DAG

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 can’t create a path back to themselves.
- **Directed:** It means that one RDD is connected to other RDD in a direction, which shows transformations (Could be narrow or wide transformations).
- **Acyclic:** Nodes can’t create a path back to themselves. Thus no cyclic graph.
- **Graph:** DAGs are a special subset of graphs in which the edges between nodes have a specific direction.
- **Nodes:** A step in the data pipeline process.
- **Edges:** The dependencies or relationships other between nodes.

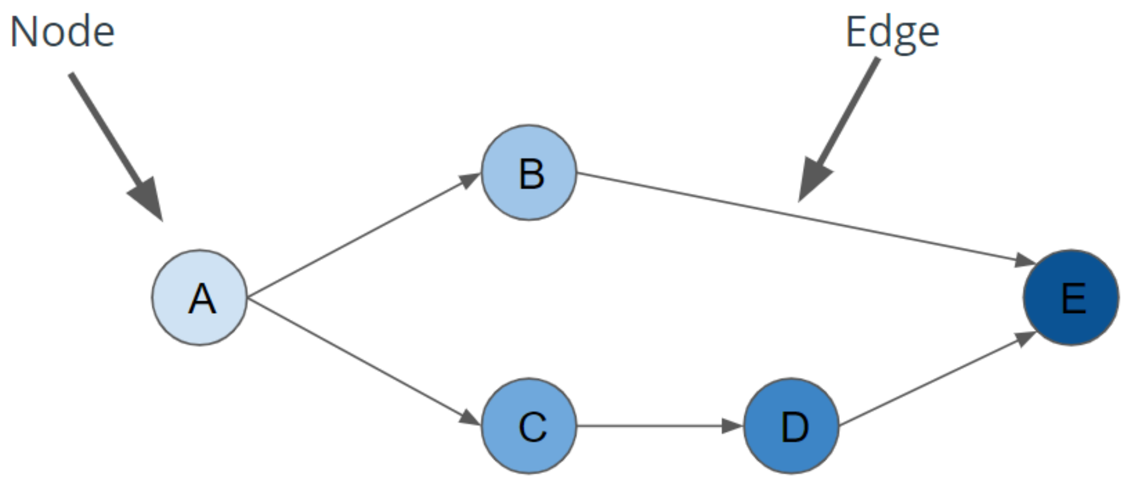


Diagram of a Directed Acyclic Graph

Apache Airflow

Airbnb open-sourced Airflow in 2015 with the goal of creating a **DAG-based, schedulable**, **data-pipeline tool** that could run in mission-critical environments.

Since then, **hundreds of companies** have successfully integrated Airflow to manage and define their data pipelines. A few highlights include HBO, Spotify, Lyft, Paypal, Google, and Stripe.

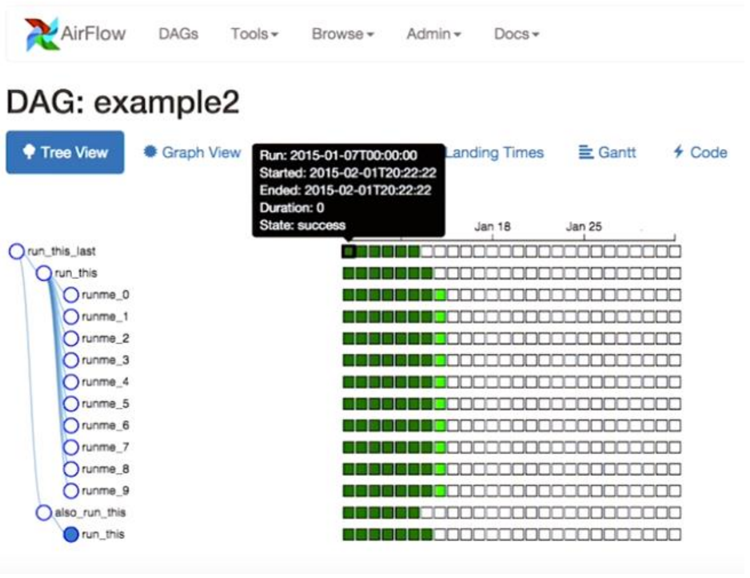
Apache Airflow

Airflow allows users to write DAGs in Python that run on a schedule and/or from an external trigger.

