Introduction to Airflow in Python

A beginner’s guide to the basic concepts of Apache Airflow

[Black Raven (James Ng)](https://medium.com/@jnyh?source=post_page-----67b554f06f0b----------------------)

[Black Raven (James Ng)](https://medium.com/@jnyh?source=post_page-----67b554f06f0b----------------------)

[Aug 9](https://medium.com/swlh/introduction-to-airflow-in-python-67b554f06f0b?source=post_page-----67b554f06f0b----------------------) · 25 min read

This is a memo to share what I have learnt in Apache Airflow, capturing the learning objectives as well as my personal notes. The course is taught by Mike Metzger from DataCamp, and it includes [4 chapters](https://github.com/JNYH/DataCamp_Introduction_to_Airflow):

Chapter 1. Intro to Airflow  
Chapter 2. Implementing Airflow DAGs  
Chapter 3. Maintaining and monitoring Airflow workflows  
Chapter 4. Building production pipelines in Airflow



Photo by [Jacek Dylag](https://unsplash.com/@dylu?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

A data engineer’s job includes writing scripts, adding complex CRON tasks, and trying various ways to meet an ever-changing set of requirements to deliver data on schedule. Airflow can do all these while adding scheduling, error handling, and reporting. This course will guide you in the basic concepts of Airflow and help you implement data engineering workflows in production. You’ll implement many different data engineering tasks in a predictable and repeatable fashion.

**Chapter 1. Intro to Airflow**

**Data engineering** is taking any action involving data and turning it into a reliable, repeatable, and maintainable process.

**Workflow**is a set of steps to accomplish a given data engineering task, such as downloading files, copying data, filtering information, writing to a database, etc.

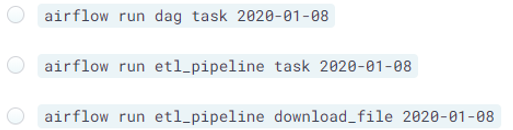
**Airflow**is a platform to  
· program workflows including: creation, scheduling, and monitoring  
· implement workflow as DAGs (Directed Acyclic Graphs)  
· be accessed via code, command-line (CLI), or web user interface (UI)

**Running a task in Airflow**

You’ve just started looking at using Airflow within your company and would like to try to run a task within the Airflow platform. You remember that you can use the airflow run command to execute a specific task within a workflow.

Note that an error while using airflow run will return airflow.exceptions.AirflowException: on the last line of output.

An Airflow DAG is set up for you with a dag\_id of etl\_pipeline. The task\_id is download\_file and the start\_date is 2020-01-08. Which command would you enter in the console to run the desired task?

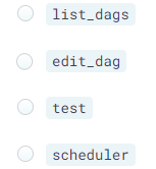


Answer: airflow run etl\_pipeline download\_file 2020–01–08  
Syntax: airflow run <dag\_id> <task\_id> <start\_date>

**Examining Airflow commands**

While researching how to use Airflow, you start to wonder about the airflow command in general. You realize that by simply running airflow you can get further information about various sub-commands that are available.

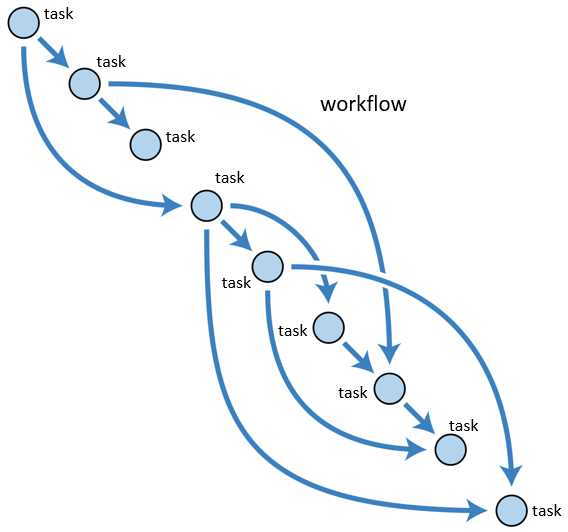
Which of the following is *NOT* an Airflow sub-command?



Answer: edit\_dag  
You can use the airflow -h command to obtain further information about any Airflow command.

**Airflow DAGs**

DAG (Directed Acyclic Graph) is  
· *Directed*, an inherent flow, dependencies between components  
· *Acyclic*, does not loop/cycle/repeat  
· *Graph*, the actual set of components



Directed Acyclic Graph (DAG), image by the author

**Defining a simple DAG**

You’ve spent some time reviewing the Airflow components and are interested in testing out your own workflows. To start you decide to define the default arguments and create a DAG object for your workflow.

# Import the DAG object  
from airflow.models import DAG

# Define the default\_args dictionary  
default\_args = {  
 'owner': 'dsmith',  
 'start\_date': datetime(2020, 1, 14),  
 'retries': 2  
}

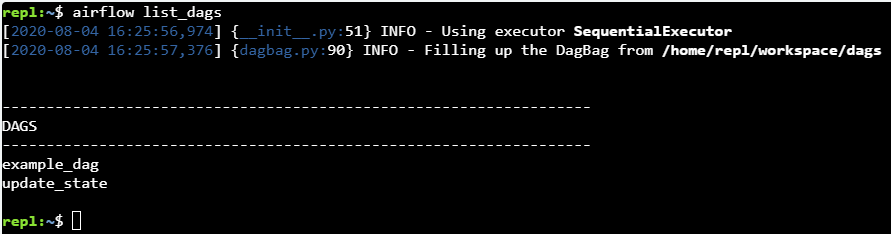
# Instantiate the DAG object  
etl\_dag = DAG('example\_etl', default\_args=default\_args)

Syntax: dag\_variable = DAG('dag\_name', default\_args=default\_args)

**Working with DAGs and the Airflow shell**

While working with Airflow, sometimes it can be tricky to remember what DAGs are defined and what they do. You want to gain some further knowledge of the Airflow shell command so you’d like to see what options are available.

Multiple DAGs are already defined for you. How many DAGs are present in the Airflow system from the command-line?



Answer: there are 2 DAGs

**Troubleshooting DAG creation**

Now that you’ve successfully worked with a couple workflows, you notice that sometimes there are issues making a workflow appear within Airflow. You’d like to be able to better troubleshoot the behavior of Airflow when there may be something wrong with the code.

Two DAGs are defined for you and Airflow is setup. Note that any changes you make within the editor are automatically saved.

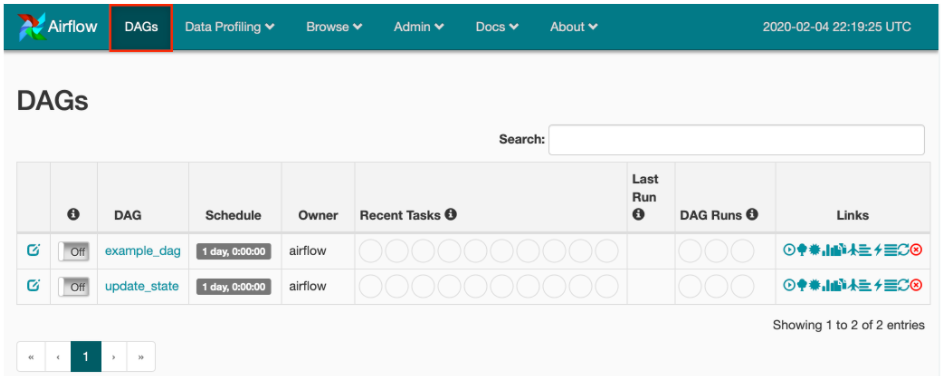
from airflow.models import DAG  
default\_args = {   
 'owner': 'jdoe',   
 'start\_date': '2019–01–01'   
}  
dag = DAG( dag\_id='etl\_update', default\_args=default\_args )

refresh\_data\_workflow.py

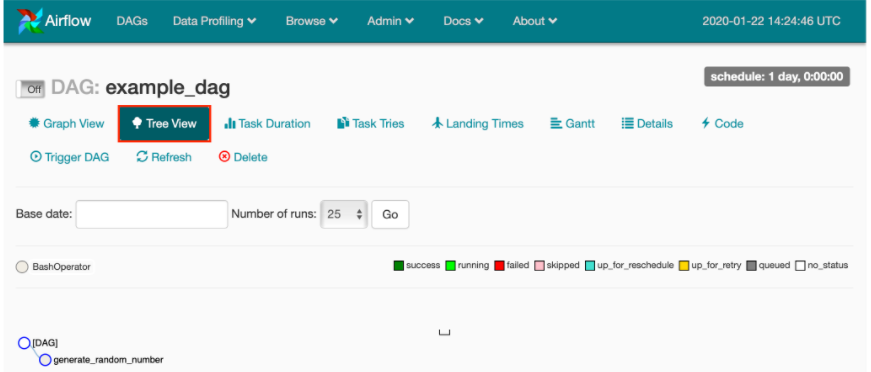
from airflow.models import DAG  
default\_args = {   
 'owner': 'jdoe',  
 'email': 'jdoe@datacamp.com',   
 'start\_date': '2019–01–01'   
}  
dag = DAG( dag\_id='refresh\_data', default\_args=default\_args )

**Airflow web interface**

To view DAGs



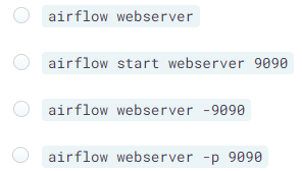
Click on example\_dag



**Starting the Airflow webserver**

You’ve successfully created some DAGs within Airflow using the command-line tools, but notice that it can be a bit tricky to handle scheduling / troubleshooting / etc. After reading the documentation further, you realize that you’d like to access the Airflow web interface. For security reasons, you’d like to start the webserver on port 9090.

Which airflow command would you use to start the webserver on port 9090?

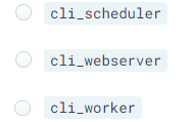


Answer: airflow webserver -p 9090  
Sometimes the defaults for Airflow aren’t exactly what you’d like to use. Using the built in tools to configure the setup to your specifications is a very common function of a data engineer.

**Navigating the Airflow UI**

To gain some familiarity with the Airflow UI, you decide to explore the various pages. You’d like to know what has happened on your Airflow instance thus far.

Which of the following events have not run on your Airflow instance?



