPUs. So, this is one of the thing where you need to slightly understand some of the CUDA related details into

more to understand the architecture and we will be looking at it in a certain while when we start looking at the demo.

(Refer Slide Time: 08:41)




So, it has various functions including string support as well. It provides a lot of regular expressions you could do element wise operation, group by operations and there are various. So, as we are told last time that RAPIDS is basically open source project and everybody it is a large project there are so many models and packages inside it.

So many contributors from around the world and it keeps on improving by the time you might start using it in your own project. You might see more functionalities, which we are seeing is futuristic might have already been added to the RAPIDS ecosystem.



(Refer Slide Time: 09:24)



### Extraction is the Cornerstone

#### cuIO for Faster Data Loading

- Follow Pandas APIs and provide >10x speedup
- CSV Reader - v0.2, CSV Writer v0.8
- Parquet Reader - v0.7, Parquet Writer v0.10
- ORC Reader - v0.7, ORC Writer v0.10
- JSON Reader - v0.8
- Avro Reader - v0.9
- GPU Direct Storage integration in progress for bypassing PCIe bottlenecks!
- Key is GPU-accelerating both parsing and decompression wherever possible

```
1) import pandas, cudf
2) %time len(pandas.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
CPU times: user 25.9 s, sys: 3.26 s, total: 29.2 s
Wall time: 29.2 s
2) 12748986
3) %time len(cudf.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))
CPU times: user 1.59 s, sys: 372 ms, total: 1.96 s
Wall time: 2.12 s
3) 12748986
4) !du -hs data/nyc/yellow_tripdata_2015-01.csv
1.96 data/nyc/yellow_tripdata_2015-01.csv
```

Source: Apache Crawl blog: [SQL Performance: Part 1 - Input File Formats](#)

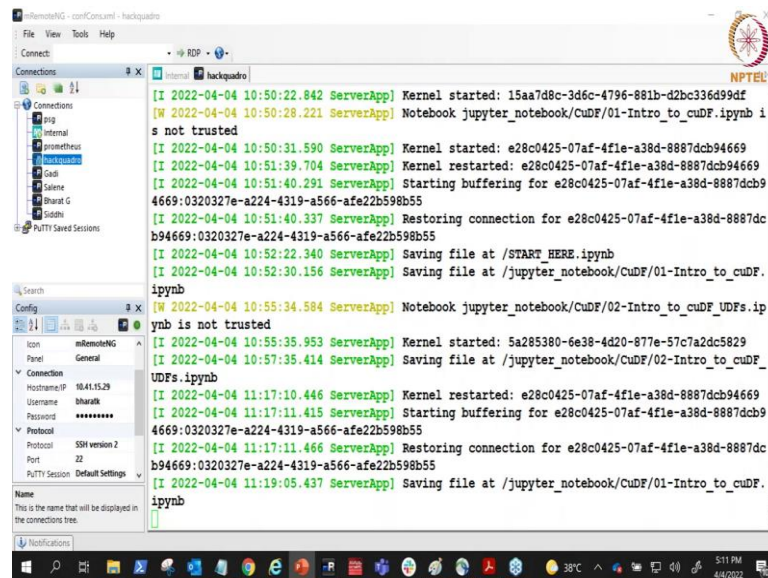
RAPIDS

There is another package which is very very key for faster data loading which is called as cuIO, cuIO is basically an API which provides you functionalities for reading files or doing a I/O operations like reading a CSV file. There are different kinds of readers like ORC reader if you are having a JSON file then read a JSON file.

And behind the scene it supports a very high speed read functionality and low latency using certain features like GPU direct storage. It basically bypasses a large amount of bottlenecks, which are there behind the scenes when you read from a file and directly store it in the GPU and that is why it is called a GPU direct storage. So, it will take care of all the optimizations whenever you want to read large files onto the GPU.

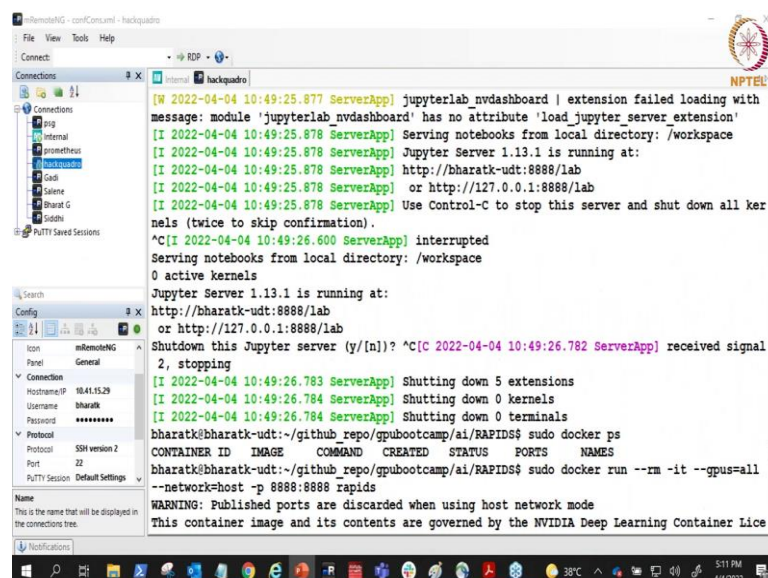
So, it does a lot of operations like parsing because like JSON you need to parse it does the parsing and decompress wherever possible. And cuIO is one of the most critical component in case you are reading from files as well. So, with that let me quickly move ahead and move towards the demo part of it.

(Refer Slide Time: 10:52)



```
Internal hackquadro
[2022-04-04 10:50:22.842 ServerApp] Kernel started: 15aa7d8c-3d6c-4796-881b-d2bc336d99df
[W 2022-04-04 10:50:28.221 ServerApp] Notebook jupyter_notebook/CuDF/01-Intro_to_cuDf.ipynb is not trusted
[I 2022-04-04 10:50:31.590 ServerApp] Kernel started: e28c0425-07af-4f1e-a38d-8887dcb94669
[I 2022-04-04 10:51:39.704 ServerApp] Kernel restarted: e28c0425-07af-4f1e-a38d-8887dcb94669
[I 2022-04-04 10:51:40.291 ServerApp] Starting buffering for e28c0425-07af-4f1e-a38d-8887dcb94669:0320327e-a224-4319-a566-afe22b598b55
[I 2022-04-04 10:51:40.337 ServerApp] Restoring connection for e28c0425-07af-4f1e-a38d-8887dcb94669:0320327e-a224-4319-a566-afe22b598b55
[I 2022-04-04 10:52:22.340 ServerApp] Saving file at /START_HERE.ipynb
[I 2022-04-04 10:52:30.156 ServerApp] Saving file at /jupyter_notebook/CuDF/01-Intro_to_cuDf.ipynb
[W 2022-04-04 10:55:34.584 ServerApp] Notebook jupyter_notebook/CuDF/02-Intro_to_cuDf.UDFs.ipynb is not trusted
[I 2022-04-04 10:55:35.953 ServerApp] Kernel started: 5a285380-6e38-4d20-877e-57c7a2dc5829
[I 2022-04-04 10:57:35.414 ServerApp] Saving file at /jupyter_notebook/CuDF/02-Intro_to_cuDf.UDFs.ipynb
[I 2022-04-04 11:17:10.446 ServerApp] Kernel restarted: e28c0425-07af-4f1e-a38d-8887dcb94669
[I 2022-04-04 11:17:11.415 ServerApp] Starting buffering for e28c0425-07af-4f1e-a38d-8887dcb94669:0320327e-a224-4319-a566-afe22b598b55
[I 2022-04-04 11:17:11.466 ServerApp] Restoring connection for e28c0425-07af-4f1e-a38d-8887dcb94669:0320327e-a224-4319-a566-afe22b598b55
[I 2022-04-04 11:19:05.437 ServerApp] Saving file at /jupyter_notebook/CuDF/01-Intro_to_cuDf.ipynb
```

