

# Airflow DAG: Creating your first DAG in 5 minutes

2 Comments / Apache Airflow, dag / By marclamberti

Looking for creating your first Airflow DAG? Wondering how to process your data in Airflow? What are the steps to code your own data pipelines? You've come to the right place! At the end of this short tutorial, you will be able to code your first Airflow DAG! You might think it's hard to start with Apache Airflow but it is not. The truth is, Airflow is so powerful that the possibilities it brings can be overwhelming. Don't worry, you are going to discover only what you need to get started now! Before jumping in, if you are looking for a solid and more complete introduction to Airflow, check my course here, you will enjoy it 😊 . Ready? You're sure? Let's go!

Avaya




  Download The White Paper

Table of Contents 

1. Use Case

2. Airflow DAG? Operators? Terminologies

2.0.1. What is a DAG?

2.0.2. What is Airflow Operator?

2.0.3. Dependencies?

3. Coding your first Airflow DAG

3.0.1. Step 1: Make the Imports

3.0.2. Step 2: Create the Airflow DAG object

3.0.3. Step 3: Add your tasks!

3.0.3.1. Training model tasks

3.0.3.2. Choosing best model

3.0.3.3. Accurate or inaccurate?

3.0.4. Step 4: Defining dependencies

3.0.5. The Final Airflow DAG!



4. Creating your first DAG in action!

5. Conclusion

## Use Case

As usual, the best way to understand a feature/concept is to have a use case. Let's say, you have the following data pipeline in mind:

Make

  Zero Coding Experience Needed



Data pipeline

Your goal is to train 3 different machine learning models, then choose the best one and execute either accurate or inaccurate based on the accuracy of the best model. You could even store the value in a database, but let’s keep things simple for now. Now, everything is clear in your head, the first question comes up:

How can I create an Airflow DAG representing my data pipeline?

Well, this is exactly what you are about to find out now!



## Airflow DAG? Operators? Terminologies

Before jumping into the code, you need to get used to some terminologies first. I know, the boring part, but stay with me, it is important.

First and foremost,

What is a DAG?

DAG stands for Directed Acyclic Graph. In simple terms, it is a graph with nodes, **directed** edges and **no cycles**. Basically, this:

is a **DAG**.

But this:

is **NOT** a DAG. Why? Because there is a cycle. As Node A depends on Node C which it turn, depends on Node B and itself on Node A, this DAG (which is not) won't run at all. A DAG has no cycles, never. Last but not least, a DAG is a data pipeline in Apache Airflow. So, whenever you read "DAG", it means "data pipeline". Last but not least, when a DAG is triggered, a **DAGRun** is created. A DAGRun is an instance of your DAG with an execution date in Airflow.

Ok, once you know what is a DAG, the next question is, what is a "Node" in the context of Airflow?

## What is Airflow Operator?

In an Airflow DAG, nodes are operators. In other words, a task in your DAG is an operator. An Operator is a class encapsulating the logic of what you want to achieve. For example, you want to execute a python function, you will use the PythonOperator. You want to execute a Bash command, you will use the BashOperator. Airflow brings a ton of operators that you can find [here](#) and [here](#). When an operator is triggered, it becomes a task, and more specifically a

```
1 from airflow.operators.dummy import DummyOperator
2 from airflow.operators.bash import BashOperator
3 # The DummyOperator is a task and does nothing
4 accurate = DummyOperator(
5     task_id='accurate'
6 )
7 # The BashOperator is a task to execute a bash command
8 commands = BashOperator(
9     task_id='commands'
10    bash_command='sleep 5'
11 )
```

As you can see, an Operator has some arguments. The first one is the *task\_id*. The *task\_id* is the **unique** identifier of the operator **in** the DAG. Each Operator must have a unique *task\_id*. The other arguments to fill in depend on the operator used. For example, with the BashOperator, you have to pass the bash command to execute. With the DummyOperator, there is nothing else to specify. At the end, to know what arguments your Operator needs, the documentation is your friend.