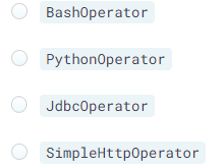
Answer: cli\_worker

**Examining DAGs with the Airflow UI**

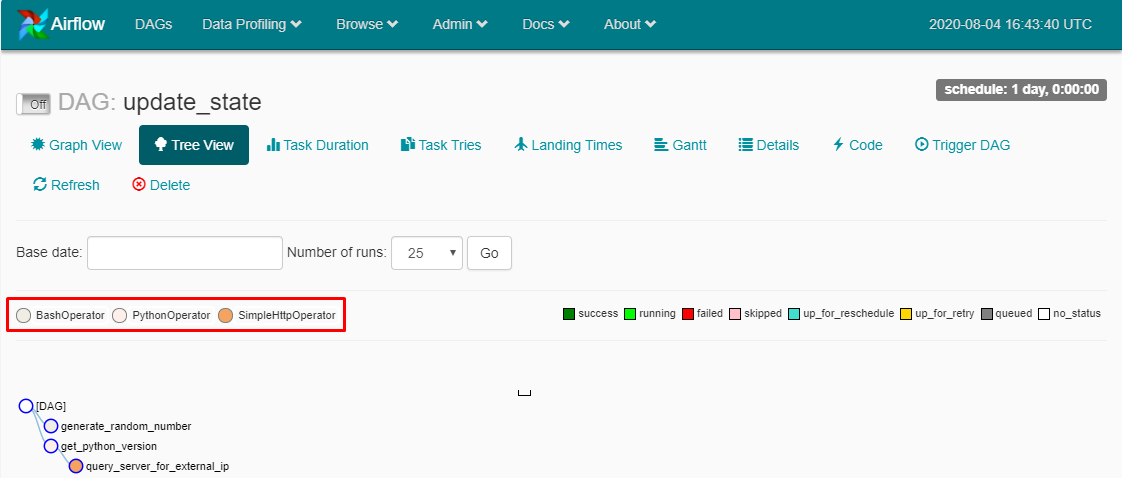
You’ve become familiar with the basics of an Airflow DAG and the basics of interacting with Airflow on the command-line. Your boss would like you to show others on your team how to examine any available DAGs. In this instance, she would like to know which operator is **NOT** in use with the DAG called update\_state, as your team is trying to verify the components used in production workflows.

Remember that the Airflow UI allows various methods to view the state of DAGs. The Tree View lists the tasks and any ordering between them in a tree structure, with the ability to compress / expand the nodes. The Graph View shows any tasks and their dependencies in a graph structure, along with the ability to access further details about task runs. The Code view provides full access to the Python code that makes up the DAG.

Remember to select the operator **NOT** used in this DAG.



Answer: JdbcOperator



**Chapter 2. Implementing Airflow DAGs**

Learn the basics of implementing Airflow DAGs using operators, tasks, and scheduling.

**Airflow operators**

Operators represent a single task in a workflow, running independently on different tasks, and generally do not share information. E.g. DummyOperator, BashOperator.



**Defining a BashOperator task**

The BashOperator allows you to specify any given Shell command or script and add it to an Airflow workflow.

As such, you’ve been running some scripts manually to clean data (using a script called cleanup.sh) prior to delivery to your colleagues in the Data Analytics group. As you get more of these tasks assigned, you've realized it's becoming difficult to keep up with running everything manually, much less dealing with errors or retries. You'd like to implement a simple script as an Airflow operator.

The Airflow DAG analytics\_dag is already defined for you and has the appropriate configurations in place.

# Import the BashOperator  
from airflow.operators.bash\_operator import BashOperator

# Define the BashOperator   
cleanup = BashOperator(  
 task\_id='cleanup\_task',  
 # Define the bash\_command  
 bash\_command='cleanup.sh',  
 # Add the task to the dag  
 dag=analytics\_dag  
)

**Multiple BashOperators**

Airflow DAGs can contain many operators, each performing their defined tasks.

You’ve successfully implemented one of your scripts as an Airflow task and have decided to continue migrating your individual scripts to a full Airflow DAG. You now want to add more components to the workflow. In addition to the cleanup.sh used in the previous exercise you have two more scripts, consolidate\_data.sh and push\_data.sh. These further process your data and copy to its final location.

The DAG analytics\_dag is available as before, and your cleanup task is still defined. The BashOperator is already imported.

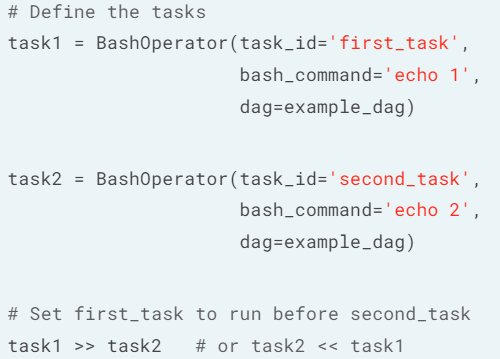
# Define a second operator to run the `consolidate\_data.sh` script  
consolidate = BashOperator(  
 task\_id='consolidate\_task',  
 bash\_command='consolidate\_data.sh',  
 dag=analytics\_dag)

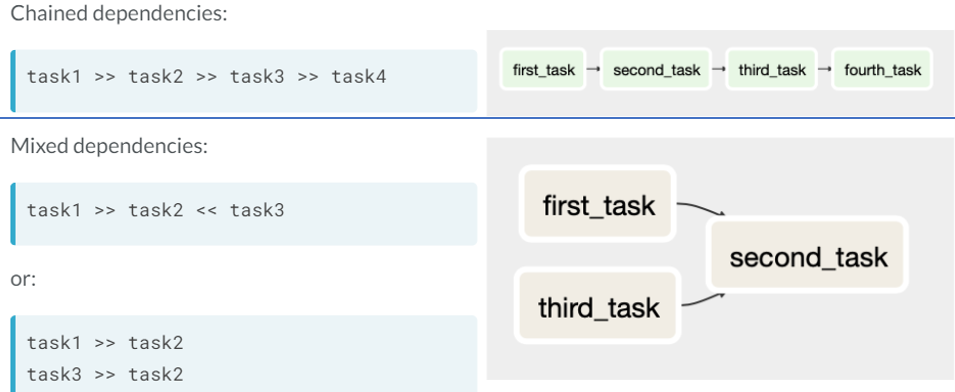
# Define a final operator to execute the `push\_data.sh` script  
push\_data = BashOperator(  
 task\_id='pushdata\_task',  
 bash\_command='push\_data.sh',  
 dag=analytics\_dag)

**Airflow tasks**

Tasks are instances of operators, usually assigned to a variable in Python, and referred to by the **task\_id**(not variable name) within the Airflow tools.

**Tasks dependencies** are referred to as upstream or downstream tasks. Upstream tasks need to be completed before downstream ones, defined using bitshift operators (>>) between 2 **task variables**.





**Define order of BashOperators**

Now that you’ve learned about the bitshift operators, it’s time to modify your workflow to include a pull step and to include the task ordering. You have three currently defined components, cleanup, consolidate, and push\_data.

The DAG analytics\_dag is available as before and the BashOperator is already imported.

# Define a new pull\_sales task  
pull\_sales = BashOperator(  
 task\_id='pullsales\_task',  
 bash\_command='wget [https://salestracking/latestinfo?json'](https://salestracking/latestinfo?json%27),  
 dag=analytics\_dag  
)

# Set pull\_sales to run prior to cleanup  
pull\_sales >> cleanup

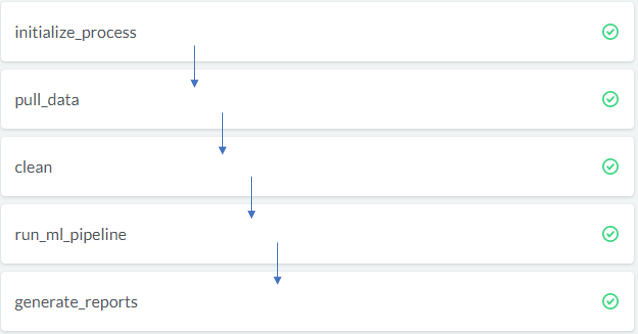
# Configure consolidate to run after cleanup  
cleanup >> consolidate

# Set push\_data to run last  
consolidate >> push\_data

**Determining the order of tasks**

While looking through a colleague’s workflow definition, you’re trying to decipher exactly in which order the defined tasks run. The code in question shows the following:

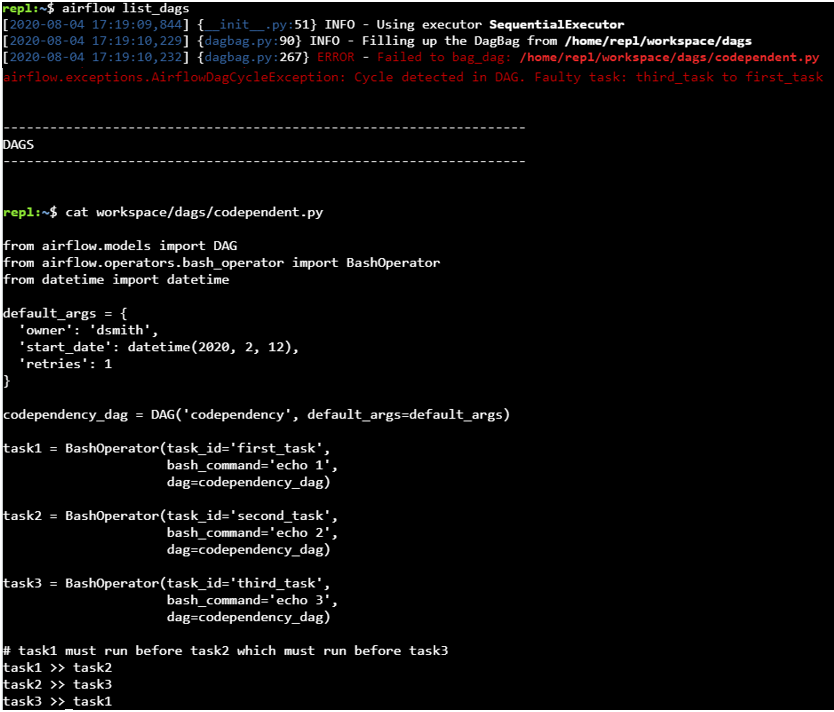
pull\_data << initialize\_process  
pull\_data >> clean >> run\_ml\_pipeline  
generate\_reports << run\_ml\_pipeline

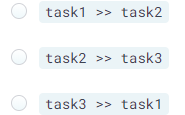


**Troubleshooting DAG dependencies**

You’ve created a DAG with intended dependencies based on your workflow but for some reason Airflow won’t load / execute the DAG. Try using the terminal to:

* List the DAGs.
* Decipher the error message.
* Use cat workspace/dags/codependent.py to view the Python code.
* Determine which of the following lines should be removed from the Python code. You may want to consider the last line of the file.

