**Data Pipelines**

**Defining New Terms**

The video above includes references to a few terms that you may not be familiar with. Below are some definitions that you might find useful.

**Extract Transform Load (ETL) and Extract Load Transform (ELT):**

"ETL is normally a continuous, ongoing process with a well-defined workflow. ETL first extracts data from homogeneous or heterogeneous data sources. Then, data is cleansed, enriched, transformed, and stored either back in the lake or in a data warehouse.

"ELT (Extract, Load, Transform) is a variant of ETL wherein the extracted data is first loaded into the target system. Transformations are performed after the data is loaded into the data warehouse. ELT typically works well when the target system is powerful enough to handle transformations. Analytical databases like Amazon Redshift and Google BigQ."  
*Source:* [Xplenty.com](https://www.xplenty.com/blog/etl-vs-elt/)

This [Quora post](https://www.quora.com/What-is-the-difference-between-the-ETL-and-ELT) is also helpful if you'd like to read more.

**What is S3?**

"Amazon S3 has a simple web services interface that you can use to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites."  
*Source:* [Amazon Web Services Documentation](https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html).

If you want to learn more, start [here](https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html).

**What is Kafka?**

"Apache Kafka is an **open-source stream-processing software platform** developed by Linkedin and donated to the Apache Software Foundation, written in Scala and Java. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. Its storage layer is essentially a massively scalable pub/sub message queue designed as a distributed transaction log, making it highly valuable for enterprise infrastructures to process streaming data."  
*Source:* *Wikipedia*.

If you want to learn more, start [here](https://kafka.apache.org/intro).

**What is RedShift?**

"Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud. You can start with just a few hundred gigabytes of data and scale to a petabyte or more... The first step to create a data warehouse is to launch a set of nodes, called an Amazon Redshift cluster. After you provision your cluster, you can upload your data set and then perform data analysis queries. Regardless of the size of the data set, Amazon Redshift offers fast query performance using the same SQL-based tools and business intelligence applications that you use today.

If you want to learn more, start [here](https://docs.aws.amazon.com/redshift/latest/mgmt/welcome.html).

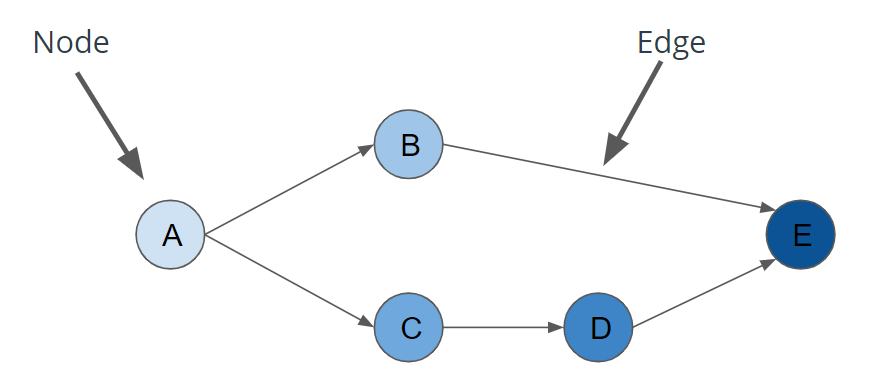
So in other words, S3 is an example of the final data store where data might be loaded (e.g. ETL). While Redshift is an example of a data warehouse product, provided specifically by Amazon.

#### Data Validation

Data Validation is the process of ensuring that data is present, correct & meaningful. Ensuring the quality of your data through automated validation checks is a critical step in building data pipelines at any organization.

**Definitions**

* **Directed Acyclic Graphs (DAGs):** DAGs are a special subset of graphs in which the edges between nodes have a specific direction, and no cycles exist. When we say “no cycles exist” what we mean is the nodes cant create a path back to themselves.
* **Nodes:** A step in the data pipeline process.
* **Edges:** The dependencies or relationships other between nodes.