You know what is a DAG and what is an Operator. Time to know how to create the directed edges, or in other words, the dependencies between tasks.

## Dependencies?

As you learned, a DAG has directed edges. Those directed edges are the dependencies in an Airflow DAG between all of your operators/tasks. Basically, if you want to say "Task A is executed before Task B", you have to defined the corresponding dependency. How?

```
task_a >> task_b
# Or
task_b << task_a
```

The >> and << respectively mean "right bitshift" and "left bitshift" or "set downstream task" and "set upstream task". In the example, on the first line we say that task\_b is a downstream task to task\_a. On the second line we say that task\_a is an upstream task of task\_b. Don't worry, we will come back at dependencies.

All right, now you got the terminologies, time to dive into the code! A disclaimer part 🙏

Data pipeline

Without further do, let's begin!

Step 1: Make the Imports

The first step is to import the classes you need. To create a DAG in Airflow, you always have to import the DAG class. After the DAG class, come the imports of Operators. Basically, for each Operator you want to use, you have to make the corresponding import. For example, you want to execute a Python function, you have to import the PythonOperator. You want to execute a bash command, you have to import the BashOperator. Finally, the last import is usually the datetime class as you need to specify a start date to your DAG.

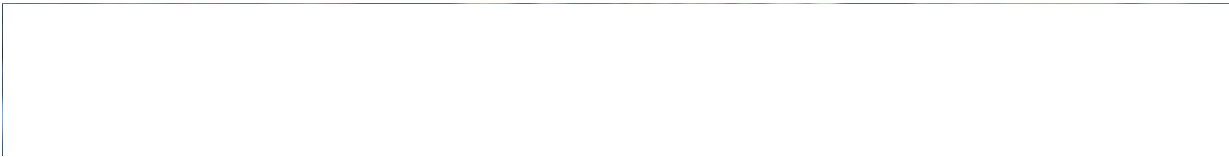
```
1 from airflow import DAG
2 from airflow.operators.python import PythonOperator, BranchPythonOperator
3 from airflow.operators.bash import BashOperator
4 from datetime import datetime
```

Step 2: Create the Airflow DAG object

After having made the imports, the second step is to create the Airflow DAG object. A DAG object must have two parameters, a *dag\_id* and a *start\_date*.

For example, if your DAG starts to execute from a date 3 years ago, you might end up with many DAG runs running at the same time.

In addition those arguments, 2 others are usually specified. The *schedule\_interval* and the *catchup* arguments.



The *schedule\_interval* defines **the interval of time at which your DAG gets triggered**. Every 10 mins, every day, every month and so on. 2 ways to define it, either with a CRON expression or with a *timedelta* object. The first option is the most often used. By the way, if you don't know how to define a CRON expression, take a look at this beautiful website and if you don't know what a CRON expression is, keep in mind that it is way to express time intervals.

Lastly, the *catchup* argument allows you to prevent from backfilling automatically the non triggered DAG Runs between the start date of your DAG and the current date. If you want don't want to end up with many DAG runs running at the same time, it's usually a best practice to set it to *False*.

That's it, no more arguments and here is the corresponding code,

```
1  with DAG("my_dag", # Dag id
2      start_date=datetime(2021, 1 ,1), # start date, the 1st of January 2021
3      schedule_interval='@daily', # Cron expression, here it is a preset of Airflow, @daily means once every day
4      catchup=False # Catchup
5  ) as dag:
```

Notice that to create an instance of a DAG, we use the *with* statement. Why? Because “with” is a context manager and allows you to better manager objects. In that case, a DAG object. I won't go into the details here but I advise you to instantiate your DAGs like that. It's clearer and better than creating a variable and put your DAG into.