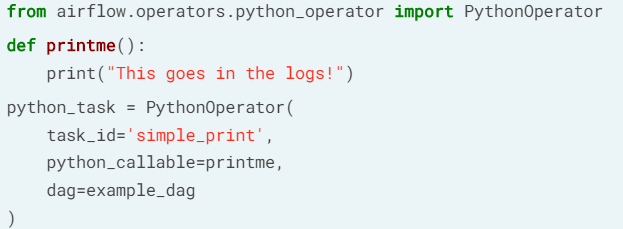




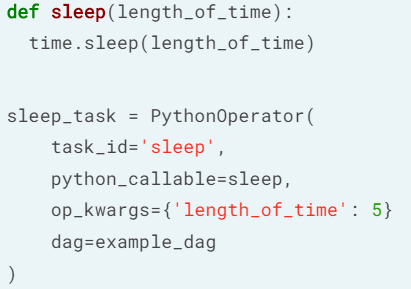
Answer: task3 >> task1  
For this particular issue, a loop, or cycle, is present within the DAG. Note that technically removing the first dependency would resolve the issue as well, but the comments specifically reference the desired effect. Commenting the desired effect in this way can often help resolve bugs in Airflow DAG execution.

**Additional operators**

**PythonOperator**executes a Python function (Example 1) or callable function with keyword arguments op\_kwargs dictionary (Example 2).

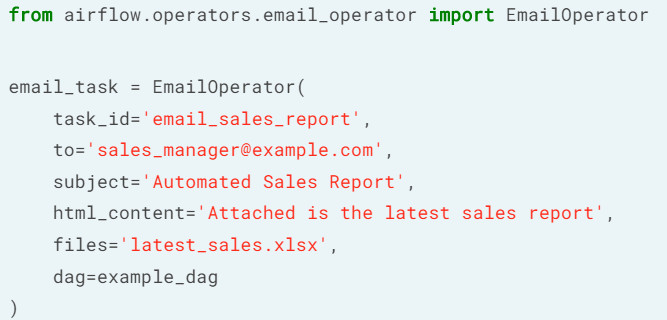


Example 1



Example 2

**EmailOperator**sends an email, with html content and attachments, after configured with email server details.



**Using the PythonOperator**

You’ve implemented several Airflow tasks using the BashOperator but realize that a couple of specific tasks would be better implemented using Python. You’ll implement a task to download and save a file to the system within Airflow.

The requests library is imported for you, and the DAG process\_sales\_dag is already defined.

# Define the method  
def pull\_file(URL, savepath):  
 r = requests.get(URL)  
 with open(savepath, 'wb') as f:  
 f.write(r.content)   
 # Use the print method for logging  
 print(f'File pulled from {URL} and saved to {savepath}')

# Import the PythonOperator class  
from airflow.operators.python\_operator import PythonOperator

# Create the task  
pull\_file\_task = PythonOperator(  
 task\_id='pull\_file',  
 # Add the callable  
 python\_callable=pull\_file,  
 # Define the arguments  
 op\_kwargs={'URL':'<http://dataserver/sales.json'>, 'savepath':'latestsales.json'},  
 dag=process\_sales\_dag

You can use .format() or other variable substitution methods as desired, especially if working with a Python version earlier than 3.6.

**More PythonOperators**

To continue implementing your workflow, you need to add another step to parse and save the changes of the downloaded file. The DAG process\_sales\_dag is defined and has the pull\_file task already added. In this case, the Python function is already defined for you, parse\_file(inputfile, outputfile).

Note that often when implementing Airflow tasks, you won’t necessarily understand the individual steps given to you. As long as you understand how to wrap the steps within Airflow’s structure, you’ll be able to implement a desired workflow.

# Add another Python task  
parse\_file\_task = PythonOperator(  
 task\_id='parse\_file',  
 # Set the function to call  
 python\_callable=parse\_file,  
 # Add the arguments  
 op\_kwargs={'inputfile':'latestsales.json', 'outputfile':'parsedfile.json'},  
 # Add the DAG  
 dag=process\_sales\_dag  
)

**EmailOperator and dependencies**

Now that you’ve successfully defined the PythonOperators for your workflow, your manager would like to receive a copy of the parsed JSON file via email when the workflow completes. The previous tasks are still defined and the DAG process\_sales\_dag is configured.

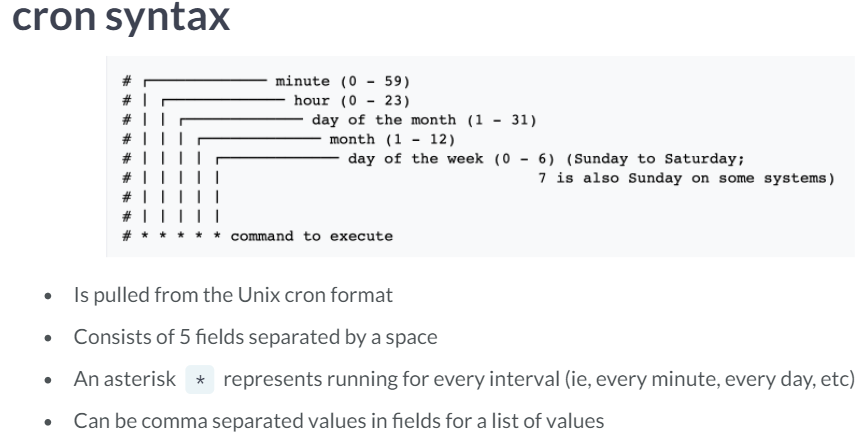
# Import the Operator  
from airflow.operators.email\_operator import EmailOperator

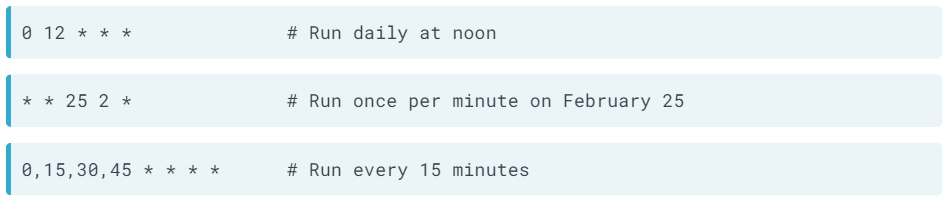
# Define the task  
email\_manager\_task = EmailOperator(  
 task\_id='email\_manager',  
 to='manager@datacamp.com',  
 subject='Latest sales JSON',  
 html\_content='Attached is the latest sales JSON file as requested.',  
 files='parsedfile.json',  
 dag=process\_sales\_dag  
)

# Set the order of tasks  
pull\_file\_task >> parse\_file\_task >> email\_manager\_task

**Airflow scheduling**

Browse -> DAG Runs  
· are specific instance of a workflow at a point in time  
· can be run manually or via schedule\_interval  
· each workflow states: running, failed, success





**Schedule a DAG via Python**

You’ve learned quite a bit about creating DAGs, but now you would like to schedule a specific DAG on a specific day of the week at a certain time. You’d like the code include this information in case a colleague needs to reinstall the DAG to a different server.

The Airflow DAG object and the appropriate datetime methods have been imported for you.

# Update the scheduling arguments as defined  
default\_args = {  
 'owner': 'Engineering',  
 'start\_date': datetime(2019, 11, 1),  
 'email': ['airflowresults@datacamp.com'],  
 'email\_on\_failure': False,  
 'email\_on\_retry': False,  
 'retries': 3,  
 'retry\_delay': timedelta(minutes=20)  
}

# Use the cron syntax for every Wednesday at 12:30pm  
dag = DAG('update\_dataflows', default\_args=default\_args, schedule\_interval='30 12 \* \* 3')

**Deciphering Airflow schedules**

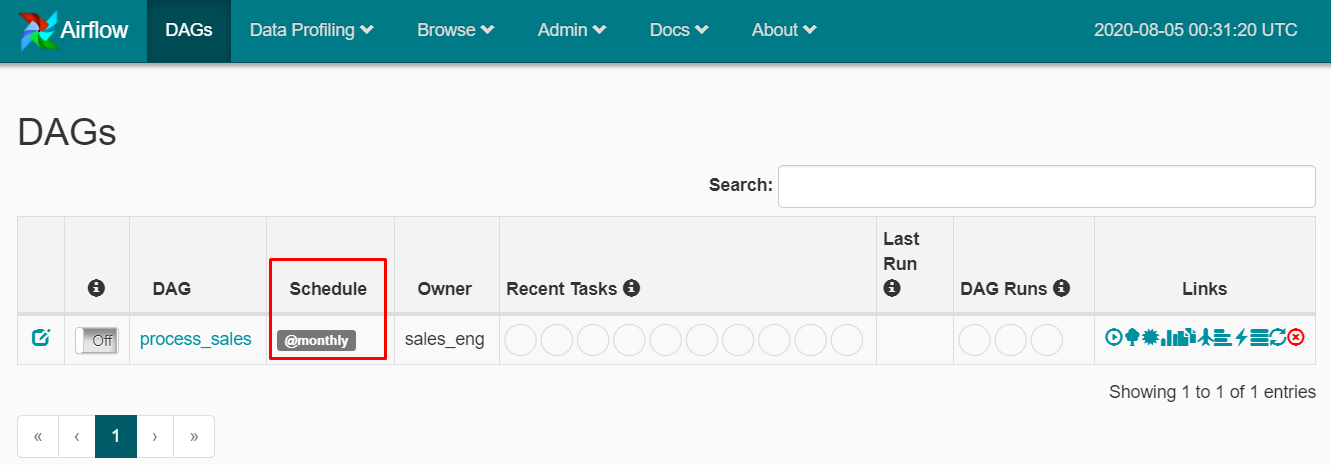
Given the various options for Airflow’s schedule\_interval, you'd like to verify that you understand exactly how intervals relate to each other, whether it's a cron format, timedelta object, or a preset.

Order the schedule intervals from least to greatest amount of time.

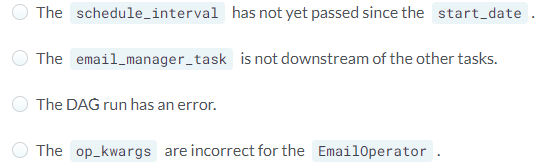


**Troubleshooting DAG runs**

You’ve scheduled a DAG called process\_sales which is set to run on the first day of the month and email your manager a copy of the report generated in the workflow. The start\_date for the DAG is set to February 15, 2020. Unfortunately it's now March 2nd and your manager did not receive the report and would like to know what happened.



Use the information you’ve learned about Airflow scheduling to determine what the issue is.



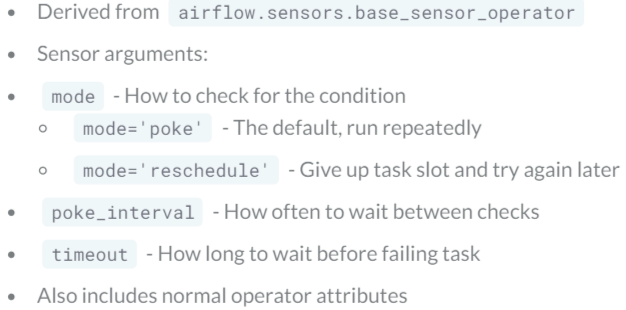
Answer: The schedule\_interval has not yet passed since the start\_date.