[[](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/574d6d41-2b84-40bb-be7e-439c9682c2c6)](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/574d6d41-2b84-40bb-be7e-439c9682c2c6)

[Diagram of a Directed Acyclic Graph](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/574d6d41-2b84-40bb-be7e-439c9682c2c6)

**Common Questions**

**Are there real world cases where a data pipeline is not DAG?**

It is possible to model a data pipeline that is not a DAG, meaning that it contains a cycle within the process. However, the vast majority of use cases for data pipelines can be described as a directed acyclic graph (DAG). This makes the code more understandable and maintainable.

**Can we have two different pipelines for the same data and can we merge them back together?**

Yes. It's not uncommon for a data pipeline to take the same dataset, perform two different processes to analyze the it, then merge the results of those two processes back together.

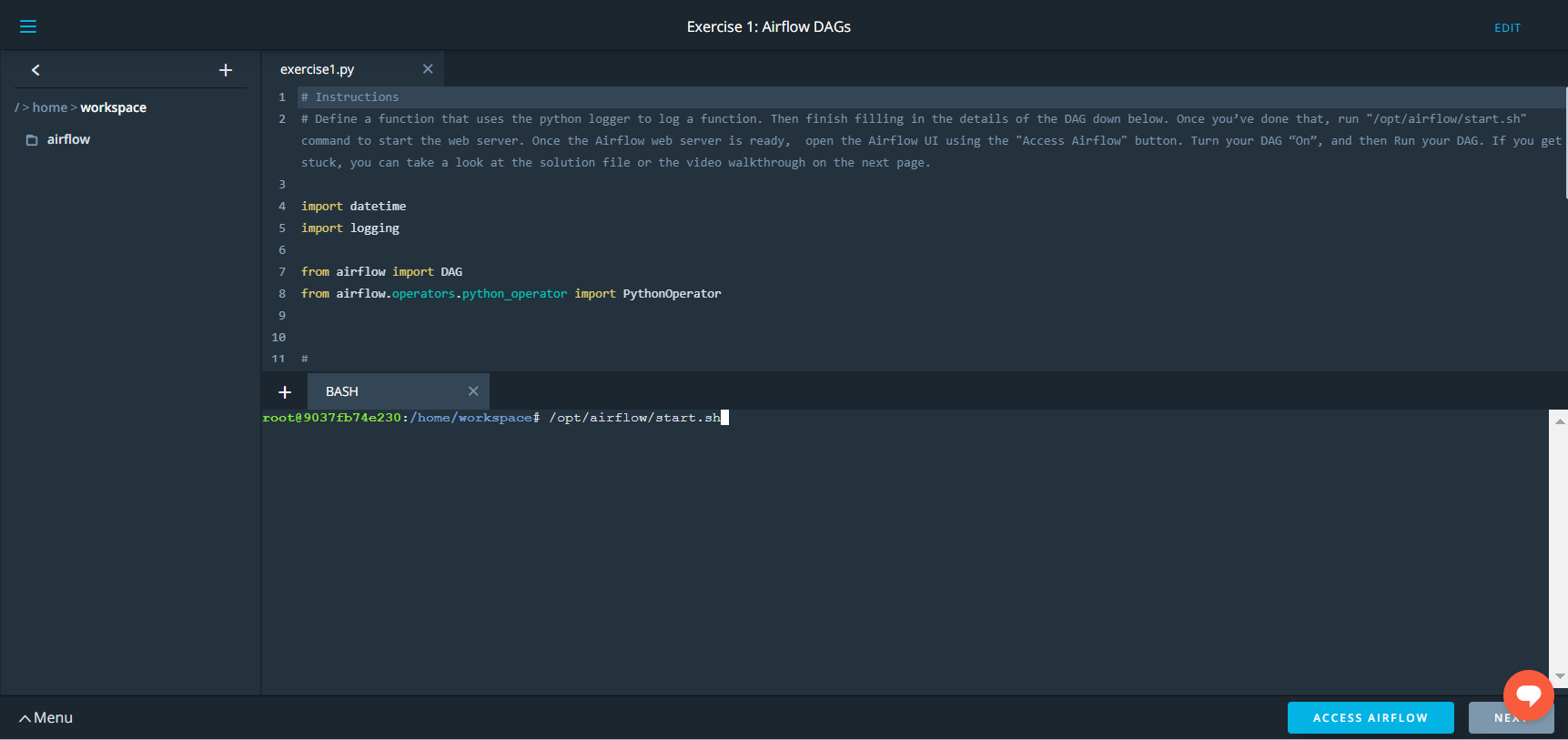
**Tips for Using Airflow's Web Server**

* Use Google Chrome to view the Web Server. Airflow sometimes has issues rendering correctly in Firefox or other browers.
* Make sure you toggle the DAG to On before you try an run it. Otherwise you'll see your DAG running, but it won't ever finish.