### Step 3: Add your tasks!

Once you have made the imports and created your DAG object, you are ready to add your tasks! Remember, a task is an operator. Therefore, based on your DAG, you have to add 6 operators. Let's dive into the tasks.

### Training model tasks

First, training model A, B and C, are implemented with the *PythonOperator*. Since we are not going to train real machine learning models (too complicated to start), each task will return a random accuracy. This accuracy will be generated from a python function named *\_training\_model*.

```
1  from random import randint # Import to generate random numbers
2  def _training_model():
3      return randint(1, 10) # return an integer between 1 - 10
```

```
10 training_model_B = PythonOperator(  
11     task_id="training_model_B",  
12     python_callable=_training_model  
13 )  
14 training_model_C = PythonOperator(  
15     task_id="training_model_C",  
16     python_callable=_training_model  
17 )
```

That's great but you can do better. Indeed, the 3 tasks are really similar. The only difference lies into the task ids. Therefore, since DAGs are coded in Python, we can benefit from that and generate the tasks dynamically. Take a look at the code below

```
1     training_model_tasks = [  
2         PythonOperator(  
3             task_id=f"training_model_{model_id}",  
4             python_callable=_training_model,  
5             op_kwargs={  
6                 "model": model_id  
7             }  
8         ) for model_id in ['A', 'B', 'C']  
9     ]
```

By defining a list comprehension, we are able to generate the 3 tasks dynamically which is.... much cleaner. Less code, the better 😎

If you are wondering how the PythonOperator works, take a look at my article [here](#), you will learn everything you need about it.

## Choosing best model

The next task is "Choosing Best ML". Since this task executes either the task "accurate" or "inaccurate" based on the best accuracy, the

```

2  accuracies = ti.xcom_pull(task_ids=[
3  'training_model_A',
4  'training_model_B',
5  'training_model_C'
6  ])
7  if max(accuracies) > 8:
8  return 'accurate'
9  return 'inaccurate'
10 with DAG(...) as dag:
11     choosing_best_model = BranchPythonOperator(
12         task_id="choosing_best_model",
13         python_callable=_choosing_best_model
14     )

```

Ok, it looks a little bit more complicated here. Let's start by the beginning. First, the BranchPythonOperator executes a python function. Here, `_choosing_best_model`. This function must return the task id of the next task to execute. For your DAG, either "accurate" or "inaccurate" as shown from the return keywords. Now, there is something we didn't talk about yet. What is `xcom_pull`? 🤖

Whenever you want to share data between tasks in Airflow, you have to use XCOMs. XCOM stands for cross-communication messages, it is a mechanism allowing to exchange small data between the tasks of a DAG. A XCOM is an object encapsulating a key, serving as an identifier, and a value, corresponding to the value you want to share. I won't go into the details here as I made a long article about it, just keep in mind that by returning the accuracy from the python function `_training_model_X`, we create a XCOM with that accuracy, and with `xcom_pull` in `_choosing_best_model`, we fetch that XCOM back corresponding to the accuracy. As we want the accuracy of each `training_model` task, we specify the task ids of these 3 tasks. That's all you need to know 😊

## Accurate or inaccurate?

The last two tasks to implements are "accurate" and "inaccurate". To do that, you can use the BashOperator and execute a very simple bash command to either print "accurate" or "inaccurate" on the standard output.

```

1  accurate = BashOperator(
2  task_id="accurate",
3  bash_command="echo 'accurate'"
4  )
5  inaccurate = BashOperator(
6  task_id="inaccurate",
7  bash_command=" echo 'inaccurate'"
8  )

```

That's it, nothing more to add. The BashOperator is used to execute bash commands and that's exactly what you're doing here.

All right, that was a lot, time to move to the last step!