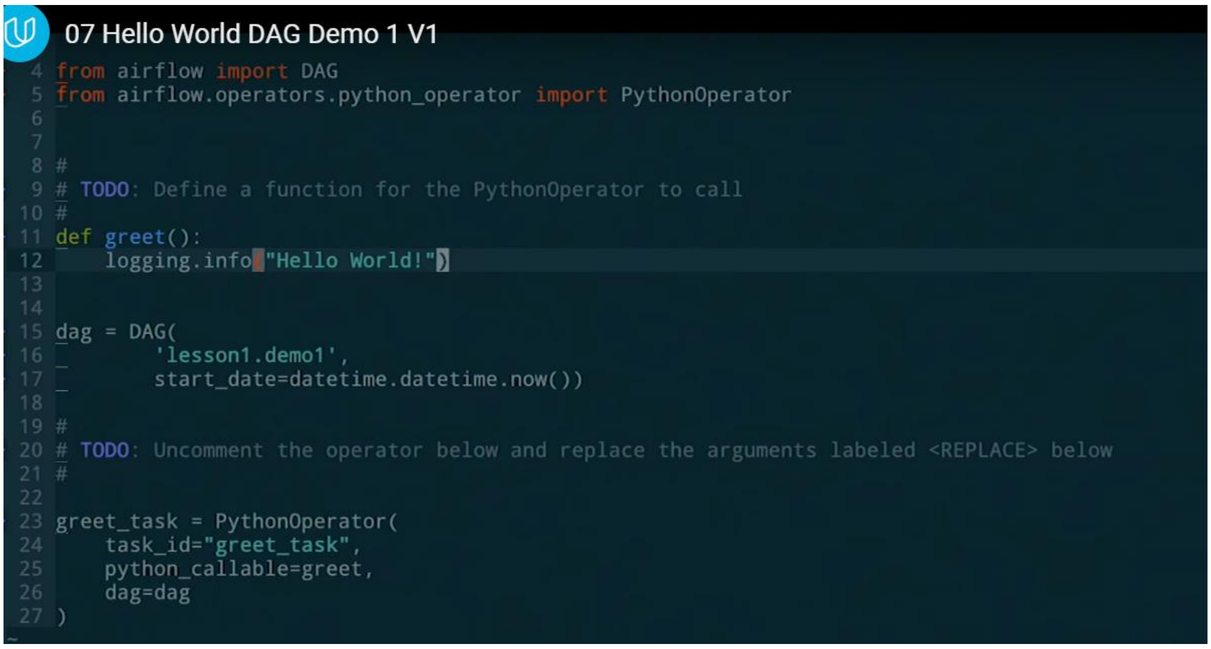
Airflow is simple to maintain and can run data analysis itself, or trigger external tools (Redshift, Spark, Presto, Hadoop, etc) during execution.

Apache Airflow

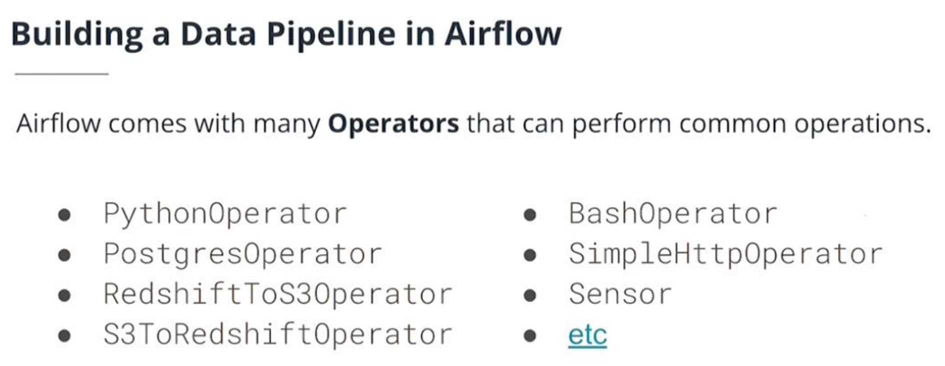
Airflow also provides a web-based UI for users to visualize and interact with their data pipelines.



Run: 2015-01-07T00:00:00
Started: 2015-02-01T20:22:22
Ended: 2015-02-01T20:22:22
Duration: 0
State: success



```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator


#
# TODO: Define a function for the PythonOperator to call
#
def greet():
    logging.info("Hello World!")


dag = DAG(
        'lesson1.demo1',
        start_date=datetime.datetime.now())

#
# TODO: Uncomment the operator below and replace the arguments labeled <REPLACE> below
#

greet_task = PythonOperator(
    task_id="greet_task",
    python_callable=greet,
    dag=dag
)
```



Building a Data Pipeline in Airflow

Airflow comes with many **Operators** that can perform common operations.

- `PythonOperator`
- `PostgresOperator`
- `RedshiftToS3Operator`
- `S3ToRedshiftOperator`
- `BashOperator`
- `SimpleHttpOperator`
- `Sensor`
- etc

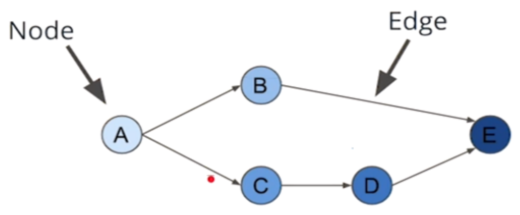
Operators

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`

Building a Data Pipeline in Airflow: Task Dependencies
(Option 1)

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

Task Dependencies in Airflow DAGs:

- Nodes = Tasks; Task can also be defined as the instantiation of the parameterized Operators. It means an instance of an operator along with required parameters is known as task.