**Chapter 3. Maintaining and monitoring Airflow workflows**

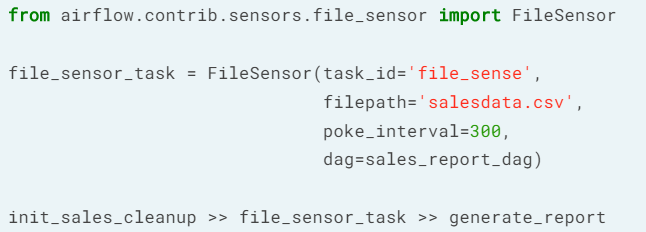
Learn more about Airflow components such as sensors and executors while monitoring and troubleshooting Airflow workflows.

**Airflow sensors**

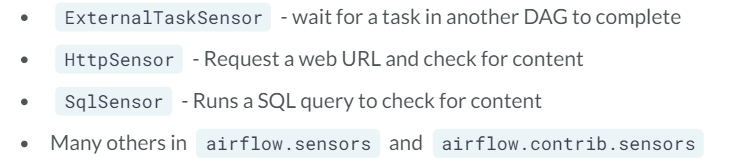
Sensors are operators that wait for a certain condition to be true, e.g. creation of a file, database record upload, response from a web. Sensors are assigned to tasks, and the frequency to check for the condition to be true can be defined.



**File sensor** checks for the existence of a file at a certain location (directory).



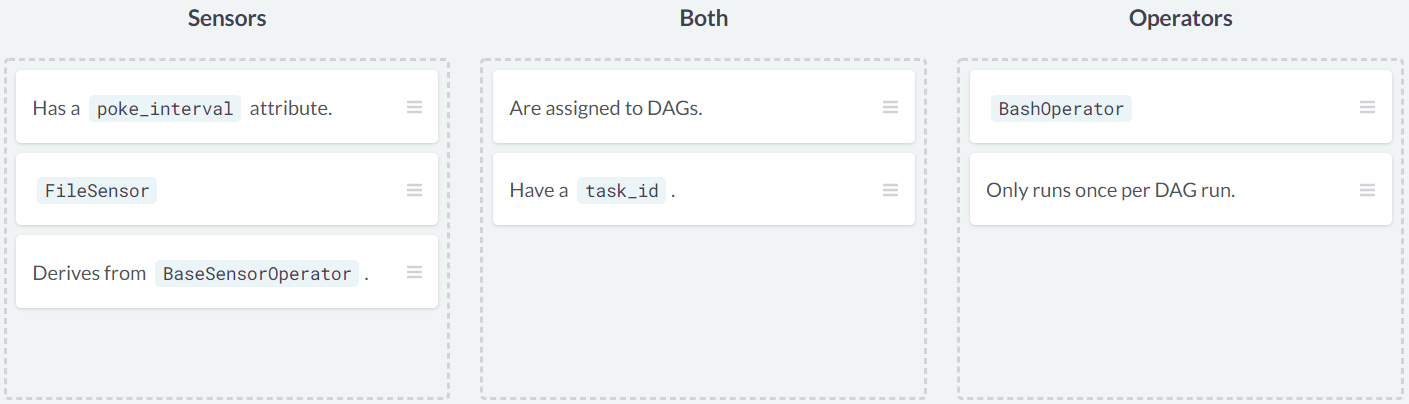
poke\_interval = 300 seconds



Other sensors available in Airflow

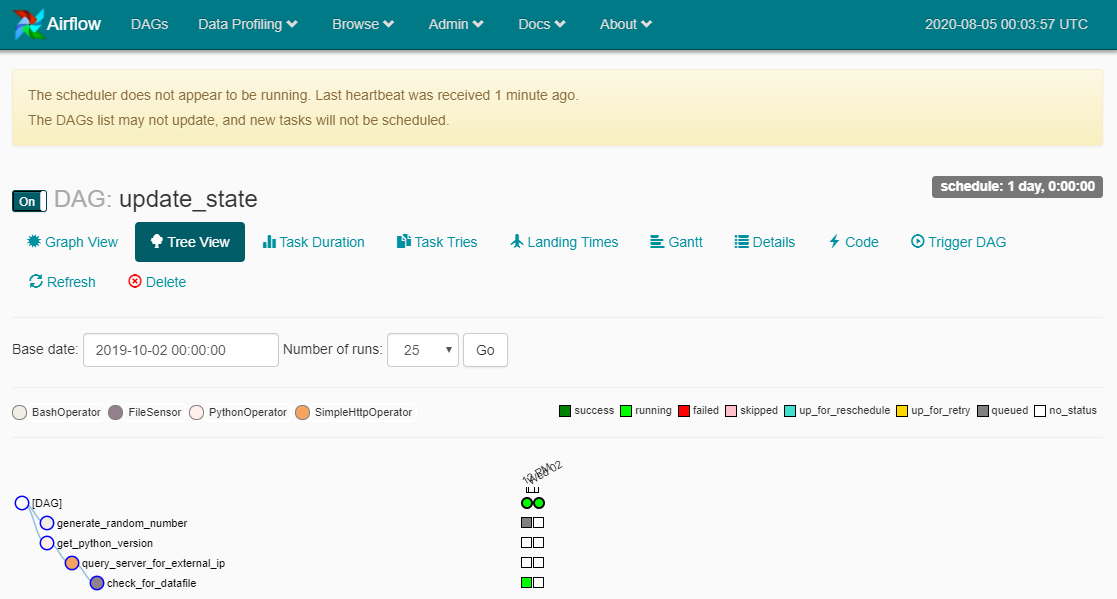
**Sensors vs operators**

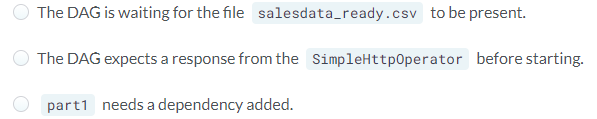
As you’ve just learned about sensors, you want to verify you understand what they have in common with normal operators and where they differ.



**Sensory deprivation**

You’ve recently taken over for another Airflow developer and are trying to learn about the various workflows defined within the system. You come across a DAG that you can’t seem to make run properly using any of the normal tools. Try exploring the DAG for any information about what it might be looking for before continuing.





Answer: The DAG is waiting for the file salesdata\_ready.csv to be present.

**Airflow executors**

An executor is the component that runs the task in a workflow, for example, SequentialExecutor, LocalExecutor, CeleryExecutor.

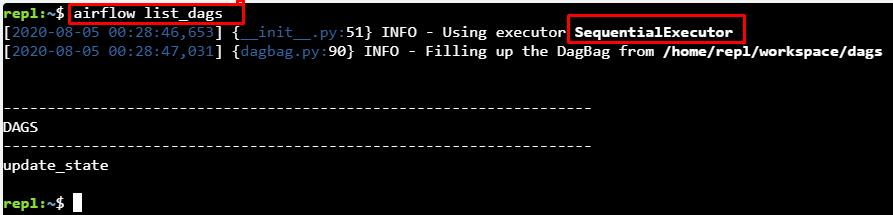
Sequential Executor is the default Airflow execution engine that runs one task at a time. It is useful for debugging, but not recommended for production due to the limitation of task resources.

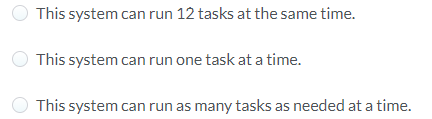
LocalExecutor runs on a single system, treats each task as a process, and is able to start as many concurrent tasks as permitted by the system resources (ie, CPU cores, memory, etc). It is a good choice for a single production Airflow system and can utilise all the resources of a given host system.

CeleryExecutor uses a Celery backend as task manager. It is a general queuing system written in Python that allows multiple systems (parallelism) to communicate as a basic cluster. Using a CeleryExecutor, multiple Airflow systems can be configured as workers for a given set of workflows/tasks. You can add extra systems at any time to better balance workflows, but it is more difficult to set up and configure.

**Determining the executor**

While developing your DAGs in Airflow, you realize you’re not certain the configuration of the system. Using the commands you’ve learned, determine which of the following statements is true.



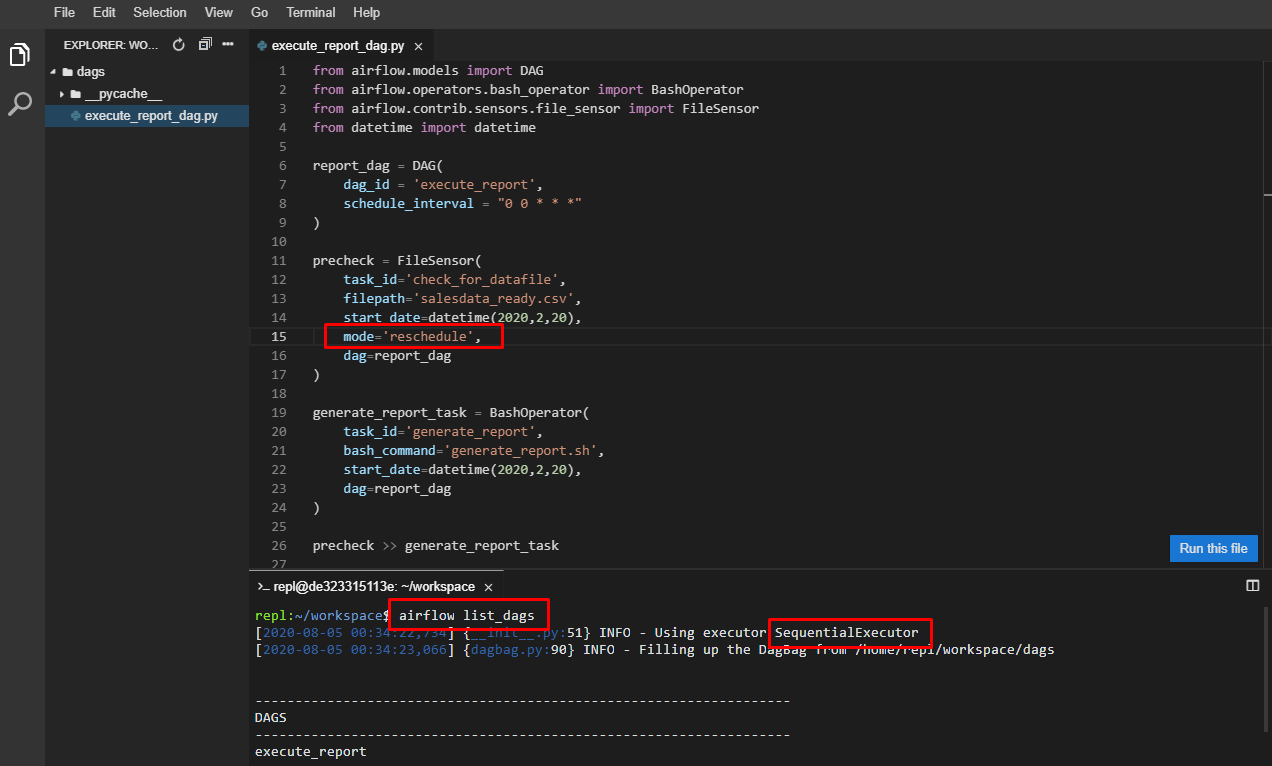


Answer: This system can run one task at a time.