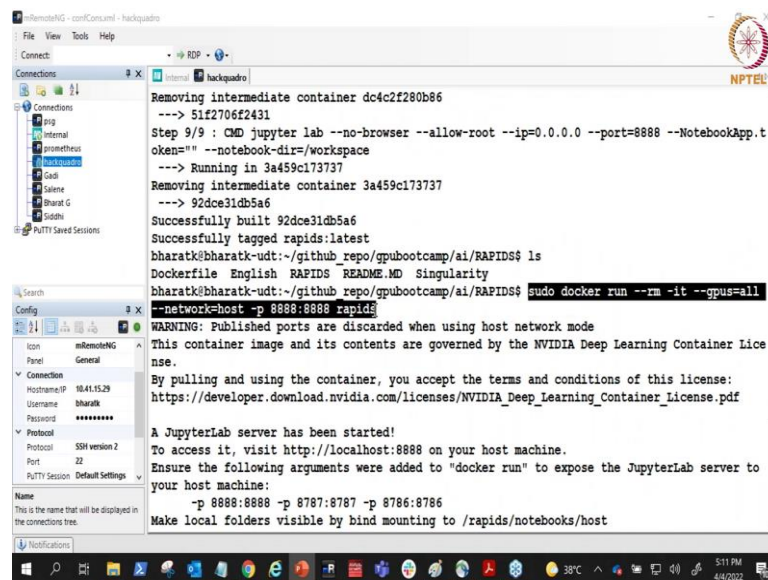
(Refer Slide Time: 10:56)



```
Internal hackquadro
[W 2022-04-04 10:49:25.877 ServerApp] jupyterlab_nvdashboard | extension failed loading with message: module 'jupyterlab_nvdashboard' has no attribute 'load_jupyter_server_extension'
[I 2022-04-04 10:49:25.878 ServerApp] Serving notebooks from local directory: /workspace
[I 2022-04-04 10:49:25.878 ServerApp] Jupyter Server 1.13.1 is running at:
[I 2022-04-04 10:49:25.878 ServerApp] http://bharatk-udt:8888/lab
[I 2022-04-04 10:49:25.878 ServerApp] or http://127.0.0.1:8888/lab
[I 2022-04-04 10:49:25.878 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
^C[I 2022-04-04 10:49:26.600 ServerApp] interrupted
Serving notebooks from local directory: /workspace
0 active kernels
Jupyter Server 1.13.1 is running at:
http://bharatk-udt:8888/lab
or http://127.0.0.1:8888/lab
Shut down this Jupyter server (y/[n])? ^C[C 2022-04-04 10:49:26.782 ServerApp] received signal 2, stopping
[I 2022-04-04 10:49:26.783 ServerApp] Shutting down 5 extensions
[I 2022-04-04 10:49:26.784 ServerApp] Shutting down 0 kernels
[I 2022-04-04 10:49:26.784 ServerApp] Shutting down 0 terminals
bharatk@bharatk-udt:~/github_repo/gpubootcamp/ai/RAPIDS$ sudo docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS          NAMES
bharatk@bharatk-udt:~/github_repo/gpubootcamp/ai/RAPIDS$ sudo docker run --rm -it --gpus=all --network=host -p 8888:8888 rapids
WARNING: Published ports are discarded when using host network mode
This container image and its contents are governed by the NVIDIA Deep Learning Container License
```

So, behind the scenes what I have done is I am basically running a particular lab and just to again show you previously we had shown this to you that I have basically created a docker container.

(Refer Slide Time: 11:10)

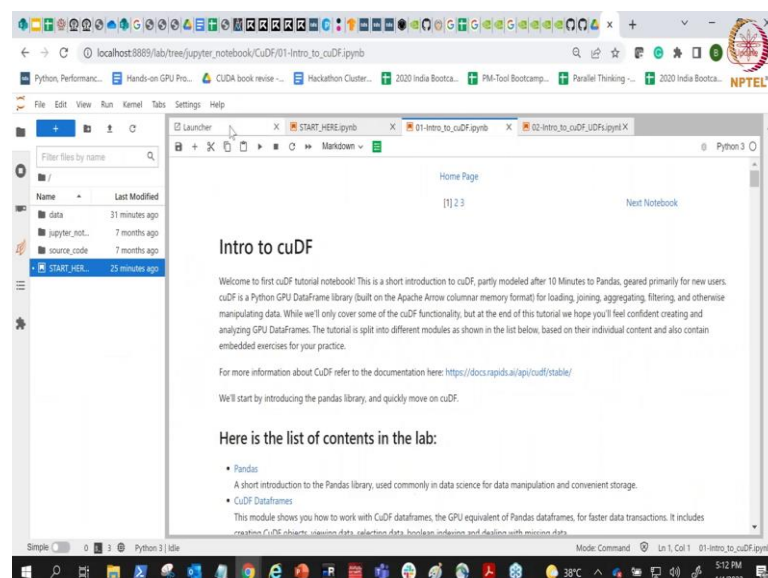


```
Removing intermediate container dc4c2f280b86
--> 51f2706f2431
Step 9/9 : CMD jupyter lab --no-browser --allow-root --ip=0.0.0.0 --port=8888 --NotebookApp.t
oken="" --notebook-dir=/workspace
--> Running in 3a459c173737
Removing intermediate container 3a459c173737
--> 92dce31db5a6
Successfully built 92dce31db5a6
Successfully tagged rapids:latest
bharatk@bharatk-udt:~/github_repo/gpubootcamp/ai/RAPIDS$ ls
Dockerfile English RAPIDS README.MD Singularity
bharatk@bharatk-udt:~/github_repo/gpubootcamp/ai/RAPIDS$ sudo docker run --rm -it --gpus=all
--network=host -p 8888:8888 rapids
WARNING: Published ports are discarded when using host network mode
This container image and its contents are governed by the NVIDIA Deep Learning Container Lice
nse.
By pulling and using the container, you accept the terms and conditions of this license:
https://developer.download.nvidia.com/licenses/NVIDIA_Deep_Learning_Container_License.pdf
A JupyterLab server has been started!
To access it, visit http://localhost:8888 on your host machine.
Ensure the following arguments were added to "docker run" to expose the JupyterLab server to
your host machine:
-p 8888:8888 -p 8787:8787 -p 8786:8786
Make local folders visible by bind mounting to /rapids/notebooks/host
```

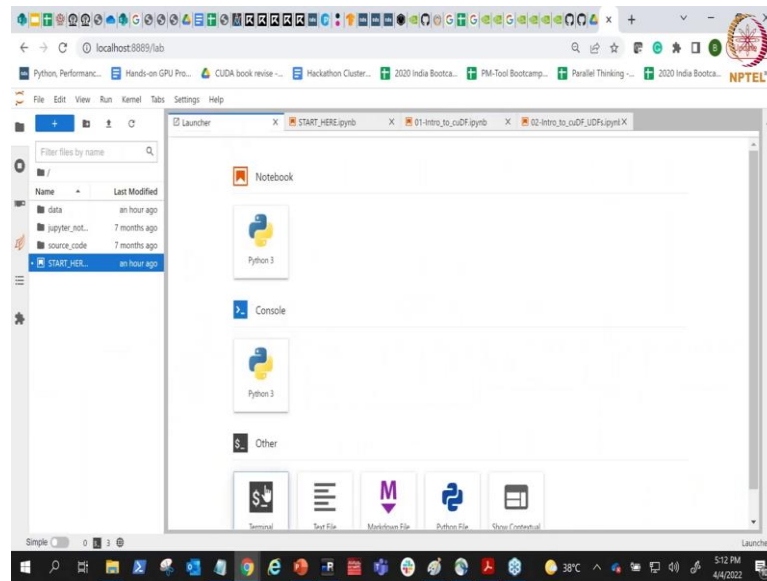
And we have provided you some of the links on the slack channel, but along with that I am going to also provide the links. Again once more at the end of the lecture on the slack channel where you can find the source code that we are showing and if you want you can also run it on the cloud infrastructures which are free versions. Or if you have a GPU access you can use the container to run it on your own machine as well.