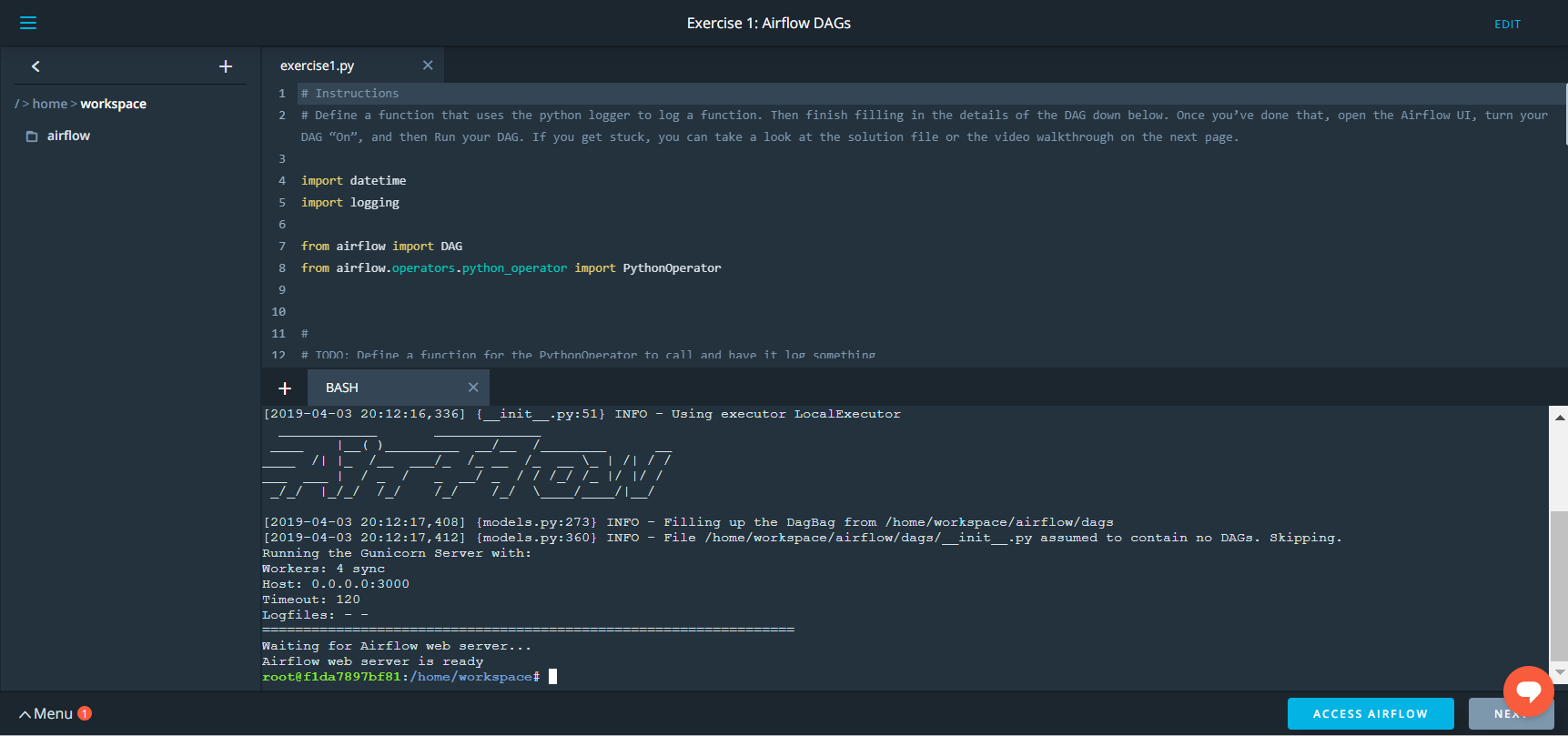
## Workspace Instructions

Before you start on your first exercise, please note the following instruction.

1. After you have updated the DAG, you will need to run /opt/airflow/start.sh command to start the Airflow webserver. See the screenshot below for the Exercise 1 Workspace.

**[[](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/2ff09459-10ca-40fb-8ab6-1f505f7c898b)](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/2ff09459-10ca-40fb-8ab6-1f505f7c898b)**

1. Wait for the Airflow web server to be ready (see screenshot below).