**Executor implications**

You’re learning quite a bit about running Airflow DAGs and are gaining some confidence at developing new workflows. That said, your manager has mentioned that on some days, the workflows are taking a lot longer to finish and asks you to investigate. She also mentions that the salesdata\_ready.csv file is taking longer to generate these days and the time of day it is completed is variable.

This exercise requires information from the previous two lessons — remember the implications of the available arguments and modify the workflow accordingly.



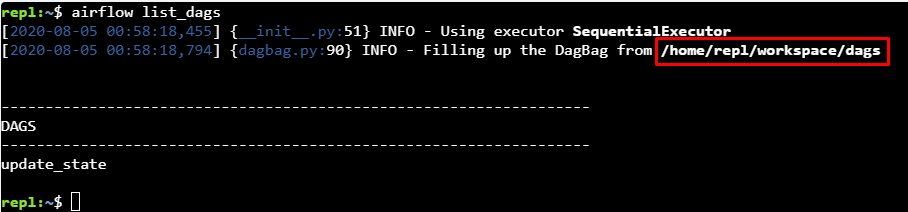
By modifying the sensor properties (from mode=’poke’ to mode=’reschedule’), Airflow is given a chance **to run another task while waiting** for the salesdata\_ready.csv file. This required recognizing the connection between an executor and the number and type of tasks in a workflow. Alternatively, you could also modify the executor type to something with a parallelism greater than 1 to allow the tasks to complete.

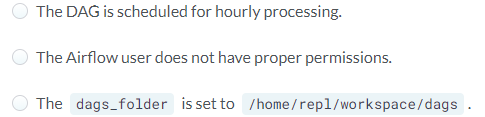
**Debugging and troubleshooting in Airflow**

Typical issues (refer to [video here](https://github.com/JNYH/Introduction_to_Airflow)):  
· DAG won’t run on schedule — scheduler is not running (fix this issue by running airflow scheduler from the command-line), not enough free slots for executor to run tasks (change to parallel executor, add system resources, or change DAG schedule to lower peak period)  
· DAG won’t load — DAG not in web UI, DAG not shown in airflow list\_dags (verify DAG file is in correct folder, determine the DAGs folder via airflow.cfg)  
· Syntax errors — code failed to compile (debug in VSCode IDE or Jupyter notebook)

**DAGs in the bag**

You’ve taken over managing an Airflow cluster that you did not setup and are trying to learn a bit more about the system configuration. Which of the following is true?



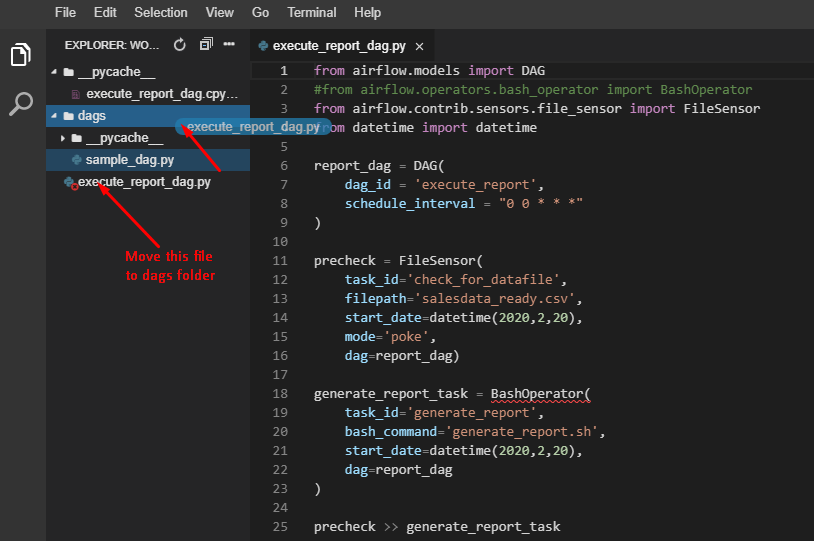


Answer: The dags\_folder is set to /home/repl/workspace/dags.

**Missing DAG**

Your manager calls you before you’re about to leave for the evening and wants to know why a new DAG workflow she’s created isn’t showing up in the system. She needs this DAG called execute\_report to appear in the system so she can properly schedule it for some tests before she leaves on a trip.

Airflow is configured using the ~/airflow/airflow.cfg file. This is a multi-layered issue for why the DAG would not load.

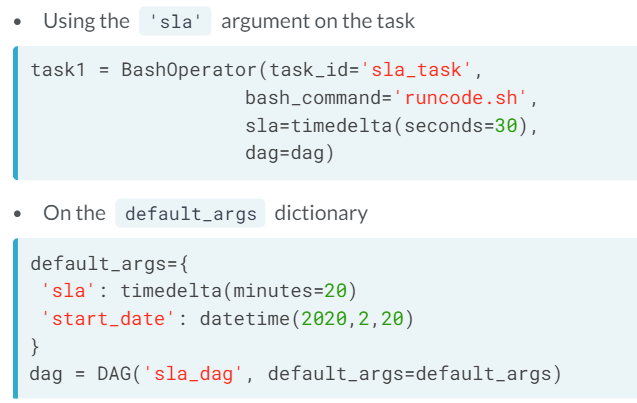


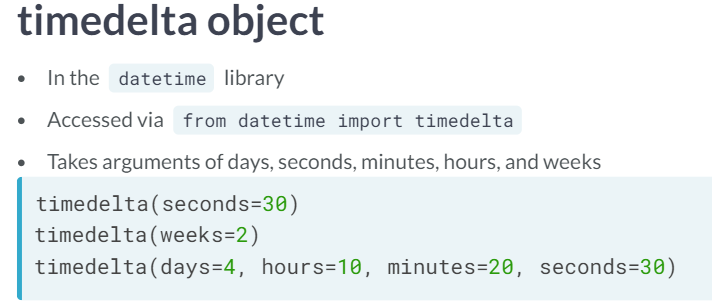
Remember that sometimes having no apparent error does not necessarily mean everything is working as expected. It is common for there to be more than one simultaneous problem with loading workflows, even if the issues appear simple at first. Always try to consider the problems that could appear, and that there might be more than one as it will simplify your usage of Airflow.

**SLAs and reporting in Airflow**

Service Level Agreement (SLA) is the amount of time a task or a DAG should require to run. If the task/DAG does not meet the expected timing, it is called SLA Miss, with logs stored in the web UI (Browse → SLA Misses).

2 ways to define SLAs:





**Defining an SLA**

You’ve successfully implemented several Airflow workflows into production, but you don’t currently have any method of determining if a workflow takes too long to run. After consulting with your manager and your team, you decide to implement an SLA at the DAG level on a test workflow.

# Import the timedelta object  
from datetime import timedelta

# Create the dictionary entry  
default\_args = {  
 'start\_date': datetime(2020, 2, 20),  
 'sla': timedelta(minutes=30)  
}

# Add to the DAG  
test\_dag = DAG('test\_workflow', default\_args=default\_args, schedule\_interval='@None')

Note that this type of SLA applies for the entire workflow, not just an individual task.

**Defining a task SLA**

After completing the SLA on the entire workflow, you realize you really only need the SLA timing on a specific task instead of the full workflow.

# Import the timedelta object  
from datetime import timedeltatest\_dag = DAG('test\_workflow', start\_date=datetime(2020,2,20), schedule\_interval='@None')

# Create the task with the SLA  
task1 = BashOperator(task\_id='first\_task',  
 sla=timedelta(hours=3),  
 bash\_command='initialize\_data.sh',  
 dag=test\_dag)

You can add specific SLAs to individual tasks as needed. Try adding various SLA settings to your workflows to determine how your systems are behaving overall.

**Generate and email a report**

Airflow provides the ability to automate almost any style of workflow. You would like to receive a report from Airflow when tasks complete without requiring constant monitoring of the UI or log files. You decide to use the email functionality within Airflow to provide this message.

All the typical Airflow components have been imported for you, and a DAG is already defined as dag.

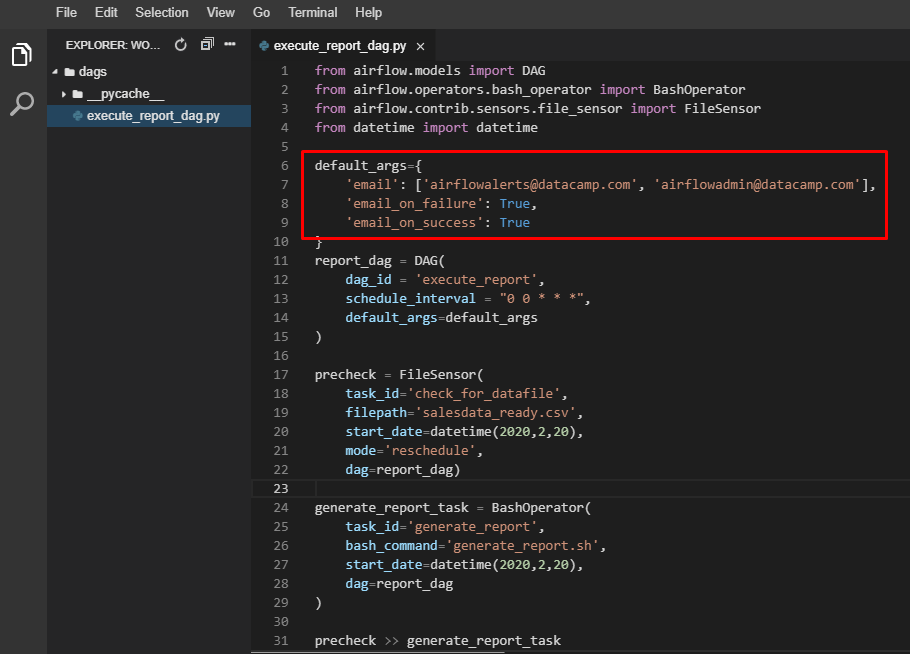
# Define the email task  
email\_report = EmailOperator(  
 task\_id='email\_report',  
 to='airflow@datacamp.com',  
 subject='Airflow Monthly Report',  
 html\_content="""Attached is your monthly workflow report - please refer to it for more detail""",  
 files=['monthly\_report.pdf'],  
 dag=report\_dag  
)

# Set the email task to run after the report is generated  
email\_report << generate\_report

Airflow will now email you with an attached report file after the generate\_report task completes. You can use Airflow's functionality to send updates via many methods in addition to email. Make sure to look through the documentation for other ideas on monitoring your workflows.