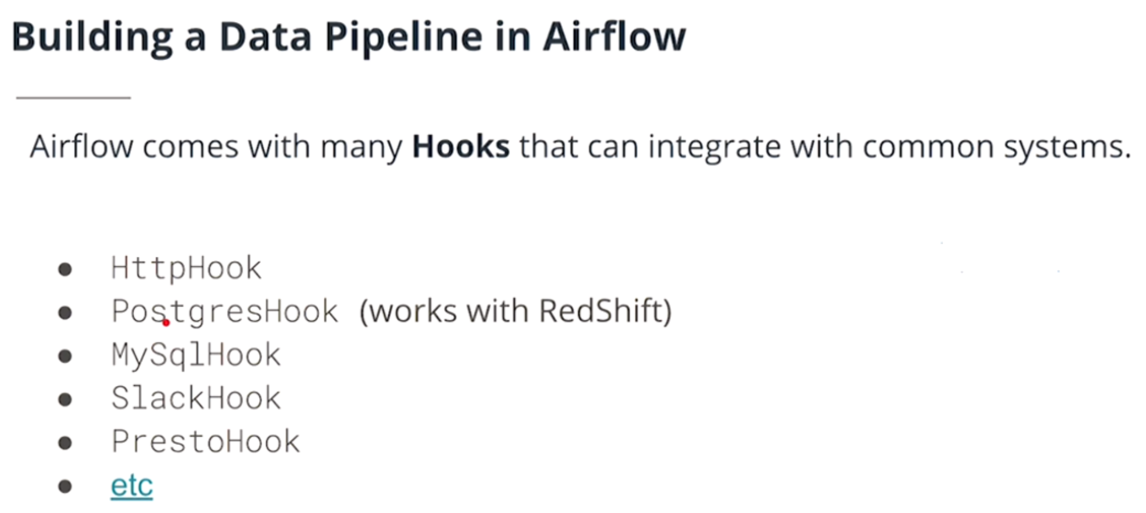
- Edges = Ordering and dependencies between tasks
  Task dependencies can be described programmatically in Airflow using `>>` and `<<`
- a `>>` b means a comes before b
- a `<<` b means a comes after b

```
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)
```



Building a Data Pipeline in Airflow

Airflow comes with many **Hooks** that can integrate with common systems.

- HttpHook
- PostgresHook (works with RedShift)
- MySqlHook
- SlackHook
- PrestoHook
- etc

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`

EXAMPLE of HOOKs and VARIABLEs:

```
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


## We will define a function, which will be meant to print all the file names in a
## S3 bucket.

#-------------------------------------------------------
def list_keys():

    ## S3Hook function helps us to connect S3 with our application so that we can    access the files
in S3 buckets. For doing this we can use the connection tab in the web UI of the airflow and enter all
the login credentials for aws and all save   inside the connection name 'aws_credentials'. ##

    hook = S3Hook(aws_conn_id='aws_credentials')

    ## Variables can also be created to save the stuffs which could change in future or which are exp
ected to change often. Here we have created the bucket name    as a variable. Variable is a saved as a
key value pair in web ui of airflow. To extract the bucket name, we need to call the corresponding key
's3_bucket' as given the code below. ##

    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
)
```

Understand args and kwargs for CONTEXT VARIABLES

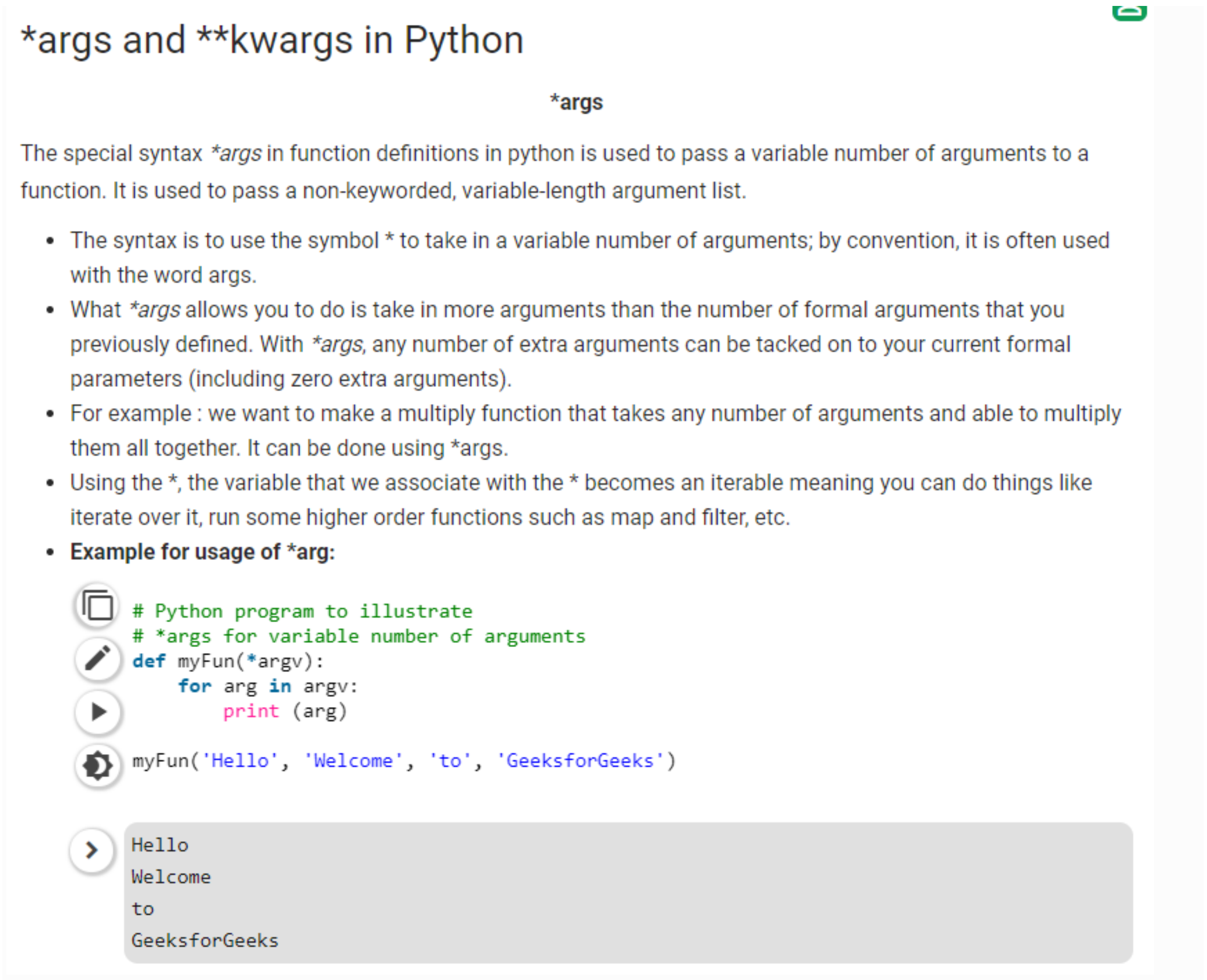
Before understanding context variables it is imperical understand args and kwargs in python.