**[[](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/2ff09459-10ca-40fb-8ab6-1f505f7c898b)](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/2ff09459-10ca-40fb-8ab6-1f505f7c898b)**

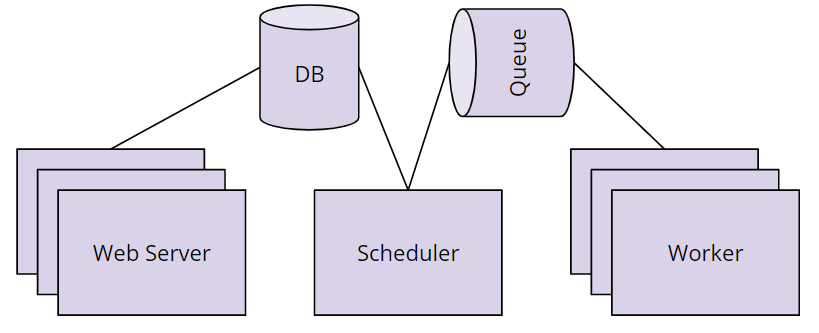
1. Access the Airflow UI by clicking on the blue "Access Airflow" button.

This should be able to access the Airflow UI without any delay.

**Please note:** Because the files located in the s3 bucket 'udacity-dend' are very large, Airflow can take up to 10 minutes to make the connection.