So, you can see here I am running a docker container I am exposing it to all of the GPUs. And behind the scenes I am actually running on a particular port.

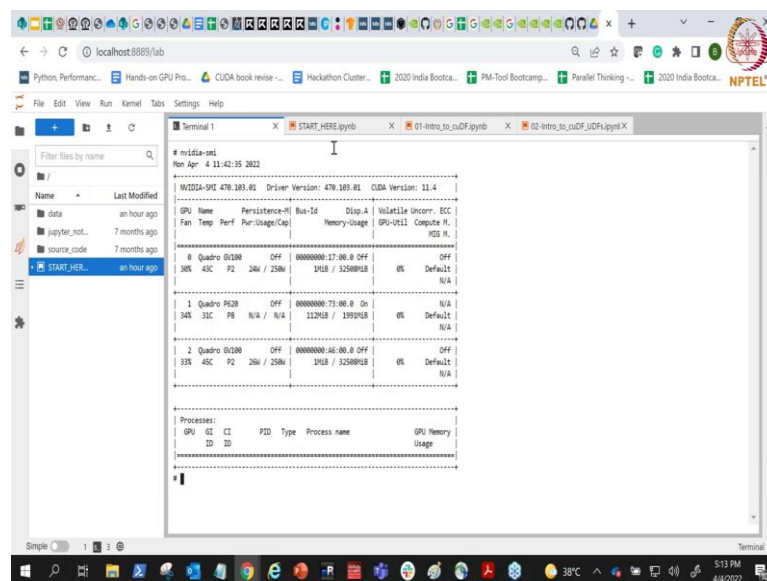
(Refer Slide Time: 11:45)



(Refer Slide Time: 11:46)



(Refer Slide Time: 11:48)

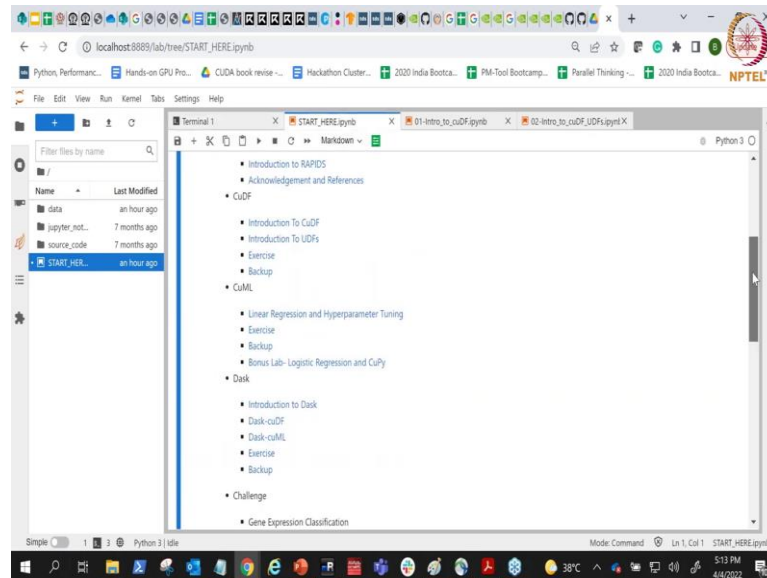


So, you can see I already launched the lab and just to show you the lab is running on the machine. And this machine basically has primarily three cards GPU cards it is a slightly different card this is Quadro GV100 can see a Quadro GV100. This is a Volta card this one as you can see has so much of memory 32 GB of memory, which is present in this Volta card it is a desktop actually not a server.

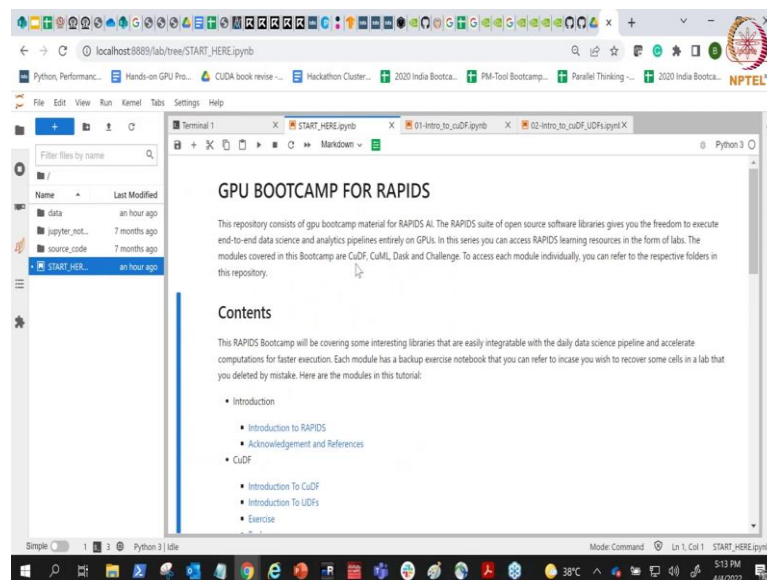
So, it is having a active cooling component with its own fan and all and it is sitting in a workstation at my desk at this point of time. So, as you can see here via nvidia-smi I can

see that I have the GPUs exposed to me inside the container. So, the lab that I am showing you here is basically a lab which has different components of RAPIDS and we are going to go through each one of them.

(Refer Slide Time: 12:41)

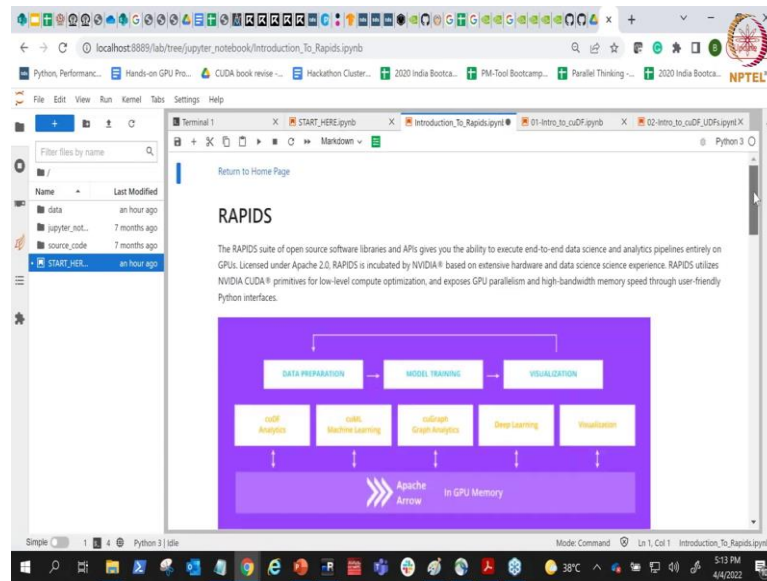


(Refer Slide Time: 12:45)

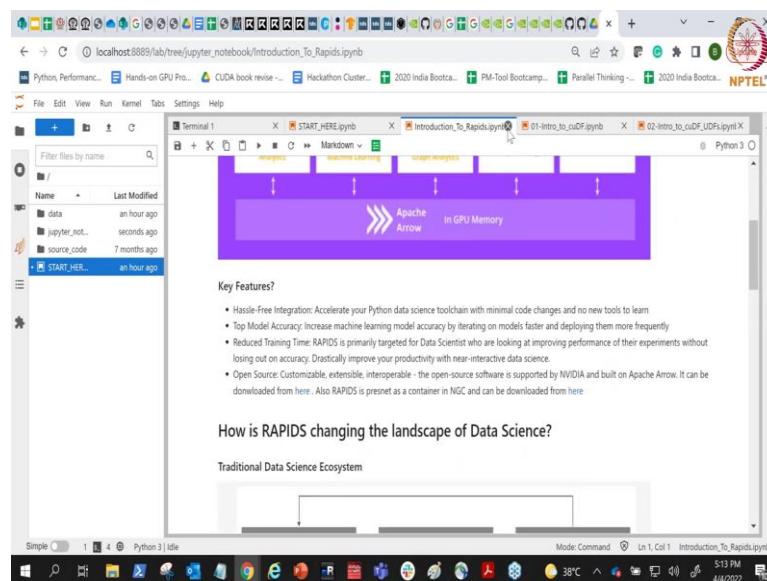




(Refer Slide Time: 12:56)