1. *args and **kwargs

I have come to see that most new python programmers have a hard time figuring out the *args and **kwargs magic variables. So what are they ? First of all let me tell you that it is not necessary to write *args or **kwargs. Only the `*` (asterisk) is necessary. You could have also written *var and **vars. Writing *args and **kwargs is just a convention. So now lets take a look at *args first.

1.1. Usage of *args

*args and **kwargs are mostly used in function definitions. *args and **kwargs allow you to pass a variable number of arguments to a function. What variable means here is that you do not know beforehand how many arguments can be passed to your function by the user so in this case you use these two keywords. *args is used to send a **non-keyworded** variable length argument list to the function. Here’s an example to help you get a clear idea:



*args and **kwargs in Python

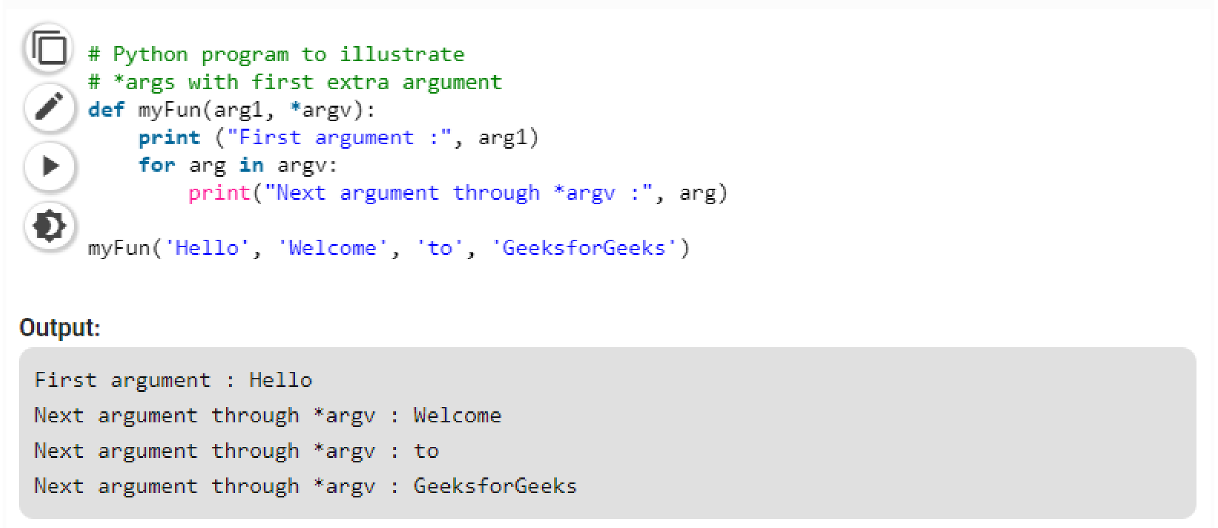
***args**

The special syntax **args* in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.

- The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.
- What **args* allows you to do is take in more arguments than the number of formal arguments that you previously defined. With **args*, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
- For example : we want to make a multiply function that takes any number of arguments and able to multiply them all together. It can be done using *args.
- Using the *, the variable that we associate with the * becomes an iterable meaning you can do things like iterate over it, run some higher order functions such as map and filter, etc.
- **Example for usage of *arg:**

```python
# Python program to illustrate
# *args for variable number of arguments
def myFun(*argv):
    for arg in argv:
        print (arg)

myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

