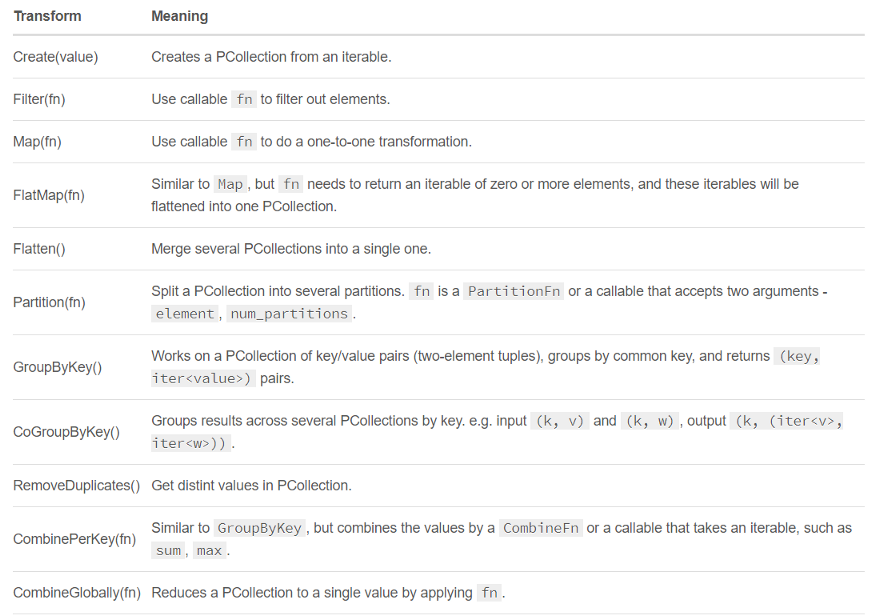
**Apache Beam: (https://github.com/apache/beam)**

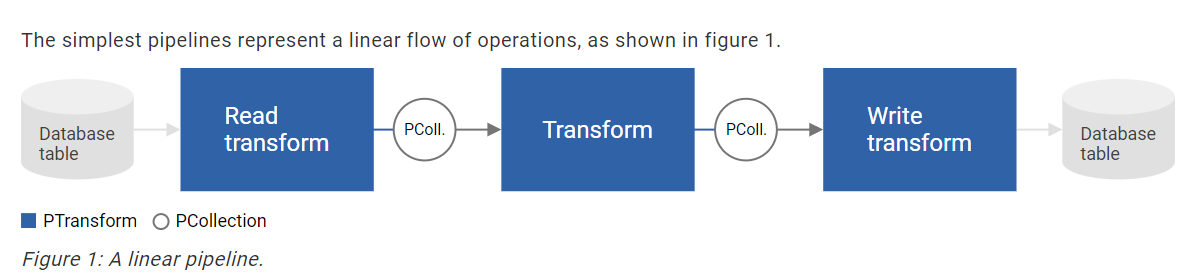
|  |  |
| --- | --- |
| Parallel computing | Distributed computing |
| * Parallel computing is the simultaneous use of more than one processor to solve a problem. * Parallel computing tasks access the same memory space. | * Distributed computing is the simultaneous use of more than one computer to solve a problem. * Distributed computing tasks is disk-based, instead of memory-based. * Some distributed computing tasks run on one computer, and some on others |



**ParDo** is a primary beam transform for generic parallel processing which is not in the above image. The ParDo processing paradigm is similar to the “Map” phase of a Map/Shuffle/Reduce-style algorithm: a ParDo transform considers each element in the input PCollection, performs some processing on that element, and emits zero, or multiple elements to an output PCollection.

**Pipe ‘|’**is the operator to apply transforms, and each transform can be optionally supplied with a unique label. Transforms can be chained, and we can compose arbitrary shapes of transforms, and at runtime, they’ll be represented as DAG.

* <http://beam.apache.org/documentation/>
* <https://beam.apache.org/documentation/programming-guide/> usethis
* <https://beam.apache.org/documentation/sdks/python/>
* Apache beam, the latest open source project of apache is a **unified** programming model for expressing efficient and **portable** big data processing pipelines.
  + **Unified** API to process both batch and streaming data.
  + **B**atch + Str**eam** -> Beam
  + **Portable,** beam pipeline once created in any language can be able to run on any execution frameworks like spark, flink, apex, cloud dataflow etc.,
  + Cloud Dataflow is fully managed service for creating and executing optimized parallel data processing pipelines.
  + Beam is a programming model whereas flink and spark are execution engines.
* **Flow of beam Programming model.**
* **Basic Terminologies in Beam.**
  + **Pipeline:** A pipeline encapsulates entire data processing task, from start to finish. Includes reading input data, transforming that data and writing output data.