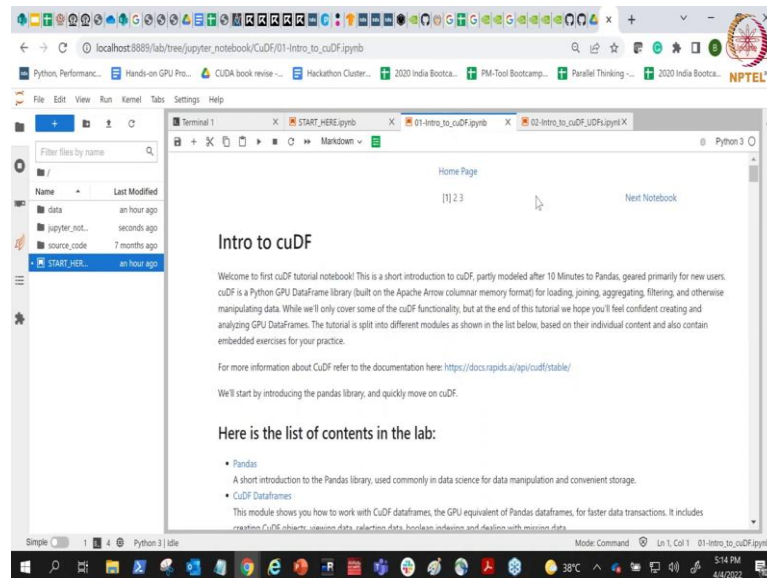


(Refer Slide Time: 13:01)



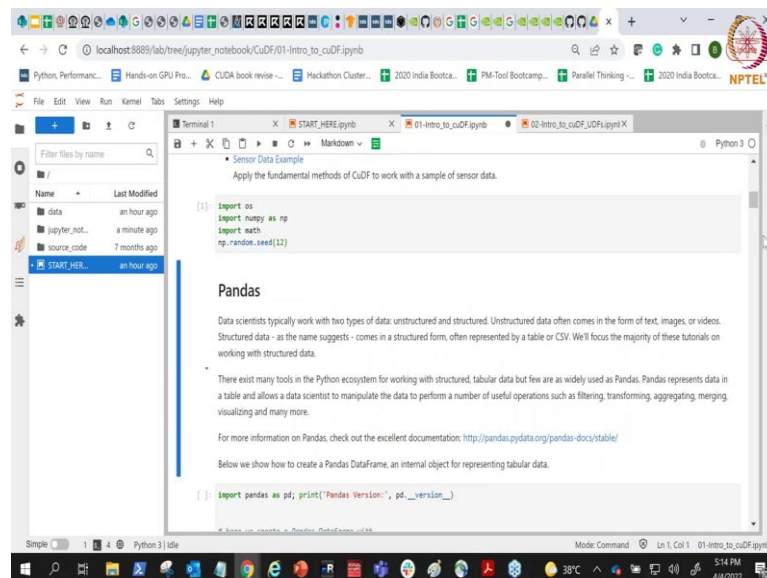
So, the lab basically whatever I showed you in terms of RAPIDS ecosystem is kind of present in the theory section and what I am going to show you now is the part of cuDF.

(Refer Slide Time: 13:08)



So, the component that we are looking at here is cuDF and as I told you basically; so, I am going to keep on running some of those things.

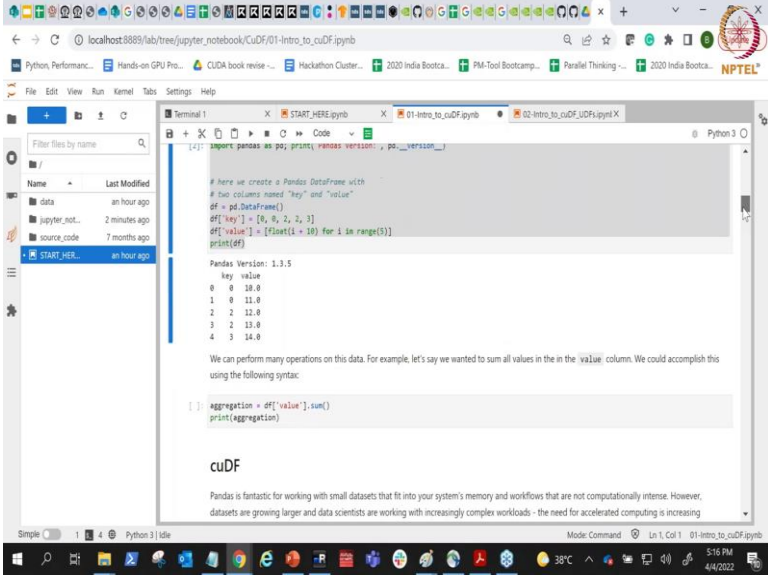
(Refer Slide Time: 13:17)



So, some of you should be already familiar with the CPU equivalent which is Pandas. Generally you will have two types of data unstructured and structured when I say unstructured I am generally referring to data like text images or videos. When I talk about structured data I am generally talking about columnar structures like in form of CSVs and all.

There are various tools which exist in the Python environment to do a lot of things. But one of the most popular one out there is Pandas which is which represents data in form of a table. And it allows you to manipulate the data perform various kinds of operations that we are going to see in this particular lab.

(Refer Slide Time: 14:06)



The screenshot shows a Jupyter Notebook interface with a file browser on the left and a code editor on the right. The code editor contains the following Python code:

```
[1]: import pandas as pd; print(pandas.__version__); pd.__version__

# Here we create a Pandas DataFrame with
# two columns named "key" and "value"
df = pd.DataFrame()
df['key'] = [0, 0, 2, 2, 3]
df['value'] = [float(i + 10) for i in range(5)]
print(df)
```

The output of the code is displayed below the code cell:

```
Pandas Version: 1.3.5
key value
0  0  10.0
1  0  11.0
2  2  12.0
3  2  13.0
4  3  14.0
```

Below the output, there is a text block explaining that many operations can be performed on this data, such as summing all values in the 'value' column. The following syntax is provided:

```
[ ] aggregation = df['value'].sum()
print(aggregation)
```

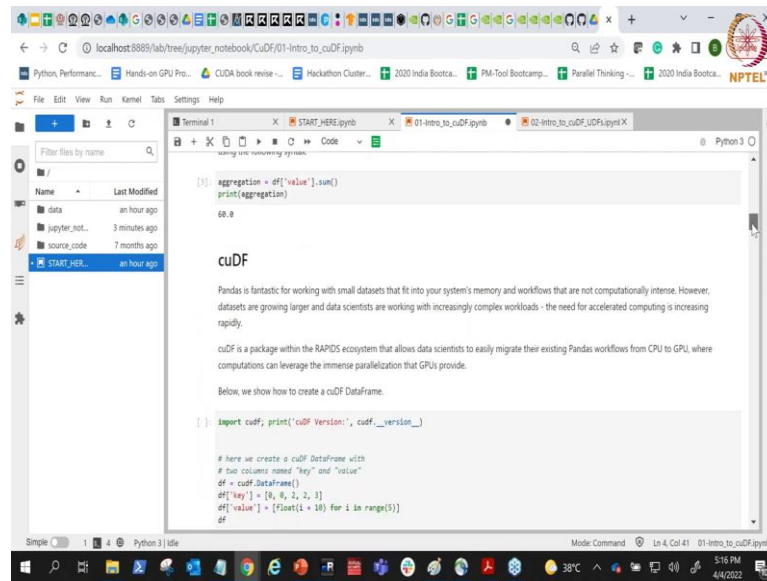
Below the syntax, there is a section titled 'cuDF' with a brief description: 'Pandas is fantastic for working with small datasets that fit into your system's memory and workflows that are not computationally intense. However, datasets are growing larger and data scientists are working with increasingly complex workloads - the need for accelerated computing is increasing.'