```
Hello
Welcome
to
GeeksforGeeks
```



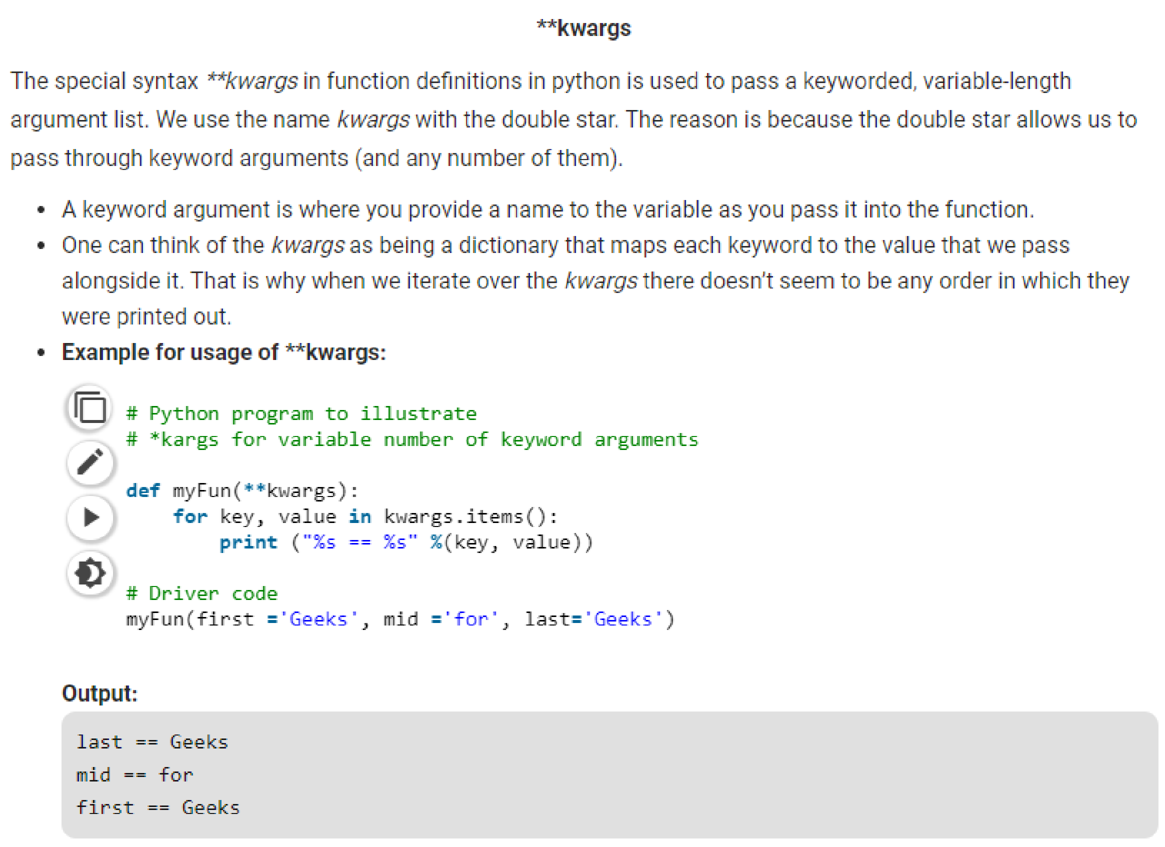
```python
# Python program to illustrate
# *args with first extra argument
def myFun(arg1, *argv):
    print ("First argument :", arg1)
    for arg in argv:
        print("Next argument through *argv :", arg)

myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output:

```
First argument : Hello
Next argument through *argv : Welcome
Next argument through *argv : to
Next argument through *argv : GeeksforGeeks
```

1.2. Usage of **kwargs

kwargs allows you to pass **keyworded variable length of arguments to a function. You should use **kwargs if you want to handle **named arguments** in a function. Here is an example to get you going with it:


****kwargs**

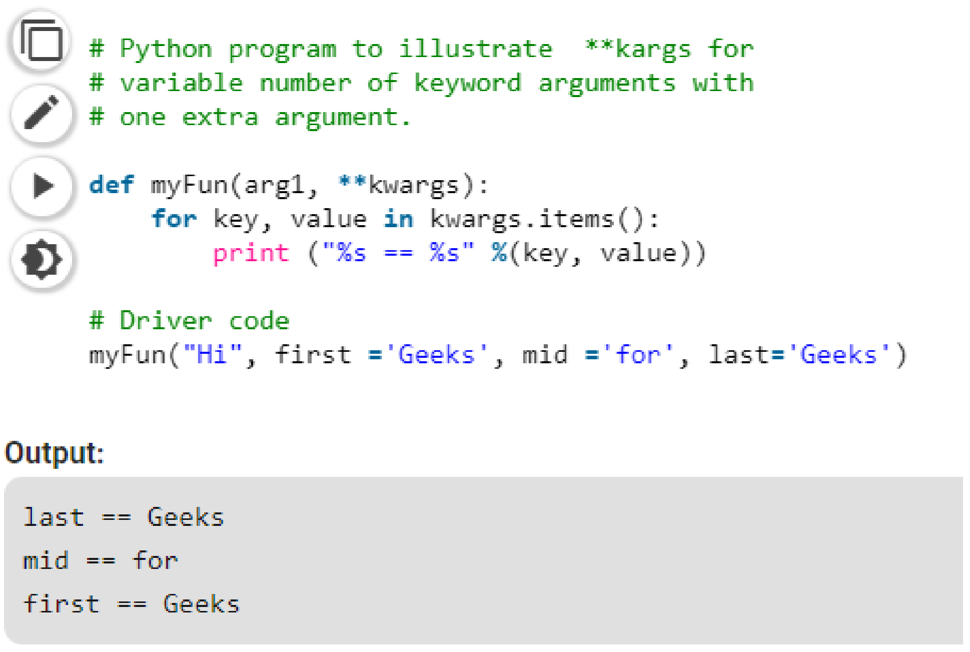
The special syntax ***kwargs* in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name *kwargs* with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn't seem to be any order in which they were printed out.
- **Example for usage of **kwargs:**

```
# Python program to illustrate
# *kargs for variable number of keyword arguments

def myFun(**kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

# Driver code
myFun(first ='Geeks', mid ='for', last='Geeks')
```

