**Lesson Summary**

In this lesson you learned how to use Faust as a Python-based alternative to build Stream Processing applications. You learned:

* How to construct a Faust application
* How to serialize and deserialize data with Python models and dataclasses
* How Faust stores state with Kafka, In-Memory and RocksDB stores
* How to create stream applications with operations and processes
* How to build table applications with windows and co-partitioned streaming data

**Glossary of Key Terms in this Lesson (same as glossary at beginning of lesson)**

* DSL - Domain Specific Language. A metaprogramming language for specific tasks, such as building database queries or stream processing applications.
* Dataclass (Python) - A special type of Class in which instances are meant to represent data, but not contain mutating functions
* Changelog - An append-only log of changes made to a particular component. In the case of Faust and other stream processors, this tracks all changes to a given processor.
* Processor (Faust) - Functions that take a value and return a value. Can be added in a pre-defined list of callbacks to stream declarations.
* Operations (Faust) - Actions that can be applied to an incoming stream to create an intermediate stream containing some modification, such as a group-by or filter

**Stream Processing with Faust**

In this lesson, you will learn how to construct stream processing applications using the Faust framework. Faust was developed at the financial services company, Robinhood, as a Python-native alternative to many of the JVM-only stream processing frameworks like Kafka Streams and Flink. By the end of this lesson, you will know how to create powerful stream processing applications quickly, and with minimal code!

**Robinhood Faust - Key Points**

* [**Built at Robinhood to tackle stream processing problems natively in Python**](https://robinhood.engineering/faust-stream-processing-for-python-a66d3a51212d?gi=25dc91767251)
* Faust takes its design inspiration from [**Kafka Streams, a JVM-only framework**](https://kafka.apache.org/documentation/streams/)
* Faust is built using [**modern Python features like asyncio, and requires Python 3.6 or higher**](https://docs.python.org/3/library/asyncio.html)
* Faust’s API implements the storage, windowing, aggregation, and other concepts discussed in Lesson 5
* Faust is a native Python API, not a Domain Specific Language (DSL) for metaprogramming
* Requires no external dependency other than Kafka. Does not require a resource manager like Yarn or Mesos.
* Faust **does not natively support Avro or Schema Registry**

## Further Optional Research - Exploring Faust Functionality in More Depth

* [**See the Faust documentation for in-depth documentation of how Faust works**](https://faust.readthedocs.io/en/latest/introduction.html)
* Every Faust application has an [**App which instantiates the top-level Faust application**](https://faust.readthedocs.io/en/latest/userguide/application.html#what-is-an-application)
* The application must be assigned a [**topic to subscribe to**](https://faust.readthedocs.io/en/latest/userguide/application.html#app-topic-create-a-topic-description)
* An output [**Table**](https://faust.readthedocs.io/en/latest/userguide/tables.html#id1) or stream receives the output of the processing
* An asynchronous function is decorated with an [**agent**](https://faust.readthedocs.io/en/latest/introduction.html#id6) which register the function as a callback for the application when data is received

**Python Dataclasses**

* A dataclass is a special type of Class in which instances are meant to represent data, but not contain mutating functions.
* Python dataclass objects can be [**marked as frozen**](https://docs.python.org/3/library/dataclasses.html#frozen-instances), which makes them immutable
  + Nothing in Python is truly immutable, but this attribute gets you about as close as you can get
* dataclass objects require type annotations on fields and will enforce those constraints on creation. This helps ensure you’re always working with data in the expected format, reducing and preventing errors.
* Can be paired with the **[asdict function](https://docs.python.org/3/library/dataclasses.html" \l "dataclasses.asdict" \t "_blank)** to quickly transform dataclasses into dictionaries
* [**New in Python 3.7**](https://docs.python.org/3/whatsnew/3.7.html)
* Default to using dataclass to work with data coming into and out of your Faust applications unless you have a good reason not to

**Faust Deserialization**

Topic deserialization in Faust is a simple and painless process. Similar to how you might specify a schema for key and value to confluent\_kafka, with Faust you can provide key and value types. These value types may be primitives such as int or str, or complex types represented as objects.

**Faust Deserialization - Key Points**

* All data model classes must inherit from the **[faust.Record class](https://faust.readthedocs.io/en/latest/userguide/models.html" \l "records" \t "_blank)** if you wish to use them with a Faust topic.
* It is a good idea to specify the **[serializer type on your](https://faust.readthedocs.io/en/latest/userguide/models.html" \l "serialization-deserialization" \t "_blank)** so that Faust will deserialize data in this format by default.
* It is a good practice to set [**validation=True on your data models**](https://faust.readthedocs.io/en/latest/userguide/models.html#model-validation). When set to true, Faust will enforce that the data being deserialized from Kafka matches the expected type.
  + E.g., if we expect a str for a field, but receive an int, Faust will raise an error.
* [**Use Faust codecs to build custom serialization and deserialization**](https://faust.readthedocs.io/en/latest/userguide/models.html#codec-registry)

**Faust Serialization**

Serialization in Faust leverages the same faust.Record rules you just learned about for deserialization. In this section, we’ll learn how Faust manages this process.