* + **PCollection:** A PCollection is equivalent to RDD of spark. It represents a distributed data set that our beam pipeline operates on.
    - **Immutability:** Pcollections are immutable in nature. Applying a transformations on a pcollection results in creation of new pcollection.
    - **Element type:** The elements in pcollection may be of any type, but all must be of same type.
    - **Operation type:** Pcollection does not support grained operations. We cannot apply transformations on specific elements in pcollection.
    - **Timestamps:** Each element in pcollection has an associated timestamp with it.
    - **Unbounded pcollections:** Source assign the timestamps.
    - **Bounded pcollections:** Every element is set to same timestamp.
    - **Ptransform:** Ptransform represent a data processing operation, or a step in our pipeline. Ex., ParDo, filter, flatten, combine etc.
  + **Spark: Bounded (Dataframes**) Unbounded (Dstreams)
  + **Flink: Bounded (Dataset) and Unbounded (Datastream)**

Install the Apache Beam SDK in Cloud Shell

Dataflow runs pipelines that are defined in Apache Beam, taking input data, transforming it, then storing the output. To use Dataflow to transform data, first you need to install the Apache Beam SDK:

1. Open Cloud Shell by clicking **Activate Cloud Shell**. Show me
2. Create a virtual environment to isolate the dependencies of the current project from other projects. Enter the following commands to create a virtual environment, then activate it:
3. python3 -m virtualenv env

source env/bin/activate

1. Download the Apache Beam SDK:

pip3 install apache-beam[gcp]

# Set up a Cloud Storage bucket for output data

1. In the Cloud Console navigation menu, click **Cloud Storage > Browser**.

You can see where it is by clicking the following button:  Cloud Storage chevron\_right Browser

1. Click **Create bucket**.
2. Give your new bucket a globally unique name. Record this to use later.
3. Expand the **Choose how to control access to objects** section.
4. Select **Fine-grained**.
5. Click **Create**.

To learn how to run a Beam pipeline with Dataflow, click **Next**.

# Run a Beam pipeline in Dataflow as a job

The Apache Beam SDK comes with a ready-made pipeline, WordCount, that counts all the unique words in an input text file — in this case, Shakespeare's King Lear.

To stage the WordCount pipeline on Dataflow as a job, run the following command, replacing <STORAGE\_BUCKET> with the name of your storage bucket:

python3 -m \

apache\_beam.examples.wordcount \

--region us-central1 --input \

gs://dataflow-samples/shakespeare/kinglear.txt \

--output \

gs://**<STORAGE\_BUCKET>**/results/output \

--runner DataflowRunner \

--project \

bigquery-demo1-326311 \

--temp\_location \

gs://**<STORAGE\_BUCKET>**/temp/

Dataflow creates Compute Engine instances as required to run the pipeline, and removes them after the job is complete. Large jobs might require multiple Compute Engine instances to process the data in parallel.

After the processing completes, you can exit the Python virtual environment with the deactivate command:

deactivate

To learn about monitoring your job, click **Next**.

# Monitor your job

To check the progress of your pipeline, complete the following steps:

1. In the Cloud Console navigation menu, click **Dataflow > Jobs**.

You can see where it is by clicking the following button:

 Dataflow chevron\_right Jobs

1. Click **Jobs**.
2. Click your job name to view its details.
3. If the **Logs** panel isn't already open, click **Show**.
4. Click a step to view its metrics in the **Step info** and **Logs** panels.

As your job finishes, you'll see the job status change, and the Compute Engine instances used by the job will stop automatically.

To learn how to view the output of your job, click **Next**.

# View the output of your job

After your job has completed, you can explore the output files in Cloud Storage. To do so:

1. In the Cloud Console navigation menu, click **Cloud Storage > Browser**.

You can see where it is by clicking the following button:  Cloud Storage chevron\_right Browser

1. Click the name of the storage bucket you created earlier.
2. There are two folders created by the job:
   * The **results** folder contains the count of all the unique words in the input text. Click the file that starts with output, then click **Download** to read the results.
   * The **temp** folder contains binaries that were created for the job execution.

## **🎉 Success!**

You've run a job in Dataflow using an Apache Beam pipeline!

To avoid incurring charges on your account and to learn about next steps, click **Next**.

PREVIOUSNEXT

# Next steps

Clean up to avoid billing charges, or keep the resources you created and do more with Dataflow.

## **Clean up**

Follow these steps to delete the resources you just created:

1. In the Cloud Console navigation menu, click **Cloud Storage > Browser**.

You can see where it is by clicking the following button:  Cloud Storage chevron\_right Browser

1. Select the checkbox next to the name of the storage bucket you created.
2. Click **Delete**, then confirm the deletion.
3. If you haven't left your Python virtual environment, do so:

deactivate

1. Remove the Python dependencies and virtual environment used in this walkthrough:

rm -rf env

# Apache Beam basics

The following concepts are used throughout the notebooks:

* **Element**: minimal unit of data.
* **PCollection**: represents a distribute data set; it can be *bounded* or *unbounded*. Made of element(s).
  + *Bounded* PCollection is data that has a fixed size. For example, text files, BigQuery tables, Avro files, and so on.
  + *Unbounded* PCollections are potentially of infinite size, coming from a data stream. Examples of this are Pub/Sub topic/subscription and Kafka.