Output:

```
last == Geeks
mid == for
first == Geeks
```



```
# Python program to illustrate  **kargs for
# variable number of keyword arguments with
# one extra argument.

def myFun(arg1, **kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

# Driver code
myFun("Hi", first ='Geeks', mid ='for', last='Geeks')
```

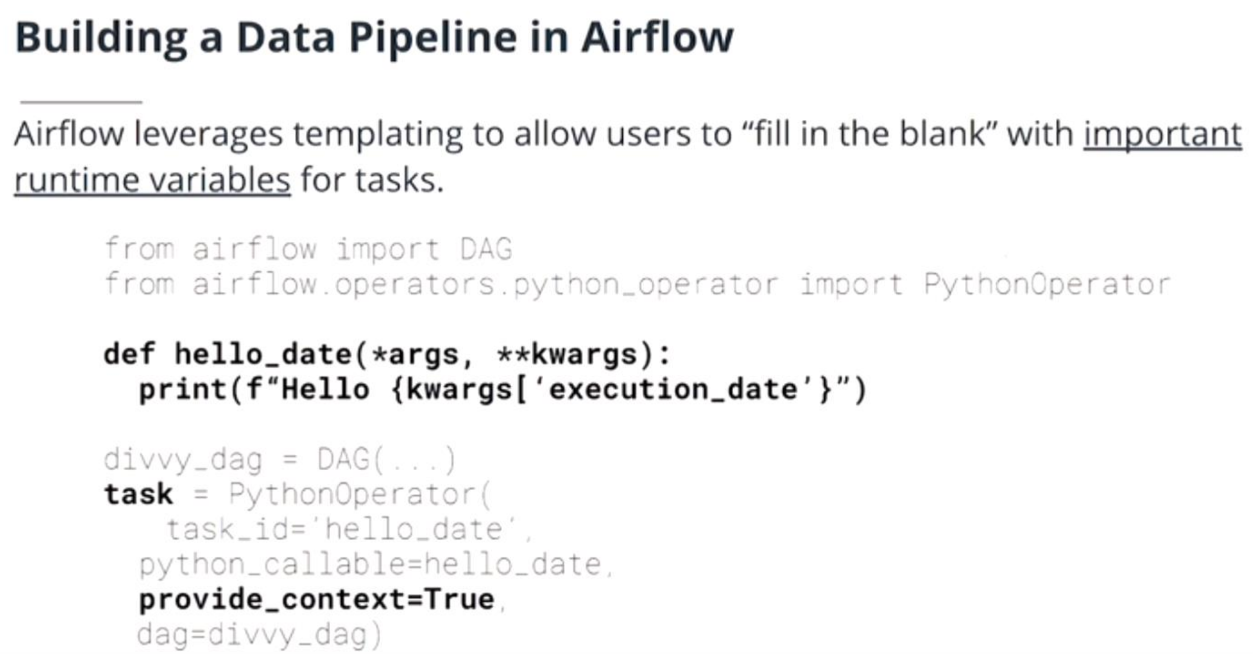
Output:

```
last == Geeks
mid == for
first == Geeks
```

CONTEXT or DEFAULT VARIABLES:

Airflow passes several default variables to its operators for the dags and tasks. These can be availed through using *args, and **kwargs in any operator function of a task or dag. These are also called context variables because they are linked with dags and their tasks such as 'task execution time', 'Dag execution time' etc. For the exhaustive list of all the variables check the below mentioned link. Always include parameter ‘provide_context = True’ in operator instantiation, when we are using context variables.