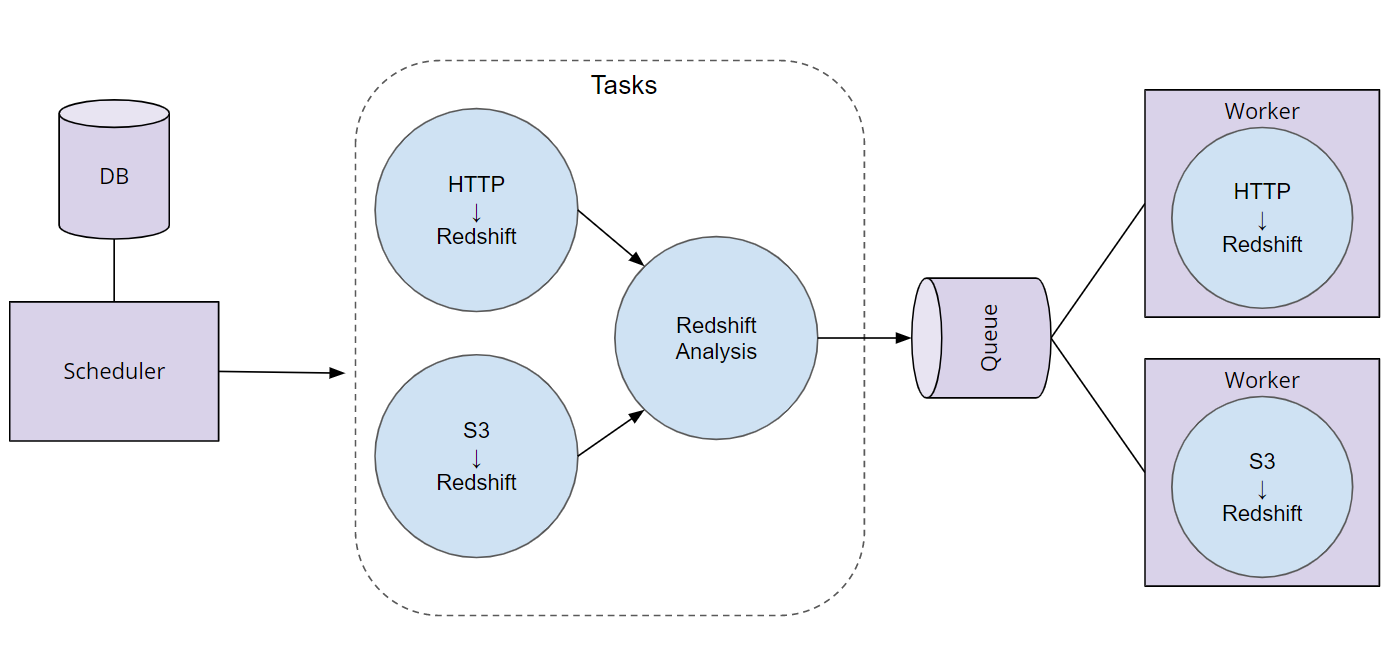
Note: Ben refers to "Coco" in the video. He's referring to using an Airflow Workspace within the Udacity classroom.

### Components of Airflow

[[](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/f5fd1f7f-6eac-4551-a490-48414e93ad41)](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/f5fd1f7f-6eac-4551-a490-48414e93ad41)

* **Scheduler** orchestrates the execution of jobs on a trigger or schedule. The Scheduler chooses how to prioritize the running and execution of tasks within the system. You can learn more about the Scheduler from the official [Apache Airflow documentation](https://airflow.apache.org/scheduler.html).
* **Work Queue** is used by the scheduler in most Airflow installations to deliver tasks that need to be run to the **Workers**.
* **Worker** processes execute the operations defined in each DAG. In most Airflow installations, workers pull from the **work queue** when it is ready to process a task. When the worker completes the execution of the task, it will attempt to process more work from the **work queue** until there is no further work remaining. When work in the queue arrives, the worker will begin to process it.
* **Database** saves credentials, connections, history, and configuration. The database, often referred to as the metadata database, also stores the state of all tasks in the system. Airflow components interact with the database with the Python ORM, [SQLAlchemy](https://www.sqlalchemy.org/).
* **Web Interface** provides a control dashboard for users and maintainers. Throughout this course you will see how the web interface allows users to perform tasks such as stopping and starting DAGs, retrying failed tasks, configuring credentials, The web interface is built using the [Flask web-development microframework](http://flask.pocoo.org/).

### How Airflow Works

[[](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/f5fd1f7f-6eac-4551-a490-48414e93ad41)](https://classroom.udacity.com/nanodegrees/nd027/parts/45d1c3b1-d87b-4578-a6d0-7e86bb5fea6c/modules/1fa8af52-682d-440b-a391-16183cb0df08/lessons/af88b28f-ceb2-4239-8262-2a55a3b67bd5/concepts/f5fd1f7f-6eac-4551-a490-48414e93ad41)

### Order of Operations For an Airflow DAG

* The Airflow Scheduler starts DAGs based on time or external triggers.
* Once a DAG is started, the Scheduler looks at the steps within the DAG and determines which steps can run by looking at their dependencies.
* The Scheduler places runnable steps in the queue.
* Workers pick up those tasks and run them.
* Once the worker has finished running the step, the final status of the task is recorded and additional tasks are placed by the scheduler until all tasks are complete.
* Once all tasks have been completed, the DAG is complete.