These InteractiveRunner is used in most pipelines, so that you can see their graphs and output. Apache Beam can use other runners such as DirectRunner, DataflowRunner, or FlinkRunner.

Before running into code, examine the basic structure for creating a pipeline:

* At the beginning, to define your pipeline, use p = beam.Pipeline().
* The pipe | separates steps within the pipeline. Every time you want to add a new step, you need a new pipe.
* At the right of the pipe, add the step you want to execute, | <STEP>. You can optionally name the step using >> between the step and the pipe | "NAME" >> <STEP>. Two steps cannot have the same name.
* At the left of the pipe, there has to be a reference to a pipeline p | <STEP>, p | <STEP1> | <STEP2>... or squares | <STEP> (where squares is a pipeline variable ).

Five steps:

1. Creating and giving pipeline a name
2. Initial Pcollection by reading data from source
3. Ptransforms according to the requirement (Map, Filter, CombinePerKey, apply or pipe)
4. Writing Pcollection to source

* ReadFromText
* ReadFromAvro
* ReadFromParquet
* ReadFromTFRecord
* WriteToText
* WriteToAvro
* WriteToParquet
* WriteToTFRecords
* WriteToPubSub

Read Transforms:

1. Apache kafka
2. Amazon kinesis
3. JMS
4. MQTT
5. Google Cloud pubsub

## Basic Operations

**Create** is used to create elements.

**Map** does an operation at the element level. Applies a simple one-to-one mapping function over each element in the collection.

**Composite Transforms**

* Composite transform is a type of transform which internally have series of inbuilt transforms implemented in it. Can have nested structure and performs multiple other transforms susch as ParDo, Combine,GroupByKey etc

**Side Inputs**

* Sideinputs data is determined at the runtime.
* Which of the following pair of  PTransforms can emit multiple outputs per input?
  + FlatMap & parDo

|  |  |
| --- | --- |
| Apache Beam | Apache Spark |
| * *A unified programming model*. It implements batch and streaming data processing jobs that run on any execution engine(multiple execution engines). It executes pipelines on multiple execution environments. * Apache Beam can be classified as a tool in the "Workflow Manager" category. * Due to the unified model, processing is carried out in the same way for both batch and stream data. * In GCP, Dataflow due to its serverless nature we didn’t need to set up a cluster each time we wanted to process data. The next big advantage of Dataflow is the Shuffle service, which addresses the shuffle issue on Spark executors as it moves heavy operation out of the worker virtual machine to the service backend. Besides, there is autoscaling out-of-the-box, Streaming engine for streaming pipeline support. Generally, Dataflow is supposed to be a self-managed platform, so less effort is required to configure it compared to Dataproc. * Logs from workers and metrics are displayed on the same UI and are available even after the job is finished. | * Fast and general engine for large-scale data processing. Spark is a fast and general processing engine compatible with Hadoop data. It can run in Hadoop clusters through YARN or Spark's standalone mode, and it can process data in HDFS, HBase, Cassandra, Hive, and any Hadoop InputFormat. It is designed to perform both batch processing (similar to MapReduce) and new workloads like streaming, interactive queries, and machine learning. * Apache Spark is grouped under "Big Data Tools". * Two different APIs for batch/stream data processing. You need to split data yourself by grouping by the time and it is not truly real-time processing, as basically Spark divides the data stream into micro batches of X seconds called Dstreams, which is a sequence of RDDs under the hood. * In GCP, Dataproc. Dataproc has autoscaling feature, but it requires more actions: creating autoscaling policy in GCP and integrating it into the job. Moreover, to achieve good performance we needed to play a lot with Spark configuration like tuning memory on workers, choosing appropriate number of shuffle partitions, executor instances and so on. * Familiar dashboard with Yarn metrics at our disposal - Spark UI is the same that we had in the Hadoop cluster on-premise. The inconvenience was that we had to switch between different tabs in the browser. Another issue was with Spark logs UI view, as if you open it by clicking on the particular running job, you can see only the part of them. The rest can be found in GCP Logger where you need to know how to build queries to fetch them. Also Dataproc does not keep the history of metrics once the cluster is shut down. You need to spin up a separate Spark history server to collect them and then configure its visualisation in GCP Monitoring. |

<https://stackshare.io/stackups/apache-beam-vs-spark>

<https://blog.allegro.tech/2021/06/1-task-2-solutions-spark-or-beam.html>

<https://www.quora.com/What-are-the-differences-between-Apache-Spark-Storm-Heron-Samza-Flink-Beam-Apex>

Cloud Guru:

**Cloud Dataflow: (Page 300)**

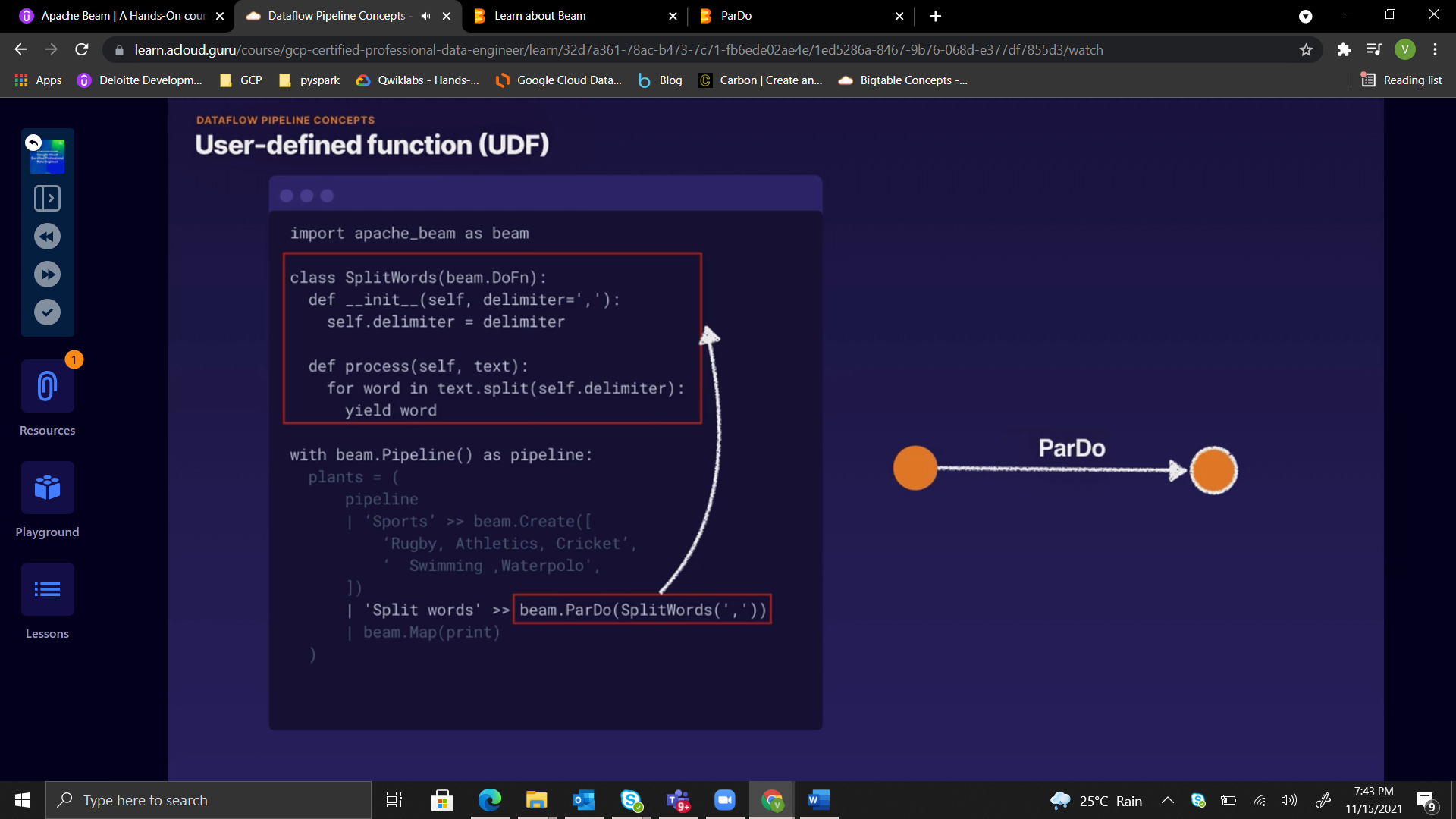
* [**https://cloud.google.com/dataflow/docs/concepts/beam-programming-model**](https://cloud.google.com/dataflow/docs/concepts/beam-programming-model)
* [**https://beam.apache.org/get-started/beam-overview/**](https://beam.apache.org/get-started/beam-overview/)

1. **Fully managed, serverless tools**
2. **Uses open source Apache beam SDK**
3. **Supports expressive SQL, JAVA and Python APIs**
4. **Real time and batch processing**
5. **Stack driver integration**

****

* **Designing a pipeline**
* **Creating a pipeline and executing it**
* **Debugging (testing) the pipeline.**
* **Considerations:**
  + **Location of data**
  + **Input data structure and format**
  + **Transformation objectives**
  + **Output data structure and location**
* **Directed Acyclic Graph(DAG)**