https://airflow.apache.org/docs/stable/macros.html



Building a Data Pipeline in Airflow

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)
```

`python print(f"Hello {kwargs['execution_date']}")`

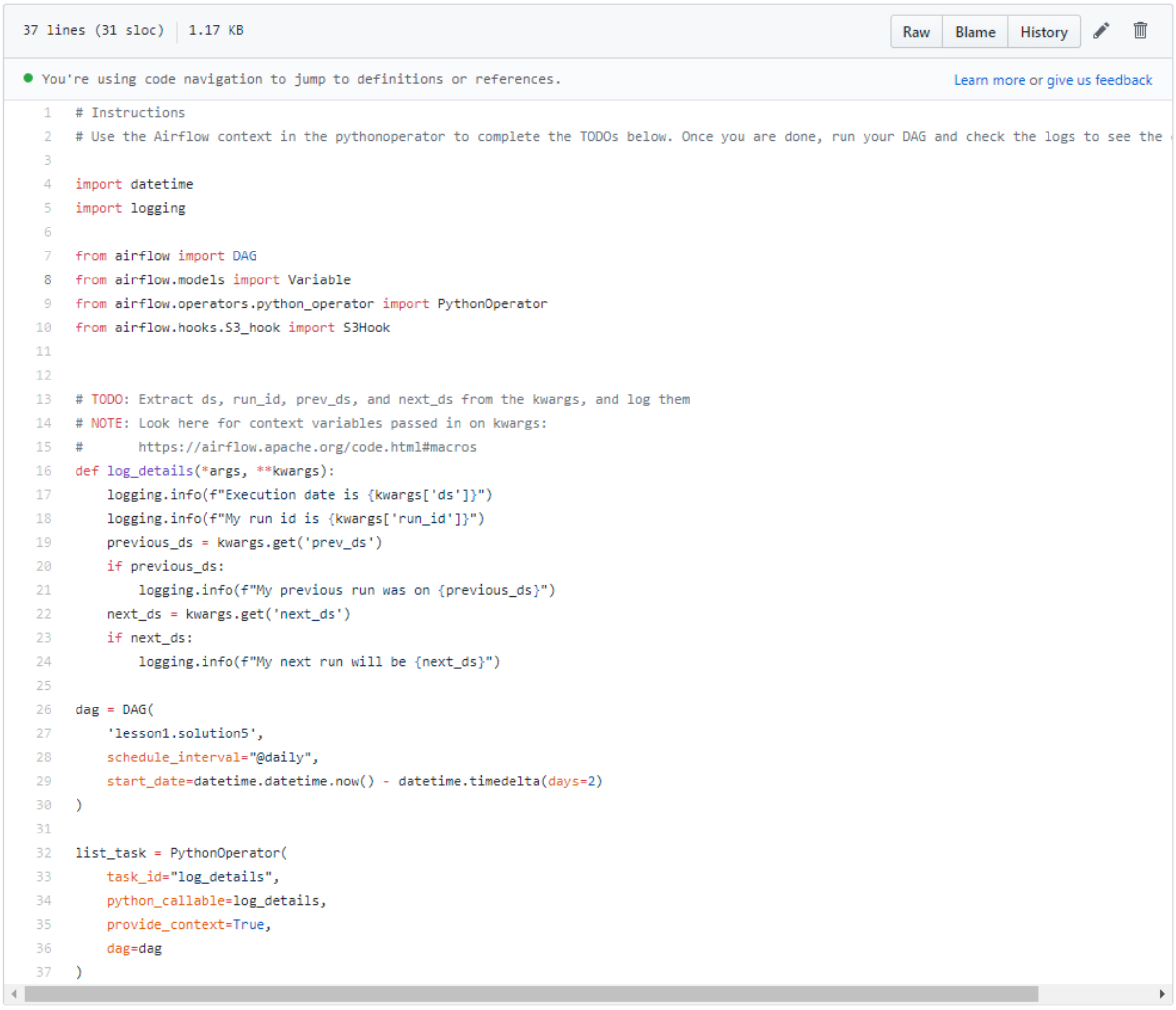
`There are several ways of accessing the context variables:`

1. `kwargs['execution_date']`
2. `kwargs.get('execution_date')`

Here, kwargs is a context variables because it contains the various run time execution related data; such as ‘execution time’, ‘run_id’ of the task, etc given in the below mentioned link. Always include parameter ‘provide_context = True’, when we are using context variables.

https://airflow.apache.org/docs/stable/macros.html

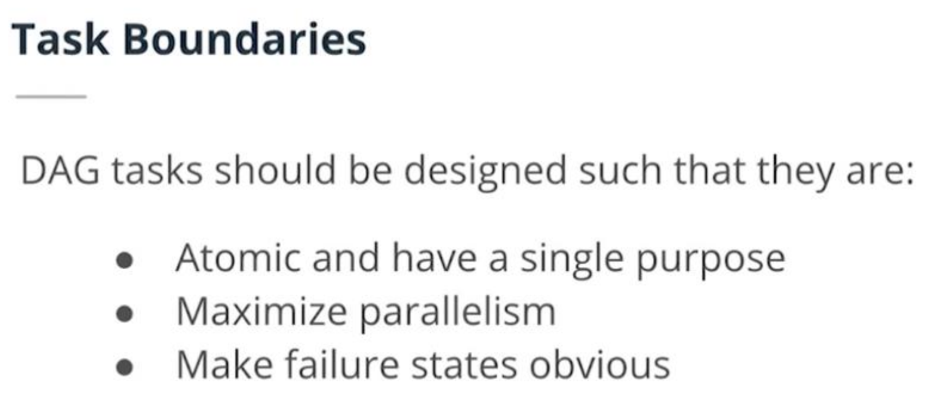
Context Variables are helpful in processing and accessing the data before processing. (How??)


37 lines (31 sloc) | 1.17 KB

You're using code navigation to jump to definitions or references.

```python
# Instructions
# Use the Airflow context in the pythonoperator to complete the TODOs below. Once you are done, run your DAG and check the logs to see the

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


# TODO: Extract ds, run_id, prev_ds, and next_ds from the kwargs, and log them
# NOTE: Look here for context variables passed in on kwargs:
#       https://airflow.apache.org/code.html#macros
def log_details(*args, **kwargs):
    logging.info(f"Execution date is {kwargs['ds']}")
    logging.info(f"My run id is {kwargs['run_id']}")
    previous_ds = kwargs.get('prev_ds')
    if previous_ds:
        logging.info(f"My previous run was on {previous_ds}")
    next_ds = kwargs.get('next_ds')
    if next_ds:
        logging.info(f"My next run will be {next_ds}")

dag = DAG(
    'lesson1.solution5',
    schedule_interval="@daily",
    start_date=datetime.datetime.now() - datetime.timedelta(days=2)
)