In your case, it's really basic as you want to execute one task after the other.

```
1 with DAG(...) as dag:
2     training_model_tasks >> choosing_best_model >> [accurate, inaccurate]
```

Here you say that *training\_model\_tasks* are executed first, then once all of the tasks are completed, *choosing\_best\_model* gets executed, and finally, either *accurate* or *inaccurate*. Keep in mind that each time you have multiple tasks that should be on the same level, in a same group, that can be executed at the same time, use a list with [ ].

### The Final Airflow DAG!

Ok, now you've gone through all the steps, time to see the final code:

```
1 from airflow import DAG
2 from airflow.operators.python import PythonOperator, BranchPythonOperator
3 from airflow.operators.bash import BashOperator
4 from datetime import datetime
5 from random import randint
6 def _choosing_best_model(ti):
7     accuracies = ti.xcom_pull(task_ids=[
8         'training_model_A',
9         'training_model_B',
10        'training_model_C'
11    ])
12    if max(accuracies) > 8:
13        return 'accurate'
14    return 'inaccurate'
15 def _training_model(model):
16     return randint(1, 10)
17 with DAG("my_dag",
18     start_date=datetime(2021, 1 ,1),
19     schedule_interval='@daily',
20     catchup=False) as dag:
```

```
26     "model": model_id
27 }
28 ) for model_id in ['A', 'B', 'C']
29 ]
30 choosing_best_model = BranchPythonOperator(
31     task_id="choosing_best_model",
32     python_callable=_choosing_best_model
33 )
34 accurate = BashOperator(
35     task_id="accurate",
36     bash_command="echo 'accurate'"
37 )
38 inaccurate = BashOperator(
39     task_id="inaccurate",
40     bash_command=" echo 'inaccurate'"
41 )
42 training_model_tasks >> choosing_best_model >> [accurate, inaccurate]
```

That’s it you’ve just created your first Airflow DAG! If you want to test it, put that code into a file my\_dag.py and put that file into the folder dags/ of Airflow. Once you’ve done that, run it from the UI and you should obtain the following output:

Graph View of my\_dag



## Conclusion

That's it about creating your first Airflow DAG. It wasn't too difficult isn't it? You've learned how to create a DAG, generate tasks dynamically, choose one task or another with the BranchPythonOperator, share data between tasks and define dependencies with bitshift operators. If you want to learn more about Apache Airflow, check my course [here](#), have a wonderful day and see you for another tutorial! 🙌

← Previous Post

Next Post →



**The Actual Cost Of One Day Full Mouth Dental Implants In Turkey Might Surprise You**  
Dental Implants | Search Ads



**Unsold Laptops Are Being Sold for Almost Nothing**  
Laptops | Search Ads



**Do You Need Dental Implants? See How Much Full Mouth Implants Cost**  
Dental Implants | Search Ads



**Don't Play This Game If You Are Under 40 Years Old**  
Raid: Shadow Legends



**Forget Expensive Roofing, 2021 Invention Changes Industry**  
Roofing | Sponsored Listings



**This Is How Much A New Roof Should Cost In 2022 (See Prices)**  
Roofing services | Search Ads

Recommended by |

2 thoughts on “Airflow DAG: Creating your first DAG in 5 minutes”

DONNA  
2021-05-13 AT 19:48

Thank you for sharing this information. I am learning the XCom concept. Do you not need to push the values into the XCom in order to later pull it in \_choosing\_best\_model?

Reply

JOHN  
2021-05-01 AT 07:00



Reply

## Leave a Comment

Your email address will not be published. Required fields are marked \*

Type here..

Name\*

Email\*

Website

Post Comment »



report this ad

Search ...



- Dynamic Task Mapping in Apache Airflow
- DAG Dependencies in Apache Airflow: The Ultimate Guide
- Dynamic DAGs in Apache Airflow: The Ultimate Guide
- Airflow TaskGroups: All you need to know!

Categories

- Airflow 2.0
- Apache Airflow
- api
- dag
- kubernetes
- Non classé
- sensors
- xcoms



report this ad



report this ad



[report this ad](#)

























[report this ad](#)