**Dataflow pipeline concepts:**

* **ParDo transforms (Parallel Do transforms)**
* **Aggregation transforms**
* **PCollections**
  + **Characters Of Pcollections:**
    - **Datatypes**
    - **Access**
    - **Immutability**
    - **Boundedness**
    - **timestamp**
* **Core beam transforms**
  + **ParDo(**[**http://beam.apache.org/documentation/transforms/python/elementwise/pardo/**](http://beam.apache.org/documentation/transforms/python/elementwise/pardo/) **)**
    - ParDo or ParallelDo is the generic parallel processing transform.
    - For each input element, a ParDo Transform produces either one output element, 0 output elements, or in some cases, a ParDo Transform can produce multiple output elements from a single input element.
    - A ParDo Transform will act on that element, and produce an element in the output PCollection.
    - 
  + **GroupByKey :** The GroupByKey is used to collect all the values associated with a unique key.
  + **CoGroupByKey :** CoGroupByKey is used when combining multiple PCollections, it performs a relational join of or more key value PCollections, where they have the same key type.
  + **Combine:** Combine is unsurprisingly the Beam Transform for combining elements. A Combine Transform requires you to provide a function that defines the logic for combining elements. The combining function needs to be associative and commutative.
  + **Flatten**
  + **Parition**
  + **Aggregation(**[**https://beam.apache.org/documentation/transforms/python/overview/**](https://beam.apache.org/documentation/transforms/python/overview/) **)**

[**https://cloud.google.com/appengine/docs/standard/python/googlecloudstorageclient/read-write-to-cloud-storage**](https://cloud.google.com/appengine/docs/standard/python/googlecloudstorageclient/read-write-to-cloud-storage)

**Pipeline creation:**

* Create pipeline object
* Create a Pcollection using read or create transform
* Apply multiple transfroms as required
* Write out final Pcollection
* Run the pipeline \

**Windowing:**

* Imagine an unbounded pcollection(stream of events), window functions is used.
* Window function allows you to subdivide the pcollections according to their timestamps, the point of windowing is to enable grouping or aggregating over unbounded collections. This is done by grouping elements in finite windows.
* Types:
  + Fixed(non overlapping) windows, it is for constant duration.
  + Sliding (time intervals) can overlap, running averages.
  + Session windows(interruption in flow of events)
  + Single global

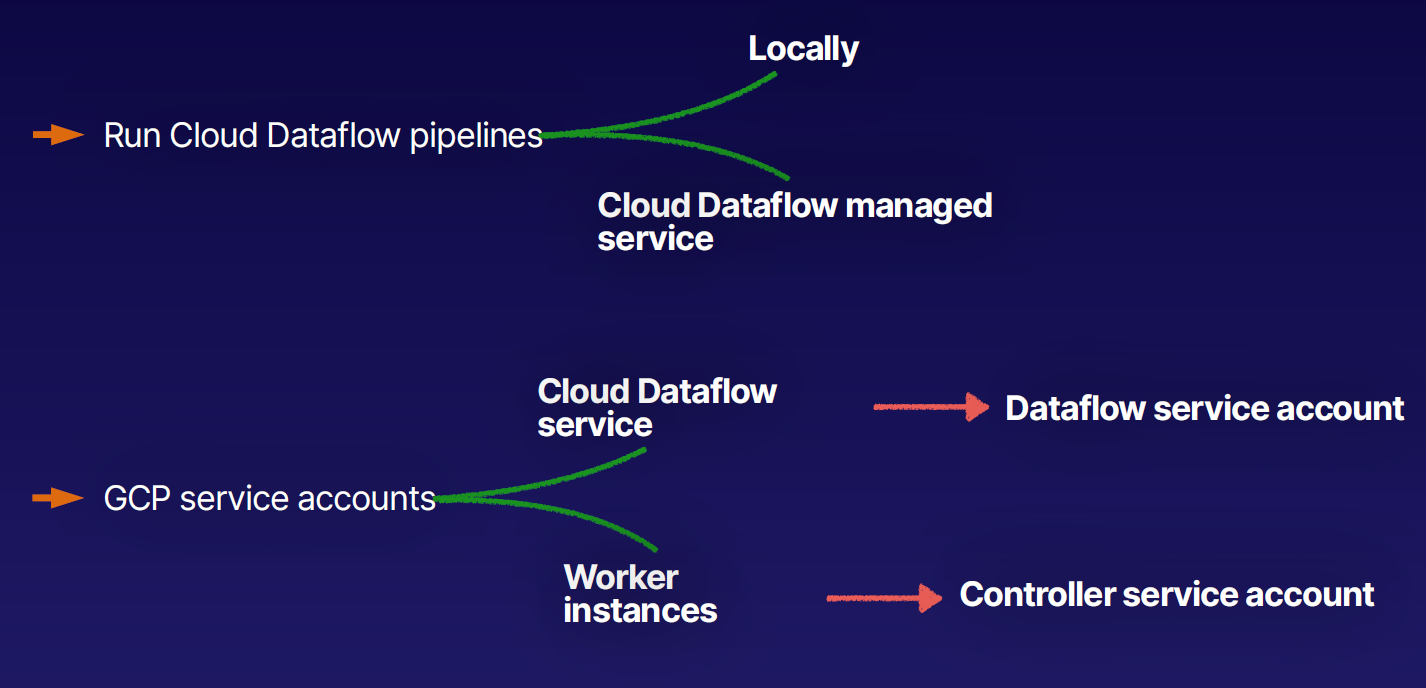
**Watermarks:**

* **Watermarks**, time lag between event time and processing time within ur pipeline.
* **Beam** is used collects all data based on event time based on watermark.