**Faust Serialization - Key Points**

* Serialization in Faust leverages the same **[faust.Record](https://faust.readthedocs.io/en/latest/userguide/models.html" \l "records" \t "_blank)** that we saw in the deserialization section. Faust runs the serializer *in reverse* to serialize the data for the output stream.
* [**Multiple serialization codecs may be specified for a given model**](https://faust.readthedocs.io/en/latest/userguide/models.html#manual-serialization)
  + e.g., serialization=”binary|json”. This means, when serializing, encode to json, then base64 encode the data.

## Faust Streams - Key Points

* [**Faust streams are simply infinite asynchronous iterables which perform some processing on incoming messages**](https://faust.readthedocs.io/en/latest/userguide/streams.html#id1)
* [**Faust handles consumption, consumer groups, and offsets for you, in addition to managing message acknowledgements**](https://faust.readthedocs.io/en/latest/userguide/streams.html#id3)
* Faust applications may choose to forward processed messages on to another stream by using the topic.send(<data>) function at the end of the processing loop.

**Faust Operations - Key Points**

* [**Faust Operations are actions that can be applied to an incoming stream to create an intermediate stream containing some modification, such as a group by or filter**](https://faust.readthedocs.io/en/latest/userguide/streams.html#operations)
* The **[group\_by](https://faust.readthedocs.io/en/latest/userguide/streams.html" \l "group-by-repartition-the-stream" \t "_blank)** operation ingests every incoming event from a source topic, and emits it to an intermediate topic with the newly specified key
* The [**filter**](https://faust.readthedocs.io/en/latest/userguide/streams.html#filter-filter-values-to-omit-from-stream) operation uses a boolean function to determine whether or not a particular record should be kept or discarded. Any records that are kept are written to a new intermediate stream.
  + The [**take**](https://faust.readthedocs.io/en/latest/userguide/streams.html#take-buffer-up-values-in-the-stream) operation bundles groups of events before invoking another iteration of the stream. Be careful to specify the within datetime.timedelta argument to this function, otherwise your program may hang.
* Faust provides a number of other operations that you may use when working with your streams. [**Have a look at the documentation for further information**](https://faust.readthedocs.io/en/latest/userguide/streams.html#operations).

## Faust Streams - Summary

In this section you learned how to take advantage of the Faust Streams functionality. You saw how to define stream applications in Python, manipulate incoming data streams with Faust operations, and how to define and apply reusable processing functions.

For in-depth documentation of all the concepts covered in this section of the course, please refer to the excellent [**Faust documentation on streams**](https://faust.readthedocs.io/en/latest/userguide/streams.html)

**Faust Tables**

Faust provides an API for creating stateful applications with streaming Tables.

In this section, you will learn how to fully leverage the Faust Tables API.

**Faust Tables**

* [**Faust tables are defined with app.Table and take a table name and default type argument**](https://faust.readthedocs.io/en/latest/userguide/tables.html#basics).
* Tables must be [**co-partitioned with the streams they are aggregating**](https://faust.readthedocs.io/en/latest/userguide/tables.html#id3). Use the group\_by operation to ensure co-partitioning.
* Tables which are not co-partitioned may lead to inaccurate results.

**Faust Windowing - Key Points**

* Faust supports [**Hopping**](https://faust.readthedocs.io/en/latest/userguide/tables.html#HoppingWindow) and [**Tumbling**](https://faust.readthedocs.io/en/latest/userguide/tables.html#TumblingWindow) windows
* Windowing applies only to Tables
* Faust provides [**semantics for classifying specifically which pool of data is desired from a window**](https://faust.readthedocs.io/en/latest/userguide/tables.html#iterating-over-keys-values-items-in-a-windowed-table), such as current(), now(), relative\_to\_now(), etc.

**Lesson Summary**

In this lesson you learned how to use Faust as a Python-based alternative to build Stream Processing applications. You learned:

* How to construct a Faust application
* How to serialize and deserialize data with Python models and dataclasses
* How Faust stores state with Kafka, In-Memory and RocksDB stores
* How to create stream applications with operations and processes
* How to build table applications with windows and co-partitioned streaming data

**Glossary of Key Terms in this Lesson (same as glossary at beginning of lesson)**

* DSL - Domain Specific Language. A metaprogramming language for specific tasks, such as building database queries or stream processing applications.
* Dataclass (Python) - A special type of Class in which instances are meant to represent data, but not contain mutating functions
* Changelog - An append-only log of changes made to a particular component. In the case of Faust and other stream processors, this tracks all changes to a given processor.
* Processor (Faust) - Functions that take a value and return a value. Can be added in a pre-defined list of callbacks to stream declarations.
* Operations (Faust) - Actions that can be applied to an incoming stream to create an intermediate stream containing some modification, such as a group-by or filter