So, the first thing that I am doing here is I am importing Pandas and after importing the Pandas you can see here. What I am doing is I am creating a data frame in the data frame what we are doing is I am creating certain key k value pair you can see here df of key.

And df for value and then I am printing values you can see here it has created these values 0, 0, 2, 2, 3 as the key and the value is form of a range from starting from 10 which is in type of float you can see and it goes till 10, 11, 12, 13, 14 right. So, this is how very easily you can actually create data frame and then you can perform different kinds of operation on this particular data.



(Refer Slide Time: 14:58)



The screenshot shows a Jupyter Notebook running in a web browser. The notebook has several tabs, with the active one being '01-Intro\_to\_cuDf.ipynb'. The left sidebar shows a file explorer with a list of files: 'data', 'jupyter\_notebook', 'source\_code', and 'START\_HERE'. The main area of the notebook displays a code cell with the following content:

```
[1]: aggregation = df['value'].sum()
    print(aggregation)

60.0
```

Below the code cell, the output '60.0' is displayed. The notebook also contains a text block titled 'cuDF' which explains that cuDF is a package within the RAPIDS ecosystem that allows data scientists to easily migrate their existing Pandas workflows from CPU to GPU, where computations can leverage the immense parallelization that GPUs provide. Below this text, there is another code cell that imports cuDF and prints its version:

```
[ ]: import cudf; print('cuDF Version:', cudf.__version__)
```

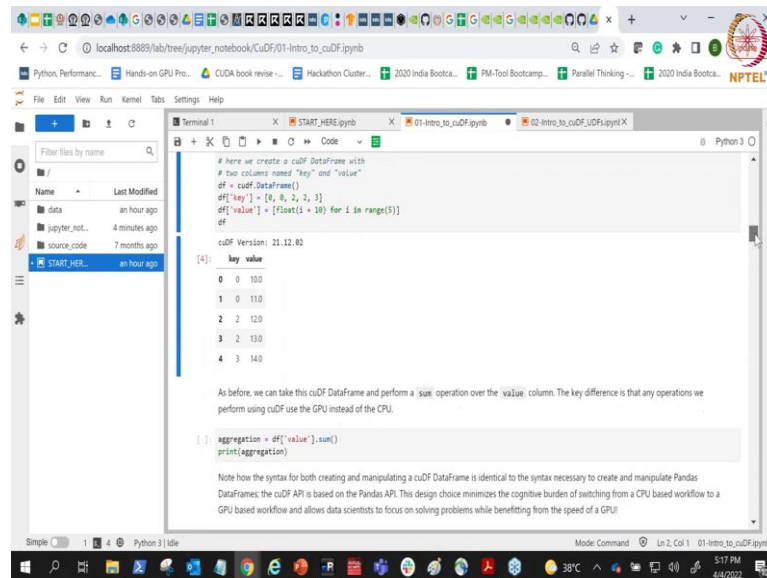
The notebook also includes a text block that explains how to create a cuDF DataFrame with two columns named 'key' and 'value'.

```
# here we create a cuDF DataFrame with
# two columns named "key" and "value"
df = cudf.DataFrame()
df['key'] = [0, 4, 2, 2, 3]
df['value'] = [float(i * 10) for i in range(5)]
df
```

So, here like one of the example is that what I am doing is I am telling that I want summation of all the columns of value and then print the finally, aggregated value right. So, you can see here if you just add up these values from 10, 11, 12, 13, 14 the value which will come is 60. So, what you are doing is you are trying to do operations on this data certain kind of operations on this data.

Now, Pandas as you can see is very flexible and it is quite good, but when we talk about working on smaller data set it works really really fine and it can give you all the benefits that you have. But if you are talking about increasingly complex workloads where you want to do more complex functionalities on it and or if you have a large data set that you need to work on this data frames on. In that case we recommend you to go towards RAPIDS which is the equivalent of the Pandas.

(Refer Slide Time: 16:03)



The screenshot shows a Jupyter Notebook interface with a file browser on the left and a code editor on the right. The code editor contains the following Python code:

```
# Here we create a cuDF DataFrame with  
# two columns named "key" and "value"  
df = cudf.DataFrame()  
df["key"] = [0, 0, 2, 2, 3]  
df["value"] = [float(i * 10) for i in range(5)]  
df
```

The output of the code is a cuDF DataFrame with 5 rows and 2 columns:

key	value
0	0
0	10
2	20
2	30
3	40

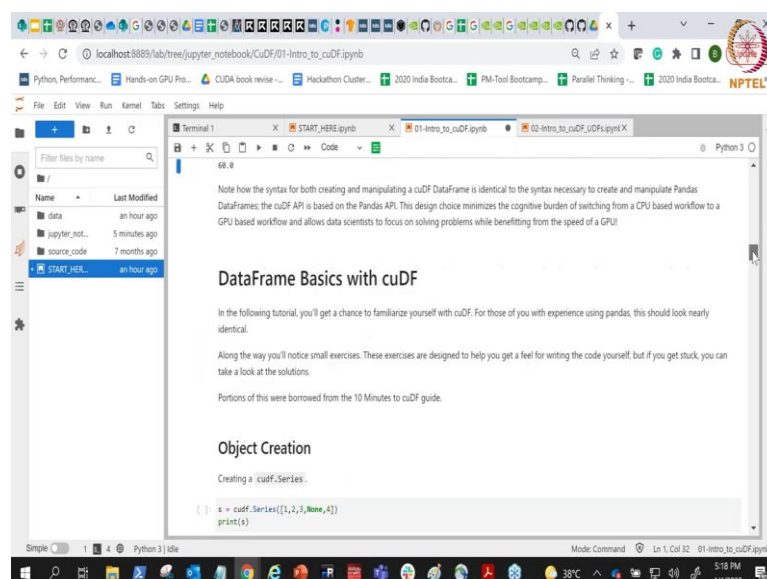
Below the output, there is a text block explaining the difference between cuDF and Pandas, followed by an aggregation operation:

```
aggregation = df["value"].sum()  
print(aggregation)
```

The output of the aggregation is 100.

So, you can see here what we are doing is we have imported cuDF. Instead of importing Pandas we have imported cuDF and rest all values remain the same. You can see here that we have cuDF data frame we have key and value there is literally no change apart from the fact that instead of importing Pandas. I have now imported cuDF and rest all things remain the same except for the fact that these values are getting created on the GPU.

(Refer Slide Time: 16:38)



The screenshot shows a Jupyter Notebook interface with a file browser on the left and a code editor on the right. The code editor contains the following text:

Note how the syntax for both creating and manipulating a cuDF DataFrame is identical to the syntax necessary to create and manipulate Pandas DataFrames; the cuDF API is based on the Pandas API. This design choice minimizes the cognitive burden of switching from a CPU based workflow to a GPU based workflow and allows data scientists to focus on solving problems while benefitting from the speed of a GPU!

### DataFrame Basics with cuDF

In the following tutorial, you'll get a chance to familiarize yourself with cuDF. For those of you with experience using pandas, this should look nearly identical.

Along the way you'll notice small exercises. These exercises are designed to help you get a feel for writing the code yourself, but if you get stuck, you can take a look at the solutions.

Portions of this were borrowed from the 10 Minutes to cuDF guide.

