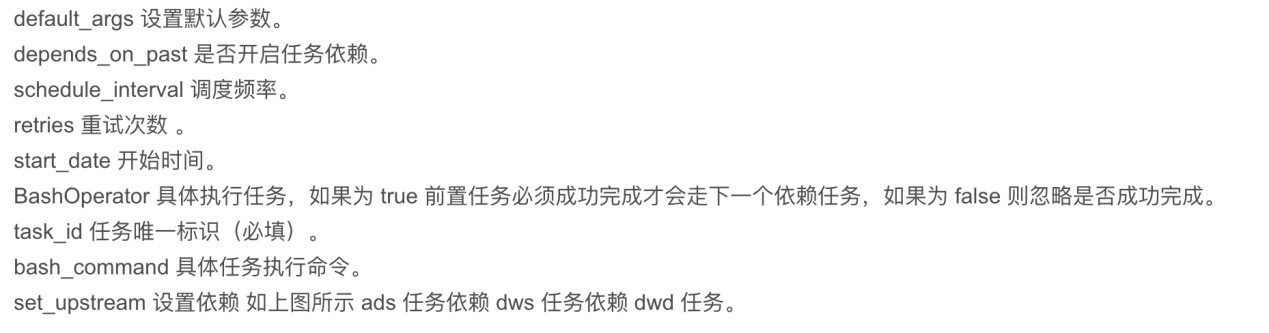
Airflow

**Airflow** is a workflow automation and scheduling system that allows data engineers to author, schedule, and monitor workflows as Directed Acyclic Graphs (DAGs). It helps automate complex data processing tasks.

**DAG** (Directed Acyclic Graph) collects Tasks together, organized with dependencies and relationships to say how they should run. You can use python to define DAG, utilizing the DAG class and Task objects.

Operators define the tasks to be executed, usually we use BashOperator. Operators use hook to interact with external system, such as databases or message queues.



retry\_delay: time interval between retries

task = BashOperator(

task\_id='example\_task',

bash\_command='exit 1',

retries=3,

retry\_delay=timedelta(minutes=5),

)

**Executors** determine how tasks are executed and scheduled. Common executors include LocalExecutor (local execution), CeleryExecutor (distributed execution), and KubernetesExecutor (execution on Kubernetes).

**XCom** (cross-communication) is a mechanism to allow tasks to exchange messages. Tasks can push and pull data to/from XCom using xcom\_push and xcom\_pull.

Kafka

**Producers, Consumers, Topics, Partitions, Brokers**

Producers send messages to topics, Consumers read messages from topics, and Brokers store and manage messages.

Producers send messages to Kafka topics, and Consumers subscribe to topics to consume messages.

Messages are distributed across partitions, and Producers and Consumers interact with Brokers to read/write data.

A **Partition** is a subset of a Kafka topic. Partitioning allows parallel processing of messages, enhancing performance and scalability by distributing load across multiple servers.

An **Offset** is the position of a message within a partition. Consumers use offsets to track which messages have been read. Offsets can be managed automatically or manually, typically using Kafka's consumer group mechanism.

**Topics** are categories of messages, and messages are the data records sent to topics. Messages within topics are stored in partitions, which are ordered logs of messages.

Handling duplicates and message loss involves setting proper consumer configurations like enable.auto.commit and auto.offset.reset, using idempotent producers, and employing transactional mechanisms to ensure exactly-once delivery semantics.

Python

**List comprehension** is used to create a list using a single line of code. for example,

list = [x for x in range(10) if x < 5]

A **decorator** is a function that modifies the behavior of another function without changing its code. Decorators are used to add functionality like logging, access control, and more.

**Generators** are functions that use yield to produce a series of values lazily, meaning they generate values one at a time as needed, which can save memory and improve performance, especially with large datasets.

The **GIL** is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes simultaneously. It limits the performance of CPU-bound multithreaded programs but has less impact on I/O-bound programs.

Exceptions are handled using **try, except, and finally** blocks.