**Trigger types:**

* **Event time**
* **Processing time**
* **Data driven**
* **Compose**

**Security:**

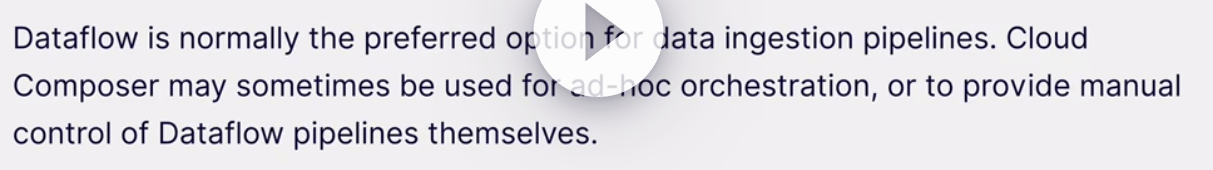


<https://cloud.google.com/dataflow/docs/guides/deploying-a-pipeline>

* Customer-managed encryption keys
* Flexible Resource Scheduling
  + Advanced scheduling
  + Cloud dataflow shuffle services
  + Preemptible Vms
* Migrating MapReduce job to cloud dataflow
* Cloud dataflow with Pub/Sub Seek

Cloud data SQL;

* Develop and run Cloud Dataflow jobs from the BigQuery web UI
* Cloud Dataflow SQL ZetaSQL variant) integrates with Apache Beam SQL
* Apache Beam SQL;
  + Query bounded and unbounded Pcollections
  + Query is converted to a SQL transform
* Cloud Dataflow SQL:
  + Utilise existing SQL skills
  + Join streams with BigQuery tables
  + Query streams or static datasets
  + Write output to BigQuery for analysis andvisualisation



mvn archetype:generate \

-DarchetypeGroupId=org.apache.beam \

-DarchetypeArtifactId=beam-sdks-java-maven-archetypes-examples \

-DarchetypeVersion=2.8.0 -DgroupId=org.example \

-DartifactId=dataflow-lab -Dversion="0.1" \

-Dpackage=org.apache.beam.examples -DinteractiveMode=false

mvn -Pdataflow-runner compile exec:java \

-Dexec.mainClass=org.apache.beam.examples.WordCount \

-Dexec.args="--project=${PROJECT\_ID} \

--stagingLocation=gs://${BUCKET\_NAME}/staging/ \

--output=gs://${BUCKET\_NAME}/output \

--runner=DataflowRunner"

Remember to activate the virtual Python environment in any terminal you use:

cd tweeper

source bin/activate

Activate the Tweep feed:

python tweeper.py

Test the pipeline locally:

python pipeline.py --streaming

Run the pipeline with Cloud Dataflow:

python pipeline.py --streaming --runner DataflowRunner \

--project <YOUR\_PROJECT\_NAME> \

--temp\_location gs://<YOUR\_BUCKET\_NAME>/temp \

--staging\_location gs://<YOUR\_BUCKET\_NAME>/staging \

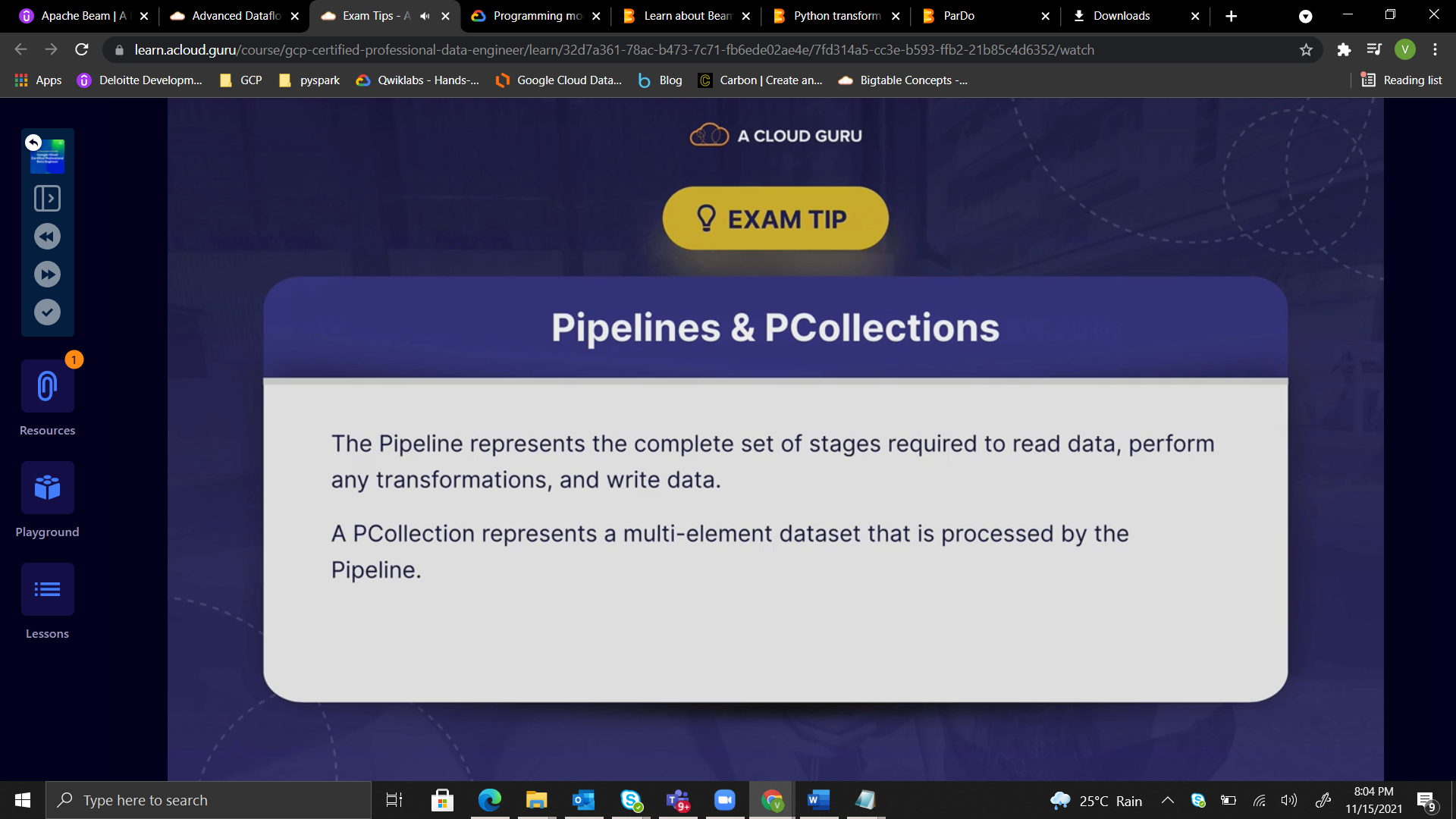
--region us-central1 \

--job\_name tweeps

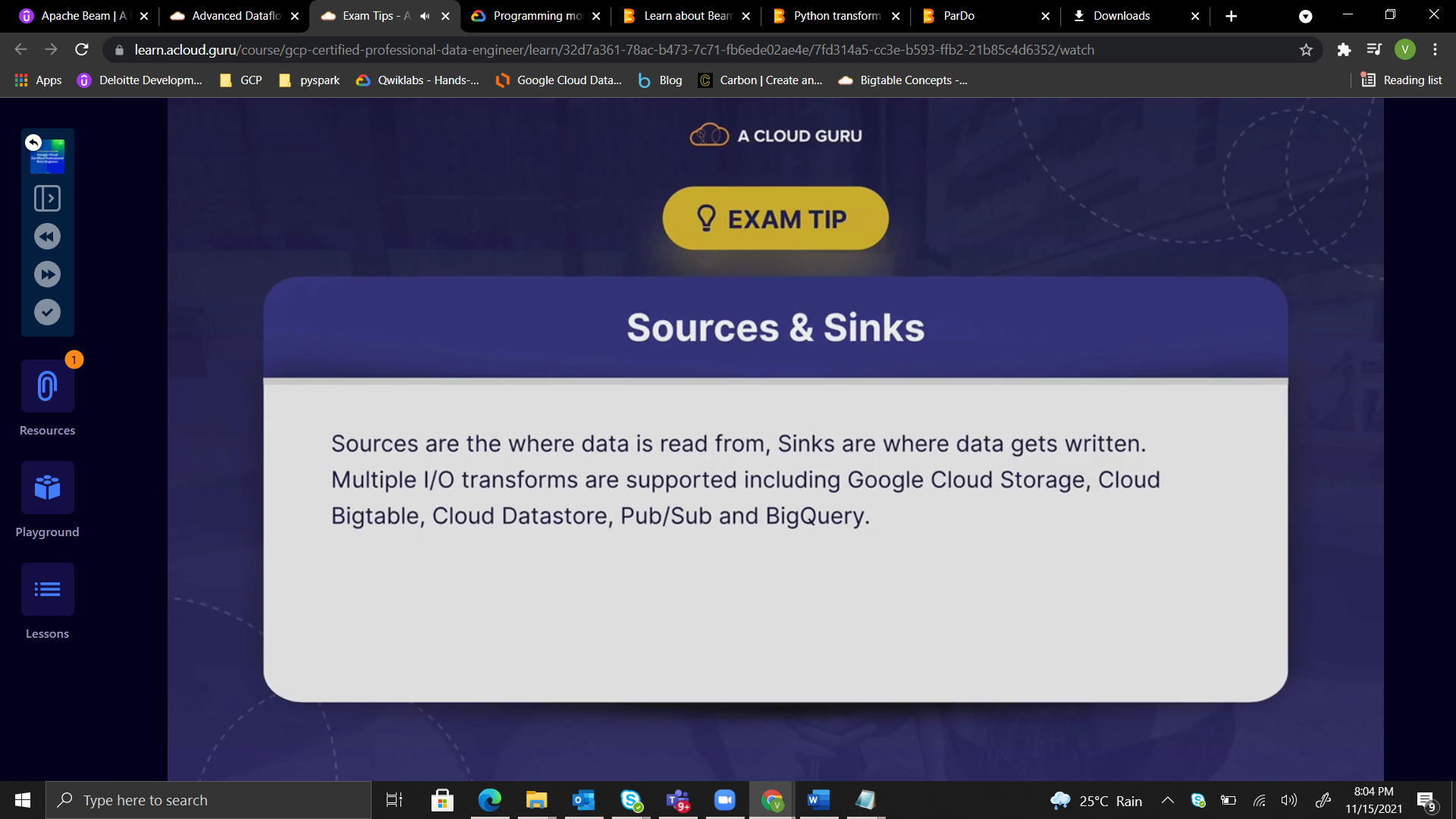
CloudShell commands:

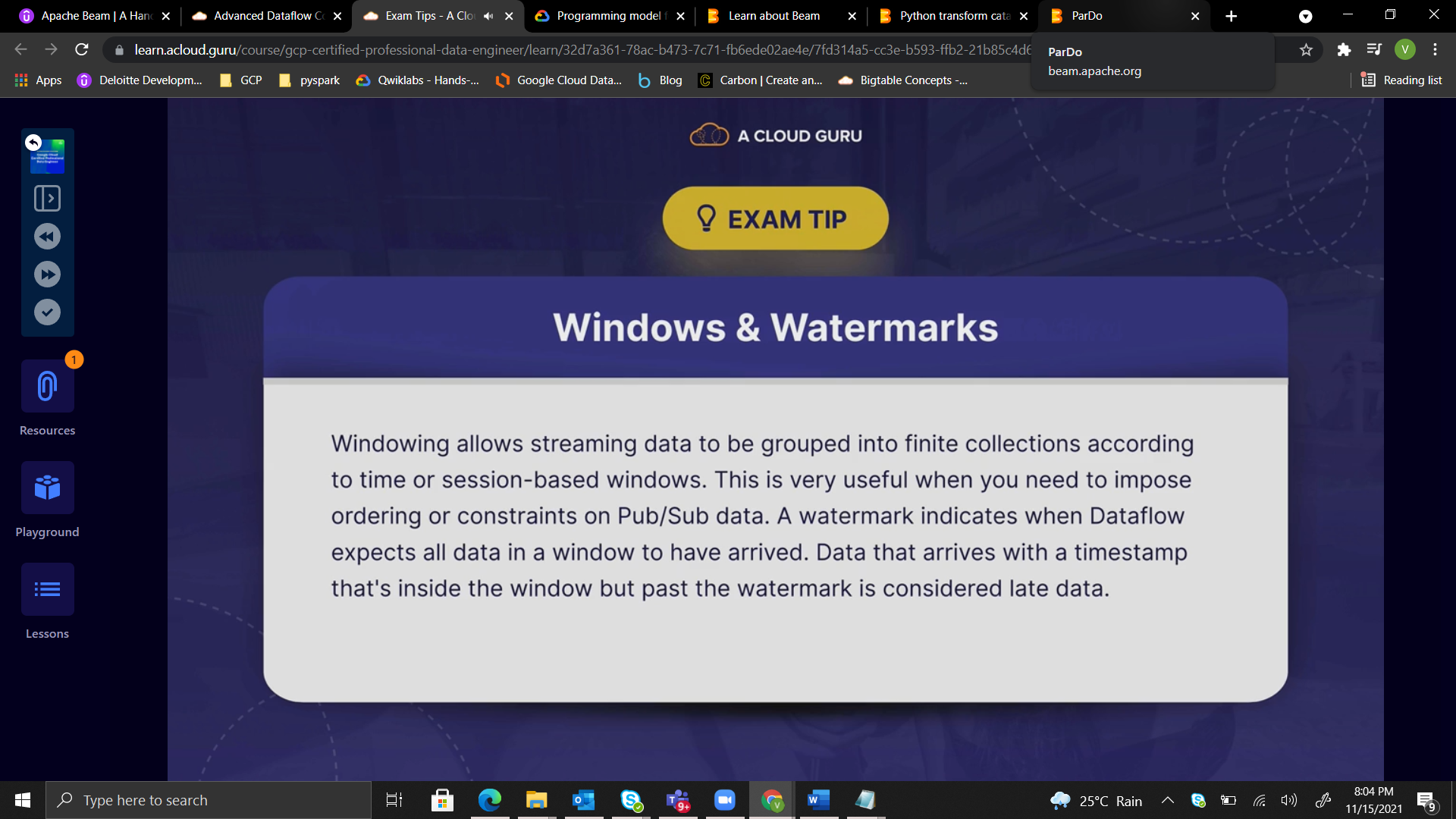
* 1. Create a new virtualenv called tweeper in editor. By running command “virtualenv tweeper”
  2. Cd tweeper
  3. To activate that environemnet source bin/activate
  4. To install files, add that in requirements files at .txt and run command as
     + “pip install -r requirement.txt”









\