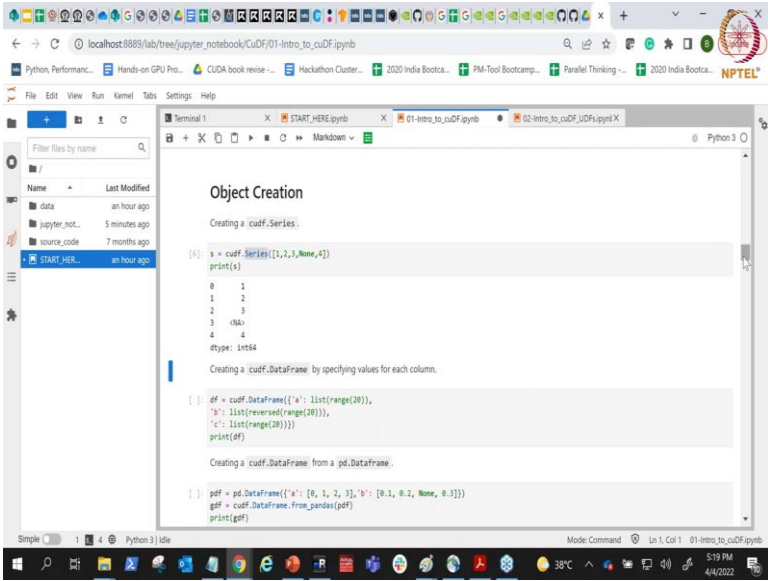
### Object Creation

Creating a cuDF Series.

```
s = cudf.Series([1,2,3,None,4])  
print(s)
```

And when I run any of the activity on this particular data frame like summation here it is actually going to run this on the GPU and not on the CPU. So, this is the only thing which means you start using the GPU functionality on by using the data frame cuDF instead of using pandas. So, now, we are going to look at different kinds of functionalities which are provided on the Pandas on the cuDF part of it. And it is based on the guide of 10 minutes to cuDF and we can see some of the activities, which will give you an idea.

(Refer Slide Time: 17:22)



```
Object Creation

Creating a cudf.Series.

In [1]: s = cudf.Series([1,2,3,None,4])
print(s)
0    1
1    2
2    3
3  <NA>
4    4
dtype: int64

Creating a cudf.DataFrame by specifying values for each column.

In [2]: df = cudf.DataFrame({'a': list(range(20)),
                             'b': list(reversed(range(20))),
                             'c': list(range(20))})
print(df)

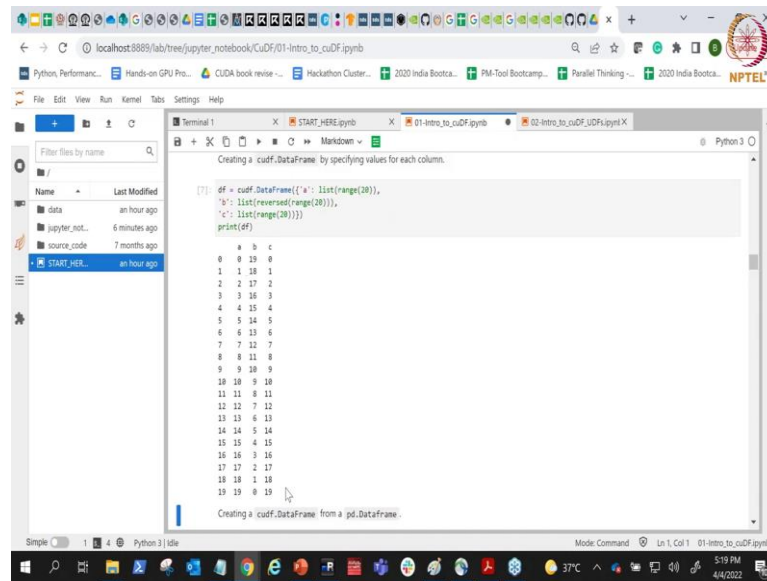
Creating a cudf.DataFrame from a pd.DataFrame.

In [3]: pdf = pd.DataFrame({'a': [0, 1, 2, 3], 'b': [0.1, 0.2, None, 0.3]})
gdf = cudf.DataFrame.from_pandas(pdf)
print(gdf)
```

In the end we will end up will showing you when can you actually get the benefit of cuDF. Ending with that in this particular notebook followed by writing your custom defined functions. So, you can see here we are actually creating object instead of creating a normal one we are creating a series and you can see here in this particular series you have certain objects which are also null.

So, you can also specify values for each column. Now you can see here you are creating a DataFrame which is of size 20 and you can see here the a column basically has the range till 20, while b is actually a reverse range and c is again the normal range here.

(Refer Slide Time: 18:12)



The screenshot shows a Jupyter Notebook interface with a file browser on the left and a code editor on the right. The code editor contains the following Python code:

```
df = cudf.DataFrame({'a': list(range(20)),  
                    'b': list(reversed(range(20))),  
                    'c': list(range(20))})  
print(df)
```

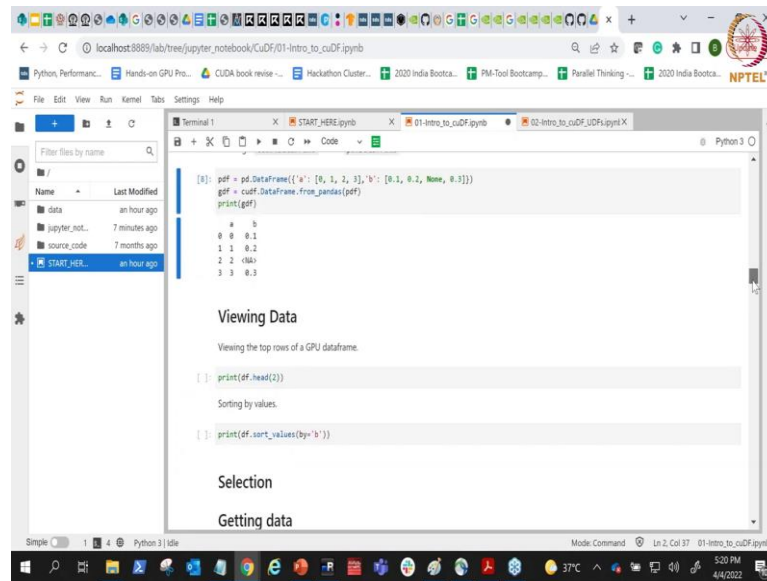
The output of the code is a table with three columns: 'a', 'b', and 'c'. The values for 'a' and 'c' are a normal range from 0 to 19, while the values for 'b' are a reverse range from 19 down to 0.

	a	b	c
0	0	19	0
1	1	18	1
2	2	17	2
3	3	16	3
4	4	15	4
5	5	14	5
6	6	13	6
7	7	12	7
8	8	11	8
9	9	10	9
10	10	9	10
11	11	8	11
12	12	7	12
13	13	6	13
14	14	5	14
15	15	4	15
16	16	3	16
17	17	2	17
18	18	1	18
19	19	0	19

So, when you print the value of df you can see here you have basically three columns a b and c, a and c being a normal range, b being starting a reverse range and you can see here the value starts from 0 and goes till 19 and for the PDF also. So, this is another version which I just told you that cuDF is kind of compatible with pandas which means you can actually create a pandas DataFrame.

You can see here I am creating a pandas DataFrame, which actually means this is going to create the data frame on the CPU and not on the GPU and then I say get cuDF dot DataFrame from pandas. So, which means I am importing the data which was created on the CPU to the GPU.