**Creating a DAG**

Creating a DAG is easy. Give it a name, a description, a start date, and an interval.

**from** airflow **import** DAG

divvy\_dag = DAG(

'divvy',

**description**='Analyzes Divvy Bikeshare Data',

start\_date=datetime(2019, 2, 4),

schedule\_interval='@daily')

**Creating Operators to Perform Tasks**

**Operators** define the atomic steps of work that make up a DAG. Instantiated operators are referred to as **Tasks**.

**from** airflow **import** DAG

**from** airflow.operators.python\_operator **import** PythonOperator

**def** **hello\_world**():

print(“Hello World”)

divvy\_dag = DAG(...)

task = PythonOperator(

task\_id=’hello\_world’,

python\_callable=hello\_world,

dag=divvy\_dag)

**Schedules**

**Schedules** are optional, and may be defined with cron strings or Airflow Presets. Airflow provides the following presets:

* @once - Run a DAG once and then never again
* @hourly - Run the DAG every hour
* @daily - Run the DAG every day
* @weekly - Run the DAG every week
* @monthly - Run the DAG every month
* @yearly- Run the DAG every year
* None - Only run the DAG when the user initiates it

**Start Date:** If your start date is in the past, Airflow will run your DAG as many times as there are schedule intervals between that start date and the current date.

**End Date:** Unless you specify an optional end date, Airflow will continue to run your DAGs until you disable or delete the DAG.

**Operators**

Operators define the atomic steps of work that make up a DAG. Airflow comes with many Operators that can perform common operations. Here are a handful of common ones:

* PythonOperator
* PostgresOperator
* RedshiftToS3Operator
* S3ToRedshiftOperator
* BashOperator
* SimpleHttpOperator
* Sensor

**Task Dependencies**

In Airflow DAGs:

* Nodes = Tasks
* Edges = Ordering and dependencies between tasks

Task dependencies can be described programmatically in Airflow using >> and <<

* a >> b means a comes before b
* a << b means a comes after b

hello\_world\_task = PythonOperator(task\_id=’hello\_world’, **...**)

goodbye\_world\_task = PythonOperator(task\_id=’goodbye\_world’, **...**)

**...**

*# Use >> to denote that goodbye\_world\_task depends on hello\_world\_task*

hello\_world\_task >> goodbye\_world\_task

Tasks dependencies can also be set with “set\_downstream” and “set\_upstream”

* a.set\_downstream(b) means a comes before b
* a.set\_upstream(b) means a comes after b

hello\_world\_task = PythonOperator(task\_id=’hello\_world’, **...**)

goodbye\_world\_task = PythonOperator(task\_id=’goodbye\_world’, **...**)

**...**

hello\_world\_task.set\_downstream(goodbye\_world\_task)

NEXT

**Connection via Airflow Hooks**

Connections can be accessed in code via hooks. Hooks provide a reusable interface to external systems and databases. With hooks, you don’t have to worry about how and where to store these connection strings and secrets in your code.

**from** airflow **import** DAG

**from** airflow.hooks.postgres\_hook **import** PostgresHook

**from** airflow.operators.python\_operator **import** PythonOperator

**def** **load**():

*# Create a PostgresHook option using the `demo` connection*

db\_hook = PostgresHook(‘demo’)

df = db\_hook.get\_pandas\_df('SELECT \* FROM rides')

print(f'Successfully used PostgresHook to return {len(df)} records')

load\_task = PythonOperator(task\_id=’load’, python\_callable=hello\_world, ...)

Airflow comes with many Hooks that can integrate with common systems. Here are a few common ones:

* HttpHook
* PostgresHook (works with RedShift)
* MySqlHook
* SlackHook
* PrestoHook

**Correction to Video**

The instructions in the video are incorrect about the S3 bucket.