**Adding status emails**

You’ve worked through most of the Airflow configuration for setting up your workflows, but you realize you’re not getting any notifications when DAG runs complete or fail. You’d like to setup email alerting for the success and failure cases, but you want to send it to two addresses.



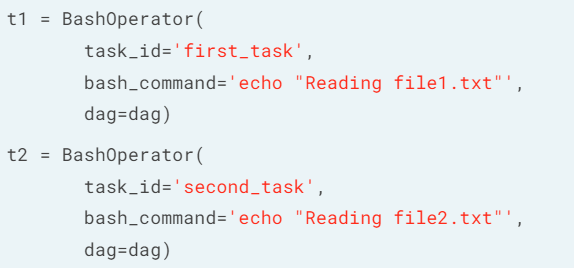
The workflow is successfully configured to send you email alerts when the DAG completes successfully or fails. Use these options in production to monitor the state of your workflows to help avoid surprises.

**Chapter 4. Building production pipelines in Airflow**

Use what you’ve learned to build a production quality workflow in Airflow.

**Working with templates**

Templates allow substitution of information during a DAG run, and provide added flexibility when defining tasks.



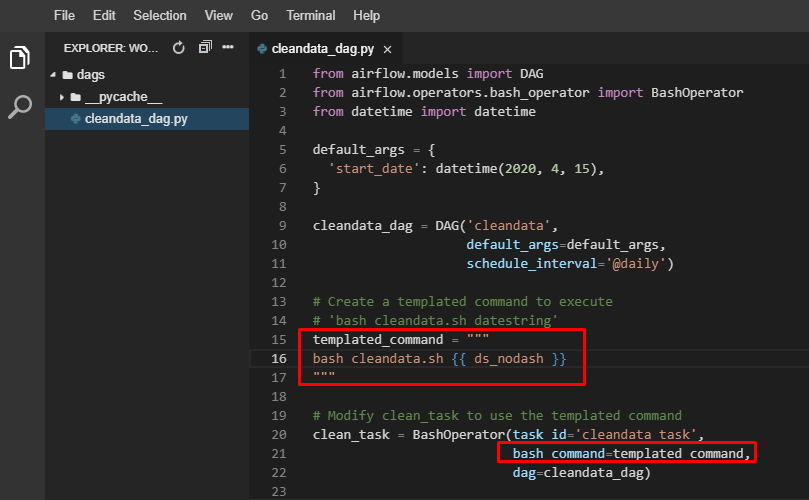
For example, the repetitive code above can be replaced with templated BashOperator.



**Creating a templated BashOperator**

You’ve successfully created a BashOperator that cleans a given data file by executing a script called cleandata.sh. This works, but unfortunately requires the script to be run only for the current day. Some of your data sources are occasionally behind by a couple of days and need to be run manually.

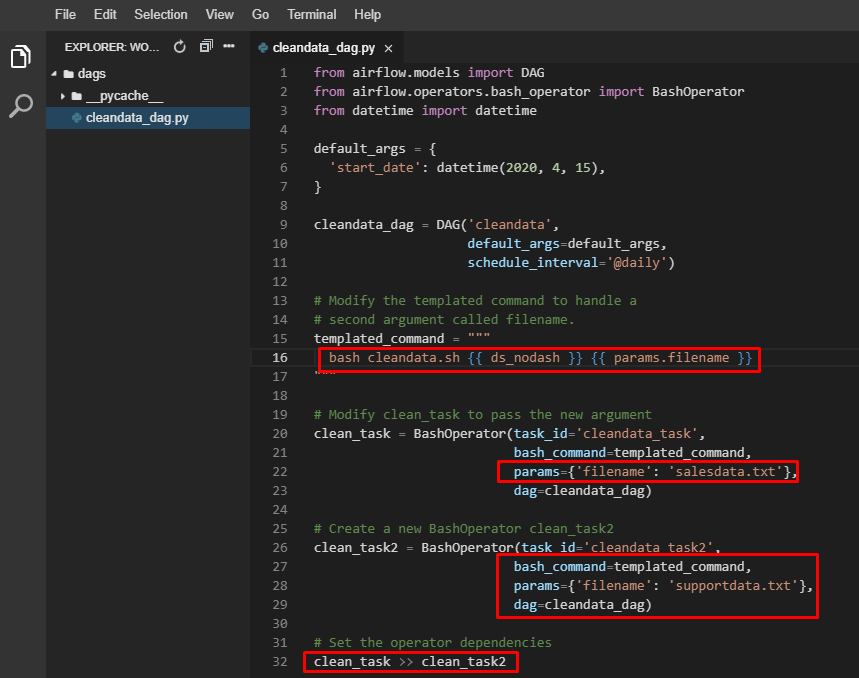
You successfully modify the cleandata.sh script to take one argument - the date in YYYYMMDD format. Your testing works at the command-line, but you now need to implement this into your Airflow DAG. For now, use the term {{ ds\_nodash }} in your template - you'll see exactly what this is means later on.



The DAG has been modified to use a templated command instead of hardcoding your workflow objects. This will come in very handy when creating production workflows. Note that for now, we didn’t need to define a params argument in the BashOperator — this is ok as Airflow handles passing some data into templates automatically for us.

**Templates with multiple arguments**

You wish to build upon your previous DAG and modify the code to support two arguments — the date in YYYYMMDD format, and a file name passed to the cleandata.sh script.



Making use of multiple operators that vary by the parameters is a great use of templated commands in Airflow!

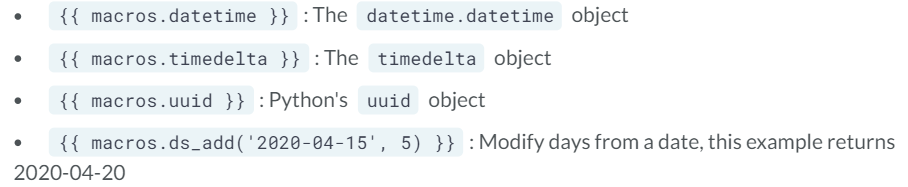
**More templates**



Airflow built-in **runtime variables** provides information about DAG runs, tasks, and even the system configuration.



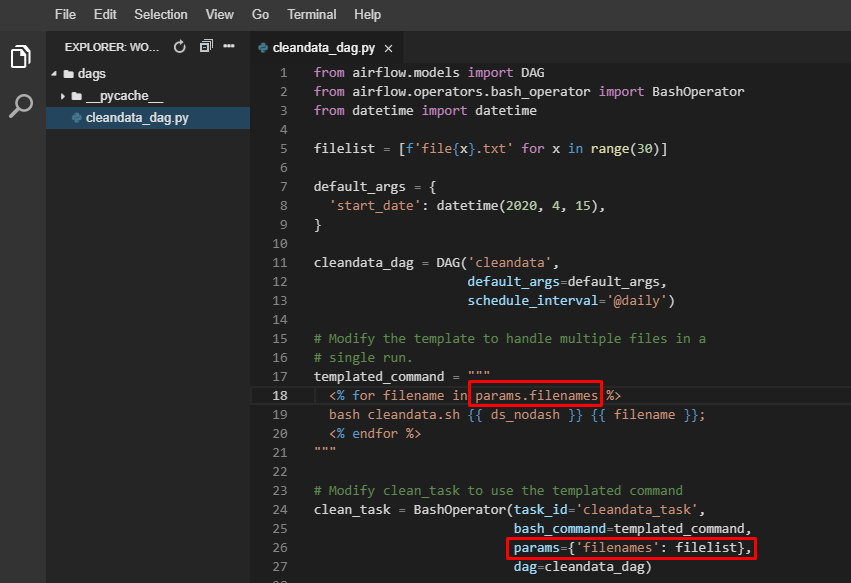
Macros variable is a reference to the Airflow macros package which provides various useful objects/methods for Airflow templates.



**Using lists with templates**

Once again, you decide to make some modifications to the design of your cleandata workflow. This time, you realize that you need to run the command cleandata.sh with the date argument and the file argument as before, except now you have a list of 30 files. You do *not* want to create 30 tasks, so your job is to modify the code to support running the argument for 30 or more files.

The Python list of files is already created for you, simply called filelist.



You’ve successfully implemented a Jinja template to iterate over the files in a list and execute a bash command for each file. This type of flexibility and power provides a lot of options to best configure a workflow using Airflow.

**Understanding parameter options**

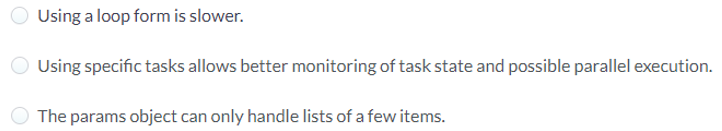
You’ve used a few different methods to add templates to your workflows. Considering the differences between options, why would you want to create individual tasks (ie, BashOperators) with specific parameters vs a list of files?

For example, why would you choose

t1 = BashOperator(task\_id='task1', bash\_command=templated\_command,   
 params={'filename': 'file1.txt'}, dag=dag)  
t2 = BashOperator(task\_id='task2', bash\_command=templated\_command,   
 params={'filename': 'file2.txt'}, dag=dag)  
t3 = BashOperator(task\_id='task3', bash\_command=templated\_command,   
 params={'filename': 'file3.txt'}, dag=dag)

over using a loop form such as

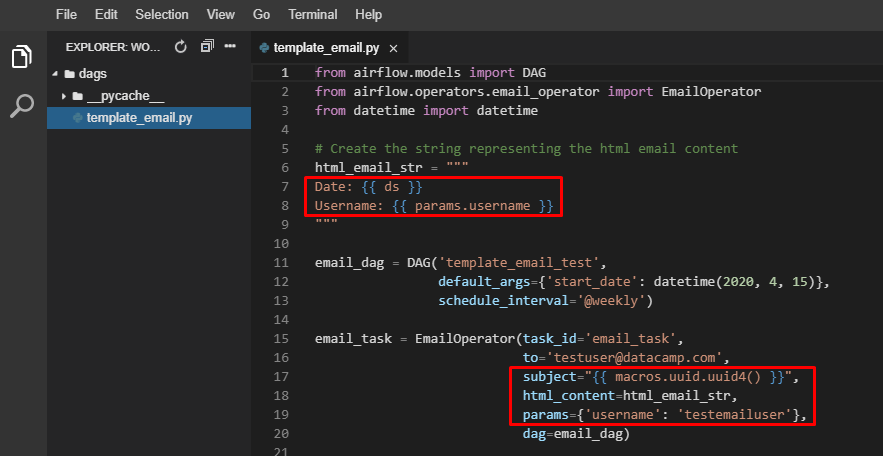
t1=BashOperator(task\_id='task1',   
 bash\_command=templated\_command,   
 params={'filenames': ['file1.txt', 'file2.txt', 'file3.txt']},  
 dag=dag)



Answer: Using specific tasks allows better monitoring of **task state** and possible **parallel**execution. When using a single task, all entries would succeed or fail as a single task. Separate operators allow for better monitoring and scheduling of these tasks.