(Refer Slide Time: 19:04)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code in the editor is as follows:

```
[8]: pdf = pd.DataFrame({'a': [0, 1, 2, 3], 'b': [0.1, 0.2, None, 0.3]})
    gdf = cudf.DataFrame.from_pandas(pdf)
    print(gdf)
```

The output of the code is a table with two columns, 'a' and 'b'.

a	b
0	0.1
1	0.2
2	<NA>
3	0.3

Below the table, the text "Viewing Data" is displayed, followed by the instruction "Viewing the top rows of a GPU dataframe." The code editor also shows the following code snippets:

```
[ ]: print(df.head(2))

Sorting by values.

[ ]: print(df.sort_values(by='b'))
```

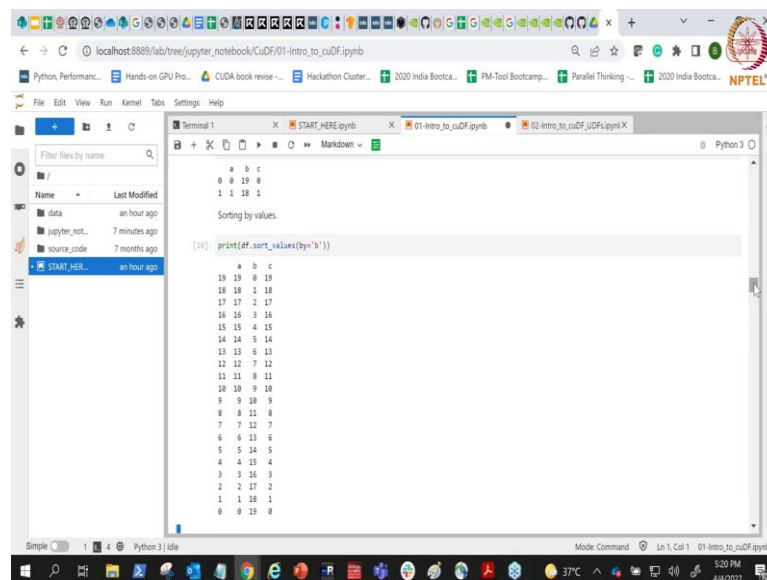
The code editor also shows the following code snippets:

```
Selection

Getting data
```

So, it in it takes the values of data frame created on the CPU using Pandas and transfers them behind the scene to the GPU to be used in a cuDF environment to be run on the GPU.

(Refer Slide Time: 19:19)



The screenshot shows the same Jupyter Notebook interface as the previous one, but with different code and output. The code in the editor is as follows:

```
[10]: print(df.sort_values(by='b'))
```

The output of the code is a table with two columns, 'a' and 'b', sorted by the values in column 'b'.

a	b
0	0.1
1	0.2
3	0.3
2	<NA>

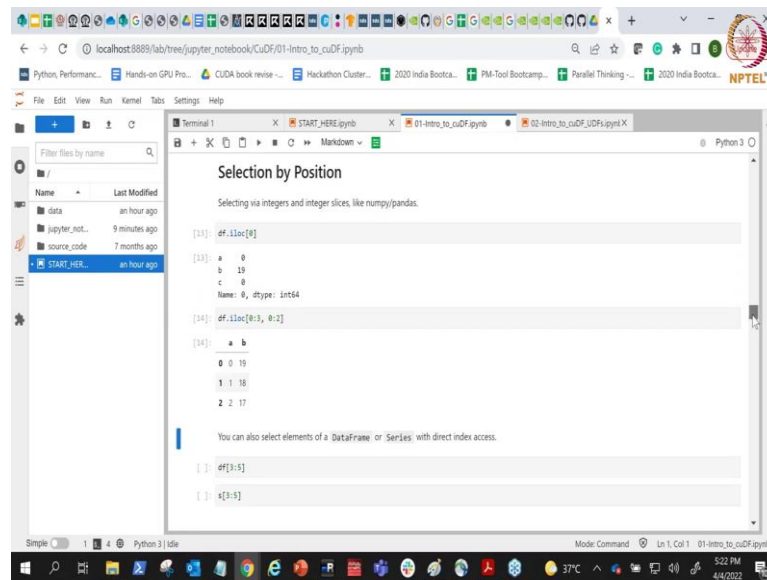
Below the table, the text "Sorting by values." is displayed. The code editor also shows the following code snippets:

```
[10]: print(df.sort_values(by='b'))
```

So, there are different kinds of visualization techniques which are also available like this will just print the head values which are the first two rows primarily. You can also sort the values based on a particular column like here we are saying I want to sort the values based on b and if you remember b was kind of inverted in nature.

So, if the value started from 19 and it went on till 1, but when I sort the values of b the values of b are getting sorted, but at the same time its equivalent row values are also getting changed accordingly. So, you are doing sort based on this particular value here right.

(Refer Slide Time: 20:03)



```
[13]: df.iloc[0]
[13]: a    0
      b   19
      c    0
      Name: 0, dtype: int64
[14]: df.iloc[0]
[14]: a    0
      b   19
      c    0
      Name: 0, dtype: int64
[15]: df.iloc[2:3, 0:2]
[15]: a    b
      0  19
      1  18
      2  17
```

You can print a single column if you want like this or. So, I just do not need to go to that part or what you can do is you can select rows particularly. Like here you are saying I want rows from index 2 to index 5 right, and the columns that I want those for is a and b and I do not want it for c particularly.

(Refer Slide Time: 20:28)

```
Selection by Label
Selecting rows from index 2 to index 5 from columns ['a' and 'b'].

[12]: print(df.loc[2:5, ['a', 'b']])

   a  b
2  2 17
3  3 16
4  4 15
5  5 14

Selection by Position
Selecting via integers and integer slices, like numpy/pandas.

[ ] df.iloc[0]
[ ] df.iloc[0:3, 0:2]

You can also select elements of a DataFrame or Series with direct index access.

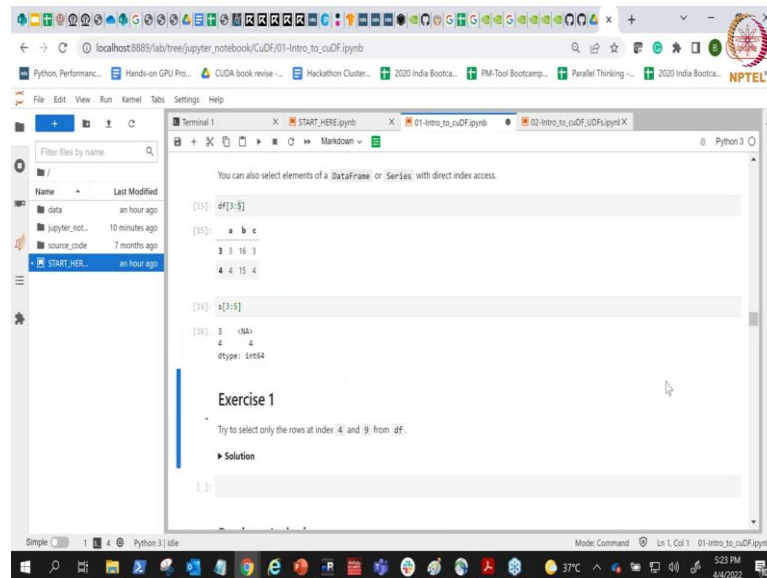
[ ] df[3:5]
```

So, you can see here it starts from 2 and it goes on till 5. And we are only getting the data which is of column a and b. So, df dot loc loc locate and then we are giving it the range the rows and columns that we want to be exported as. We can also select via a particular location using the iloc function and you can see here that I want only the location 0.

So, in that particular case it is giving me all the three values of the columns at the zeroth row. So, a having 0, c having 0 and the b column having the value 19, so it is giving me the zeroth row of all the three columns and you would have guessed it by now what does this do.

So, you can see that it is basically saying I want the rows 0, 1 and 2. And I want it for column 0 and 1. So, if you can see here when I give a range in this particular fashion it excludes the last part right, so I am saying I want 0, 1 and 2. So, it is taking from 0, 1 and 2 and this one is also 0 and 1. So, it is giving the first two columns and the first three rows primarily here.

