! Both Spark Streaming and Structured Streaming are components of Apache Spark that enable processing and analyzing real-time streaming data. Let's go through each of them with code examples in Python using PySpark.

### 13. Spark Streaming:

#### DStreams:
DStreams, or Discretized Streams, are the basic abstraction in Spark Streaming. They represent a continuous stream of data divided into small batches, processed by Spark engine, and the results can be manipulated using high-level operations.

```python
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working threads and batch interval of 1 second
ssc = StreamingContext("local[2]", "Spark Streaming Example", 1)

# Create a DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Perform a simple transformation: count the occurrences of each word
word_counts = lines.flatMap(lambda line: line.split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda a, b: a + b)

# Print the result to the console
word_counts.pprint()

# Start the computation
ssc.start()
ssc.awaitTermination()
```

This example sets up a local streaming context, connects to a socket at localhost:9999, splits the lines into words, and counts the occurrences of each word in each batch.

#### Windowed Operations:
Windowed operations allow you to perform operations on a sliding window of DStream data. This is useful for handling data over a specified window of time.

```python
# Create a windowed DStream with a window size of 10 seconds and a sliding interval of 5 seconds
windowed_word_counts = word_counts.window(windowDuration=10, slideDuration=5)
```

```
# Print the result to the console
windowed_word_counts.pprint()

# Start the computation
ssc.start()
ssc.awaitTermination()
```

In this example, we create a windowed DStream that calculates word counts over a 10-second window with a sliding interval of 5 seconds.

### 14. Structured Streaming:

#### Introduction to Structured Streaming:
Structured Streaming is a high-level API built on top of Spark SQL that allows you to express streaming computations using DataFrame and SQL-like operations.

```python
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder.appName("StructuredStreamingExample").getOrCreate()

# Read data from a socket as a streaming DataFrame
lines_df = spark.readStream.format("socket").option("host", "localhost").option("port",
9999).load()

# Perform a simple transformation: count the occurrences of each word
word_counts_df = lines_df.selectExpr("explode(split(value, ' ')) as word") \
                    .groupBy("word") \
                    .count()

# Display the result to the console
query = word_counts_df.writeStream.outputMode("complete").format("console").start()
query.awaitTermination()
```

This example reads data from a socket as a streaming DataFrame, explodes the lines into words, and calculates word counts using DataFrame operations.

#### Handling Streaming Data with DataFrames:
Structured Streaming allows you to handle streaming data in a way similar to static data. You can use the same DataFrame API to process both streaming and batch data.

```python
# Read streaming data from a Kafka topic
streaming_df = spark.readStream.format("kafka").option("kafka.bootstrap.servers",
"localhost:9092").option("subscribe", "topic").load()

# Perform operations on the streaming DataFrame
result_df = streaming_df.selectExpr("key", "CAST(value AS STRING) as value")

# Write the result to the console
query = result_df.writeStream.outputMode("append").format("console").start()
query.awaitTermination()
```

In this example, streaming data is read from a Kafka topic, and operations are performed using the DataFrame API, followed by writing the result to the console.

These examples provide a basic understanding of Spark Streaming and Structured Streaming in PySpark. Depending on your use case, you may need to adapt and extend these examples to suit your specific requirements.

Certainly! Let's dive deeper into some advanced features and concepts for both Spark Streaming and Structured Streaming.

### 13. Spark Streaming (Continued):

#### Checkpointing:
Checkpointing is crucial for fault-tolerance in Spark Streaming. It allows the system to recover from failures and ensures that the processed data is saved to a reliable distributed file system.

```python
# Set the checkpoint directory
ssc.checkpoint("hdfs://path/to/checkpoint")

# ... (define DStream operations as before)

# Start the computation
ssc.start()
ssc.awaitTermination()
```

In this example, the checkpoint directory is set, enabling Spark Streaming to recover from failures.

#### Stateful Operations:
Spark Streaming supports stateful operations, where you can maintain state across batches. This is useful for scenarios like tracking counts over time.

```python
# Update state by key
stateful_word_counts = word_counts.updateStateByKey(lambda new_values, last_state:
sum(new_values) + (last_state or 0))

# Print the result to the console
stateful_word_counts.pprint()

# Start the computation
ssc.start()
ssc.awaitTermination()
```

Here, `updateStateByKey` is used to maintain a running count of words.

### 14. Structured Streaming (Continued):

#### Event Time Processing:
Structured Streaming supports event time processing, allowing you to handle events based on their timestamps rather than arrival times. This is crucial for correctly processing out-of-order data.

```python
from pyspark.sql.functions import window, expr

# Read streaming data with timestamp column
streaming_df = spark.readStream.format("kafka").option("kafka.bootstrap.servers",
"localhost:9092").option("subscribe", "topic").load()

# Define watermark and windowed aggregation
result_df = streaming_df \
    .selectExpr("CAST(value AS STRING) as value", "timestamp") \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(window("timestamp", "5 minutes"), "value") \
    .count()

# Write the result to the console
query = result_df.writeStream.outputMode("append").format("console").start()
query.awaitTermination()
```

In this example, a watermark is set on the timestamp column, and windowed aggregation is performed based on event time.

#### Output Modes:
Structured Streaming supports different output modes such as `append`, `update`, and `complete`. The choice depends on the type of aggregation and the desired output.

```python
query = result_df.writeStream.outputMode("append").format("console").start()
# or
query = result_df.writeStream.outputMode("update").format("console").start()
# or
query = result_df.writeStream.outputMode("complete").format("console").start()

query.awaitTermination()
```

Choose the appropriate output mode based on your use case. For example, `update` mode is suitable for aggregations where only the changed rows are written.

These additional examples provide insights into more advanced features and considerations when working with Spark Streaming and Structured Streaming. Adjust and combine these concepts based on your specific streaming data processing requirements.