You should create two Airflow variables in the UI:

* s3\_bucket: udacity-dend
* s3\_prefix: data-pipelines

Using this S3 prefix filters the returned S3 keys, so the number of objects displayed for that bucket will be greatly reduced, and you'll avoid the UI freezing.

# Solution 4: Connections and Hooks

Below is the solution for Exercise 4: Connections and Hooks.

**import** datetime

**import** logging

**from** airflow **import** DAG

**from** airflow.models **import** Variable

**from** airflow.operators.python\_operator **import** PythonOperator

**from** airflow.hooks.S3\_hook **import** S3Hook

**def** **list\_keys**():

hook = S3Hook(aws\_conn\_id='aws\_credentials')

bucket = Variable.get('s3\_bucket')

prefix = Variable.get('s3\_prefix')

logging.info(f"Listing Keys from {bucket}/{prefix}")

keys = hook.list\_keys(bucket, prefix=prefix)

**for** key **in** keys:

logging.info(f"- s3://{bucket}/{key}")

dag = DAG(

'lesson1.exercise4',

start\_date=datetime.datetime.now())

list\_task = PythonOperator(

task\_id="list\_keys",

python\_callable=list\_keys,

dag=dag

)

Correction: At timestamp 0.05 in the video above, the ending square bracket is missing. It should be  
python print(f"Hello {kwargs['execution\_date']}")

[**Here**](https://airflow.apache.org/macros.html) is the Apache Airflow documentation on **context variables** that can be included as kwargs.

Here is a link to a [**blog post**](https://blog.godatadriven.com/zen-of-python-and-apache-airflow) that also discusses this topic.

### Runtime Variables

Airflow leverages templating to allow users to “fill in the blank” with important runtime variables for tasks.

**from** airflow **import** DAG

**from** airflow.operators.python\_operator **import** PythonOperator

**def** **hello\_date**(\*args, \*\*kwargs):

print(f“Hello {kwargs[‘execution\_date’]}”)

divvy\_dag = DAG(...)

task = PythonOperator(

task\_id=’hello\_date’,

python\_callable=hello\_date,

provide\_context=**True**,

dag=divvy\_dag)

The link for the Airflow documentation on context variables has changed since the video was created. Here is the new link: [**https://airflow.apache.org/macros.html**](https://airflow.apache.org/macros.html)

# Solution 6: Build the S3 to Redshift Dag

Below is the solution for Exercise 6: Build the S3 to Redshift Dag.

**import** datetime

**import** logging

**from** airflow **import** DAG

**from** airflow.contrib.hooks.aws\_hook **import** AwsHook

**from** airflow.hooks.postgres\_hook **import** PostgresHook

**from** airflow.operators.postgres\_operator **import** PostgresOperator

**from** airflow.operators.python\_operator **import** PythonOperator

**import** sql\_statements

**def** **load\_data\_to\_redshift**(\*args, \*\*kwargs):

aws\_hook = AwsHook("aws\_credentials")

credentials = aws\_hook.get\_credentials()

redshift\_hook = PostgresHook("redshift")

redshift\_hook.run(sql.COPY\_ALL\_TRIPS\_SQL.format(credentials.access\_key, credentials.secret\_key))

dag = DAG(

'lesson1.exercise6',

start\_date=datetime.datetime.now()

)

create\_table = PostgresOperator(

task\_id="create\_table",

dag=dag,

postgres\_conn\_id="redshift",

sql=sql\_statements.CREATE\_TRIPS\_TABLE\_SQL

)

copy\_task = PythonOperator(

task\_id='load\_from\_s3\_to\_redshift',

dag=dag,

python\_callable=load\_data\_to\_redshift

)

location\_traffic\_task = PostgresOperator(

task\_id="calculate\_location\_traffic",

dag=dag,

postgres\_conn\_id="redshift",

sql=sql\_statements.LOCATION\_TRAFFIC\_SQL

)

create\_table >> copy\_task

copy\_task >> location\_traffic\_task