(Refer Slide Time: 22:01)



```
[15]: df[3:5]
[15]:
   a  b  c
3  3 16  3
4  4 15  4

[16]: s[3:5]
[16]:
3  <NA>
4  4
dtype: int64
```

**Exercise 1**

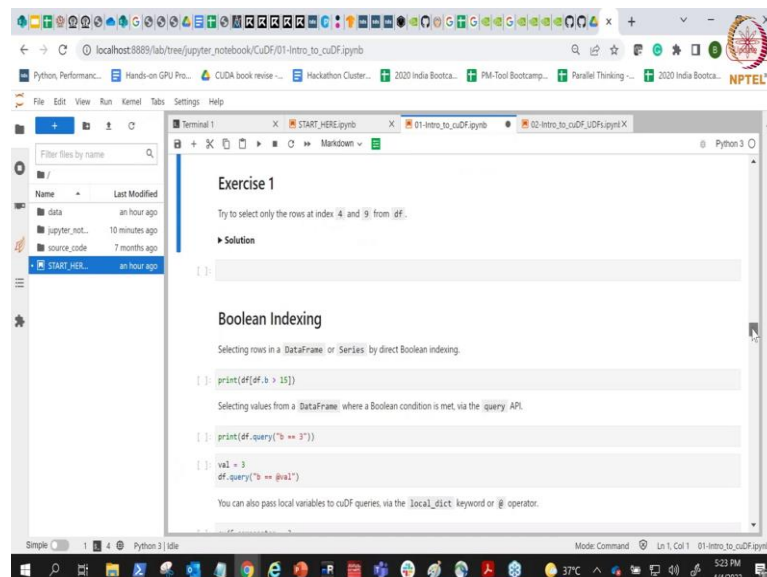
Try to select only the rows at index 4 and 5 from `df`.

**Solution**

```
[1]:
```

Similarly, I think by now you would have guessed it what does this do. So, it will give me the third row and the fourth row for all the columns which is a b and c. Because we did not explicitly state the column part of it excluding the last excluding the last one here yeah right; so this is how it is.

(Refer Slide Time: 22:25)



**Exercise 1**

Try to select only the rows at index 4 and 5 from `df`.

**Solution**

```
[1]:
```

**Boolean Indexing**

Selecting rows in a `DataFrame` or `Series` by direct Boolean indexing.

```
[1]: print(df[b > 15])
```

Selecting values from a `DataFrame` where a Boolean condition is met, via the `query` API.

```
[1]: print(df.query("b == 3"))
```

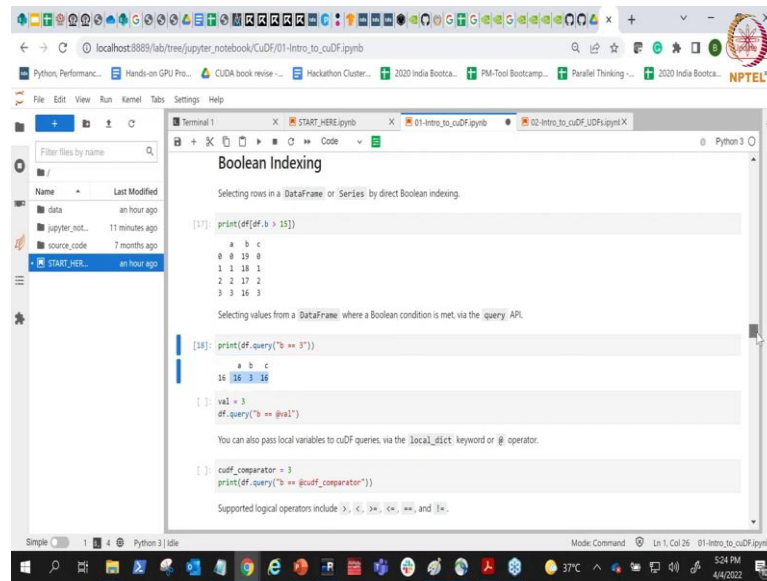
```
[1]: val = 3
df.query("b == @val")
```

You can also pass local variables to `cuDF` queries, via the `local_dict` keyword or `@` operator.

And so, let me just not give you this exercise this is kind of very simple. So, we will give you much more tougher exercise in a bit where you would be able to select something.



(Refer Slide Time: 22:37)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The notebook is titled 'Boolean Indexing'. The code in the editor demonstrates how to select rows from a DataFrame based on a condition. It includes a comment 'Selecting rows in a DataFrame or Series by direct Boolean indexing.' followed by a code cell that prints the result of a Boolean indexing operation. The output shows a DataFrame with columns 'a', 'b', and 'c', and rows where 'b' is greater than 15. Below this, another comment 'Selecting values from a DataFrame where a Boolean condition is met, via the query API.' is followed by a code cell that uses the 'query' method to select rows where 'b' is equal to 3. The output shows a single row with 'a' as 16 and 'c' as 16. A third code cell shows how to use a local variable 'cutoff\_comparator' in the query. The notebook also includes a note about supported logical operators.

```
[17]: print(df[df.b > 15])

a  b  c
0  8 15 0
1  1 18 1
2  2 17 2
3  3 16 3

Selecting values from a DataFrame where a Boolean condition is met, via the query API.

[18]: print(df.query("b == 3"))

a  b  c
16 3 16

val = 3
df.query("b == @val")

You can also pass local variables to cuDF queries, via the .local_dict keyword or @ operator.

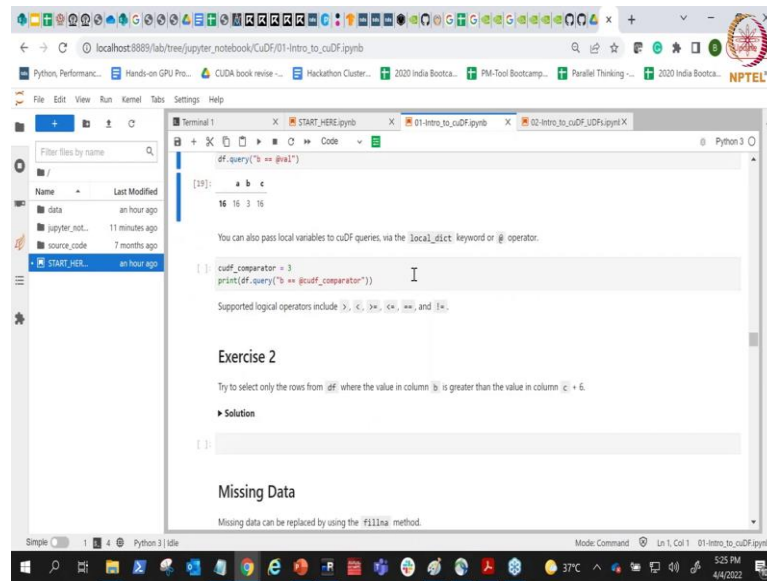
[ ]: cutoff_comparator = 3
print(df.query("b == @cutoff_comparator"))

Supported logical operators include >, <, >=, <=, ==, and !=.
```

So, selection of rows in a data frame of series by direct Boolean indexing. So, Boolean indexing is another feature where you have your data frame and what you are saying is that only if the value of b is greater than 15 then only put it then only print it right. So, only the values of only the rows where the b value is greater than 15 becomes part of the new frame right or if the b value is only 3.

In that particular case so if you see here I am using a particular API called as query. So, we can use the query to get this details. So, here I use the conditional statement in form of logic. While here I am using another API which is the query and query basically states that only if the b's value is 3 then only select that particular value.

(Refer Slide Time: 23:41)



The screenshot shows a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The code editor contains the following Python code:

```
[19]: df.query("b == @val")  
      a b c  
      16 16 3 16  
  
You can also pass local variables to cuDF queries, via the _local_dict keyword or @ operator.  
  
[ ]: cudf_comparator = 3  
     print(df.query("b == @cudf_comparator"))  
  
Supported logical operators include >, <, >=, <=, ==, and !=.

Exercise 2



Try to select only the rows from df where the value in column b is greater than the value in column c + 5.



Solution



[ ]:



Missing Data



Missing data can be replaced by using the fillna method.


```

You can also assign certain variables if you are writing Python codes you are taking your data from somewhere else you can do the same thing by using component inside Python. So, at the rate basically refers to a variable which has been previously assigned. So, this is all fine.