**Sending templated emails**

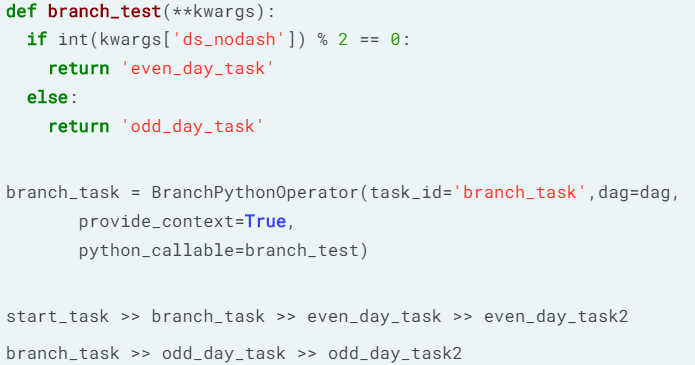
While reading through the Airflow documentation, you realize that various operations can use templated fields to provide added flexibility. You come across the docs for the EmailOperator and see that the content can be set to a template. You want to make use of this functionality to provide more detailed information regarding the output of a DAG run.

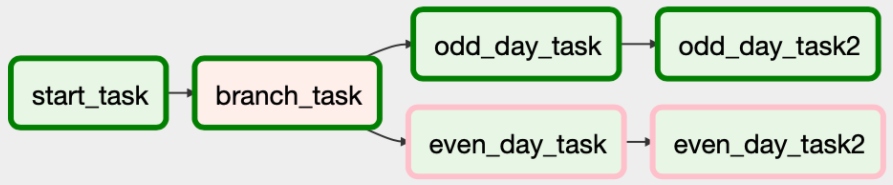


As mentioned, there are many operators that can accept templated fields. When browsing the documentation, if a field is referred to as *templated*, it can use these techniques.

**Branching**

Branching provides conditional logic (tasks can be selectively executed or skipped), using **BranchPythonOperator**, which takes a python\_callable to return the next task\_id (or list of ids) to follow.





Workflow on an odd day (not even day)

**Define a BranchPythonOperator**

After learning about the power of conditional logic within Airflow, you wish to test out the BranchPythonOperator. You’d like to run a different code path if the current execution date represents a new year (ie, 2020 vs 2019).

The DAG is defined for you, along with the tasks in question. Your current task is to implement the BranchPythonOperator.

# Create a function to determine if years are different  
def year\_check(\*\*kwargs):  
 current\_year = int(kwargs['ds\_nodash'][0:4])  
 previous\_year = int(kwargs['prev\_ds\_nodash'][0:4])  
 if current\_year == previous\_year:  
 return 'current\_year\_task'  
 else:  
 return 'new\_year\_task'

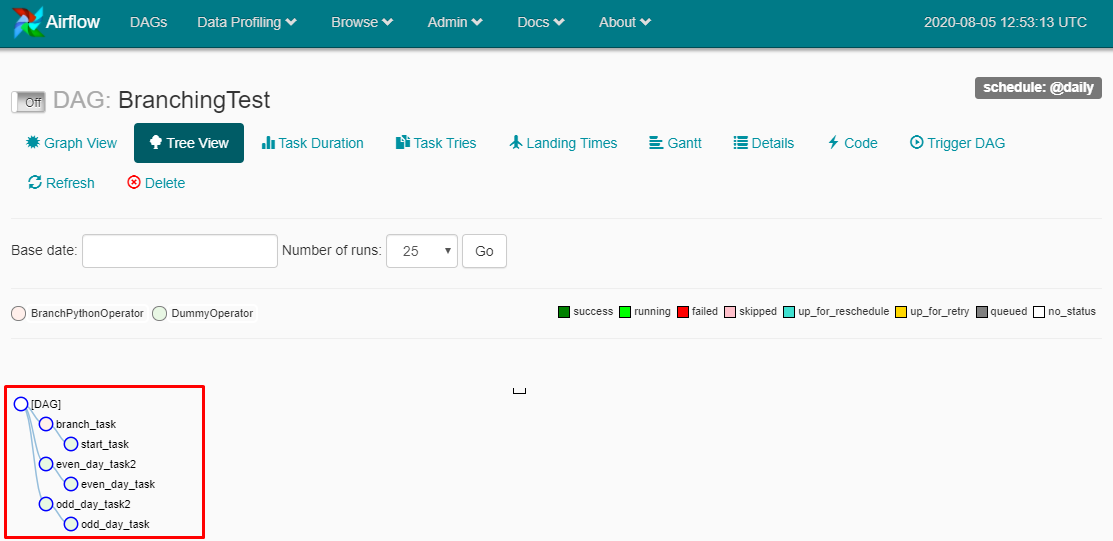
# Define the BranchPythonOperator  
branch\_task = BranchPythonOperator(task\_id='branch\_task',   
 dag=branch\_dag, python\_callable=year\_check,   
 provide\_context=True)

# Define the dependencies  
branch\_dag >> current\_year\_task  
branch\_dag >> new\_year\_task

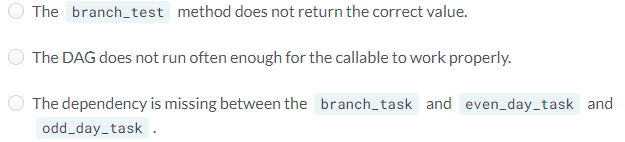
This is a simple but effective use of branching to perform an occasional set of tasks without requiring significant code changes. Make sure to remember the various capabilities with branching to make your workflows more robust.

**Branch troubleshooting**

While working with a workflow defined by a colleague, you notice that a branching operator executes, but there’s never any change in the DAG results. You realize that regardless of the state defined by the branching operator, all other tasks complete, even as some should be skipped.

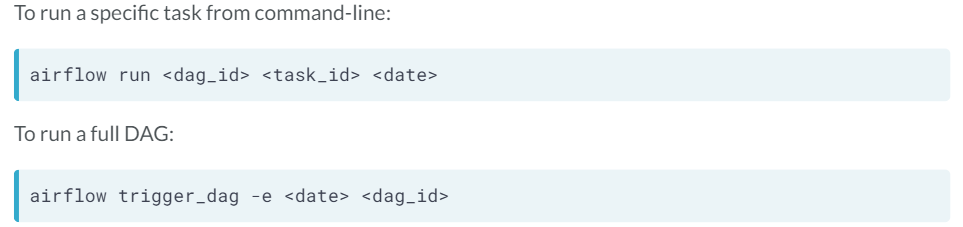


Use what you’ve learned to determine the most likely reason that the branching operator is ineffective.

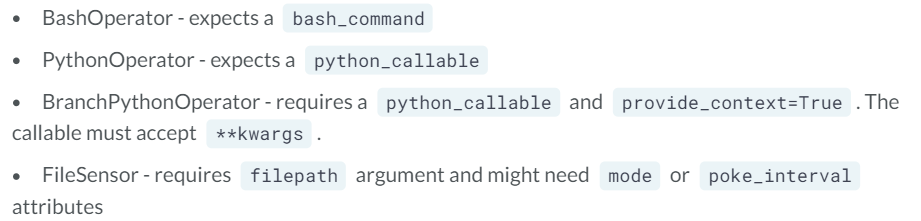


Answer: The dependency is missing between the branch\_task and even\_day\_task and odd\_day\_task. Always remember to look for the simple issues first before trying to modify your code or processes too deeply.

**Creating a production pipeline**



Operators recap:

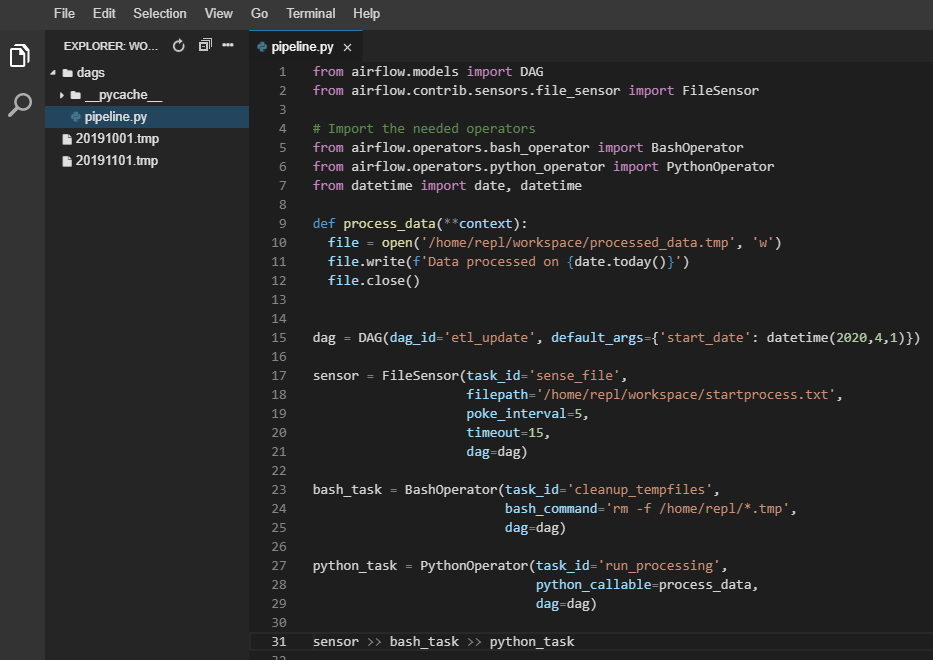


**Creating a production pipeline #1**

Now it’s time to implement your workflow into a production pipeline consisting of many objects including sensors and operators. Your boss is interested in seeing this workflow become automated and able to provide SLA reporting as it provides some extra leverage for closing a deal the sales staff is working on. The sales prospect has indicated that once they see updates in an automated fashion, they’re willing to sign-up for the indicated data service.

From what you’ve learned about the process, you know that there is sales data that will be uploaded to the system. Once the data is uploaded, a new file should be created to kick off the full processing, but something isn’t working correctly.

Refer to the source code of the DAG to determine if anything extra needs to be added.

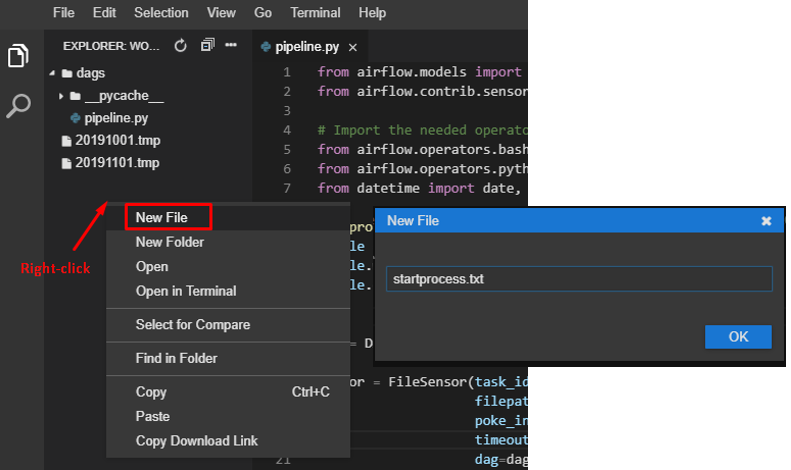


from airflow.models import DAG  
from airflow.contrib.sensors.file\_sensor import FileSensor

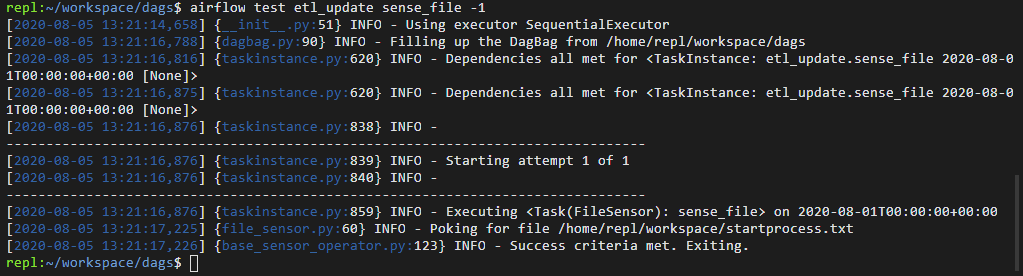
# Import the needed operators  
from airflow.operators.bash\_operator import BashOperator  
from airflow.operators.python\_operator import PythonOperator  
from datetime import date, datetimedef process\_data(\*\*context):  
 file = open('/home/repl/workspace/processed\_data.tmp', 'w')  
 file.write(f'Data processed on {date.today()}')  
 file.close()dag = DAG(dag\_id='etl\_update',   
 default\_args={'start\_date': datetime(2020,4,1)})sensor = FileSensor(task\_id='sense\_file',   
 filepath='/home/repl/workspace/startprocess.txt',  
 poke\_interval=5,  
 timeout=15,  
 dag=dag)bash\_task = BashOperator(task\_id='cleanup\_tempfiles',   
 bash\_command='rm -f /home/repl/\*.tmp',  
 dag=dag)python\_task = PythonOperator(task\_id='run\_processing',   
 python\_callable=process\_data,  
 dag=dag)sensor >> bash\_task >> python\_task

Run this at command line: airflow test etl\_update sense\_file -1

Snap! Time is out. You ran the correct command though, to find out why the sense\_file task would not complete. It's looking for a startprocess.txt file and it's not finding it, so it keeps poking every 5 seconds to see if it's there. You just need to create this file! You can use the touch command in the terminal, or right click and select "New File" in the menu on the left of the editor to create startprocess.txt (empty text file).



Run this at command line: airflow test etl\_update sense\_file -1



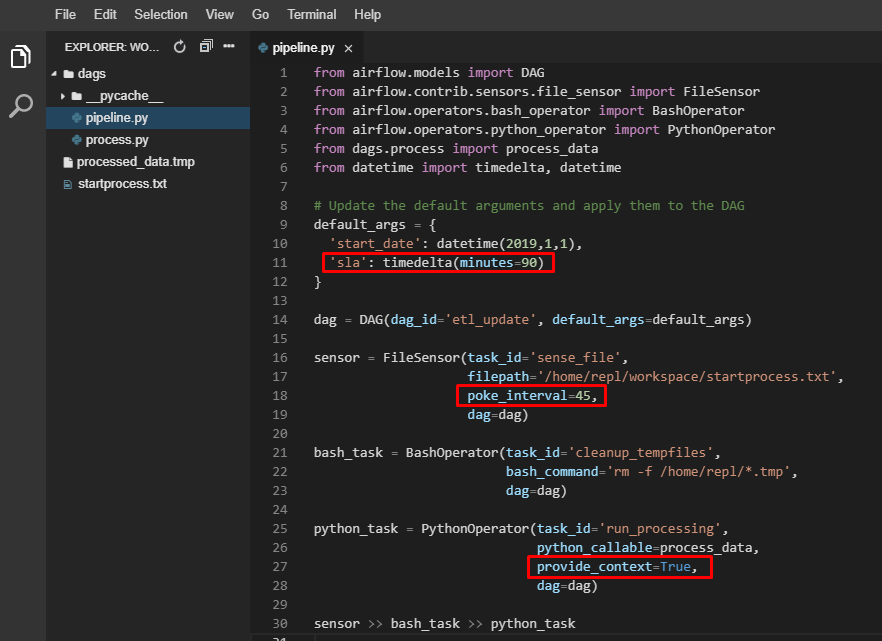
Successful run!

You’ve just successfully modified and troubleshot a DAG within Airflow. Nice job verifying the startprocess.txt file existed to allow the DAG to continue. While this DAG is relatively simple, it implements many components of a production level workflow. These same troubleshooting principles can assist you when building a production system.

**Creating a production pipeline #2**

Continuing on your last workflow, you’d like to add some additional functionality, specifically adding some SLAs to the code and modifying the sensor components.

Refer to the source code of the DAG to determine if anything extra needs to be added. The default\_args dictionary has been defined for you, though it may require further modification.



from airflow.models import DAG  
from airflow.contrib.sensors.file\_sensor import FileSensor  
from airflow.operators.bash\_operator import BashOperator  
from airflow.operators.python\_operator import PythonOperator  
from dags.process import process\_data  
from datetime import timedelta, datetime

# Update the default arguments and apply them to the DAG  
default\_args = {  
 'start\_date': datetime(2019,1,1),  
 'sla': timedelta(minutes=90)  
}dag = DAG(dag\_id='etl\_update', default\_args=default\_args)sensor = FileSensor(task\_id='sense\_file',   
 filepath='/home/repl/workspace/startprocess.txt',  
 poke\_interval=45,  
 dag=dag)bash\_task = BashOperator(task\_id='cleanup\_tempfiles',   
 bash\_command='rm -f /home/repl/\*.tmp',  
 dag=dag)python\_task = PythonOperator(task\_id='run\_processing',   
 python\_callable=process\_data,  
 provide\_context=True,  
 dag=dag)sensor >> bash\_task >> python\_task

You’ve correctly added support for SLAs in this DAG and modified the file sensor object to only look for its file every 45 seconds. These types of incremental improvements are often used when creating workflows in production. You may have also noticed that we’re using the provide\_context entry with the PythonOperator, rather than just the BranchPythonOperator. Most operators within Airflow can accept the provide\_context argument for the intended purpose.

**Adding the final changes to your pipeline**

To finish up your workflow, your manager asks that you add a conditional logic check to send a sales report via email, only if the day is a weekday. Otherwise, no email should be sent. In addition, the email task should be templated to include the date and a project name in the content.

from airflow.models import DAG  
from airflow.contrib.sensors.file\_sensor import FileSensor  
from airflow.operators.bash\_operator import BashOperator  
from airflow.operators.python\_operator import PythonOperator  
from airflow.operators.python\_operator import BranchPythonOperator  
from airflow.operators.dummy\_operator import DummyOperator  
from airflow.operators.email\_operator import EmailOperator  
from dags.process import process\_data  
from datetime import datetime, timedelta

# Update the default arguments and apply them to the DAG.  
default\_args = {  
 'start\_date': datetime(2019,1,1),  
 'sla': timedelta(minutes=90)  
}  
   
dag = DAG(dag\_id='etl\_update', default\_args=default\_args)sensor = FileSensor(task\_id='sense\_file',   
 filepath='/home/repl/workspace/startprocess.txt',  
 poke\_interval=45,  
 dag=dag)bash\_task = BashOperator(task\_id='cleanup\_tempfiles',   
 bash\_command='rm -f /home/repl/\*.tmp',  
 dag=dag)python\_task = PythonOperator(task\_id='run\_processing',   
 python\_callable=process\_data,  
 provide\_context=True,  
 dag=dag)email\_subject="""  
 Email report for {{ params.department }} on {{ ds\_nodash }}  
"""email\_report\_task=EmailOperator(task\_id='email\_report\_task',  
 to='sales@mycompany.com',  
 subject=email\_subject,  
 html\_content='',  
 params={'department':'Data subscription services'},  
 dag=dag)no\_email\_task = DummyOperator(task\_id='no\_email\_task', dag=dag)def check\_weekend(\*\*kwargs):  
 dt = datetime.strptime(kwargs['execution\_date'],'%Y-%m-%d')  
 #If dt.weekday() is 0-4, it's Mon-Fri. If 5-6, it's Sat/Sun  
 if (dt.weekday() < 5):  
 return 'email\_report\_task'  
 else:  
 return 'no\_email\_task'  
   
branch\_task = BranchPythonOperator(task\_id='check\_if\_weekend',  
 python\_callable=check\_weekend,  
 provide\_context=True,  
 dag=dag)sensor >> bash\_task >> python\_taskpython\_task >> branch\_task >> [email\_report\_task, no\_email\_task]

You’ve completed building a complex workflow using almost everything we’ve learned during this course — Operators, tasks, sensors, conditional logic, templating, SLAs, dependencies, and even alerting!

**Summary**

Congratulations! Let’s recap what we have learnt:

* Workflows / DAGs / Tasks
* Operators (BashOperator, PythonOperator, BranchPythonOperator, EmailOperator)
* Dependencies between tasks / Bitshift operators
* Sensors (to react to workflow conditions and state)
* Scheduling DAGs
* SLAs / Alerting to maintain visibility on workflows
* Templates for maximum flexibility when defining tasks
* Branching, to add conditional logic to DAGs
* Airflow interfaces: command line / UI
* Airflow executors
* Debugging / Troubleshooting

**Next steps**

* Setup your own environment for practice
* Other operators (eg. Amazon’s S3, Postgresql) / sensors (eg. HDFS)
* Experiment with dependencies with a large number of tasks
* Look into parts of Airflow: XCom, Connections, etc
* Refer to Airflow documentations
* Keep building workflows

Happy learning Airflow!