list_task = PythonOperator(
    task_id="log_details",
    python_callable=log_details,
    provide_context=True,
    dag=dag
)
```

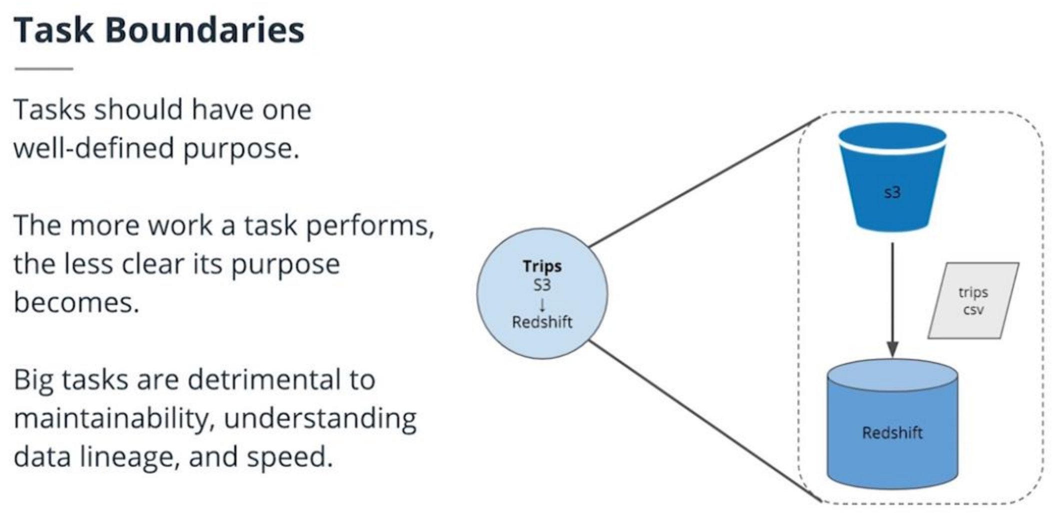
Best Practices:



Task Boundaries

DAG tasks should be designed such that they are:

- Atomic and have a single purpose
- Maximize parallelism
- Make failure states obvious



Task Boundaries

Tasks should have one well-defined purpose.

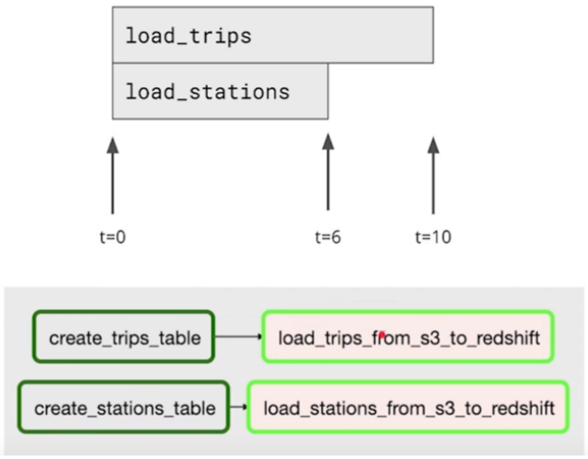
The more work a task performs, the less clear its purpose becomes.

Big tasks are detrimental to maintainability, understanding data lineage, and speed.

Task Boundaries

Properly scoped tasks minimize dependencies and are often more easily parallelized.

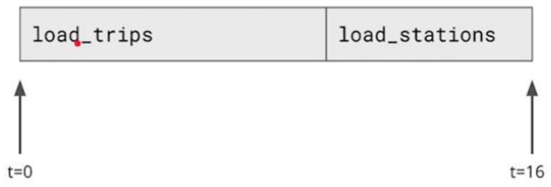
Parallelization can offer a significant speedup in the execution of Airflow DAGs.



Task Boundaries

Properly scoped tasks minimize dependencies and are often more easily parallelized.

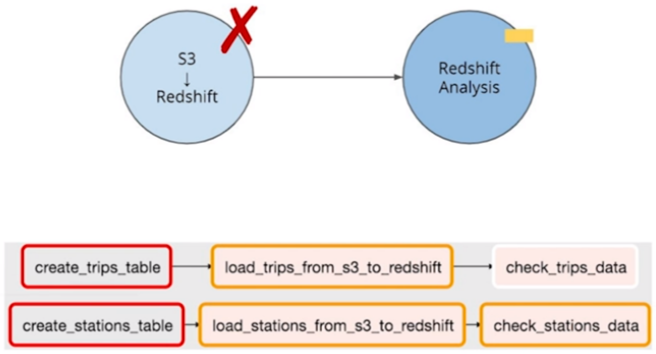
Parallelization can offer a significant speedup in the execution of Airflow DAGs.



Task Boundaries

Debugging errors in your DAGs is much simpler if your tasks perform a single task.

By simply looking at the Airflow UI you can pinpoint precisely what went wrong when something failed



Task Boundaries

DAG tasks should be designed such that they are:

- Atomic and have a single purpose
- Maximize parallelism
- Make failure states obvious

Every task in your dag should perform **only one job**.

> "Write programs that do one thing and do it well." - Ken Thompson's Unix Philosophy

Benefits of Task Boundaries

- Re-visitable: Task boundaries are useful for you if you revisit a pipeline you wrote after a 6 month absence. You'll have a much easier time understanding how it works and the lineage of the data if the boundaries between tasks are clear and well defined. This is true in the code itself, and within the Airflow UI.
- Tasks that do just one thing are often more easily parallelized. This parallelization can offer a significant speedup in the execution of our DAGs.

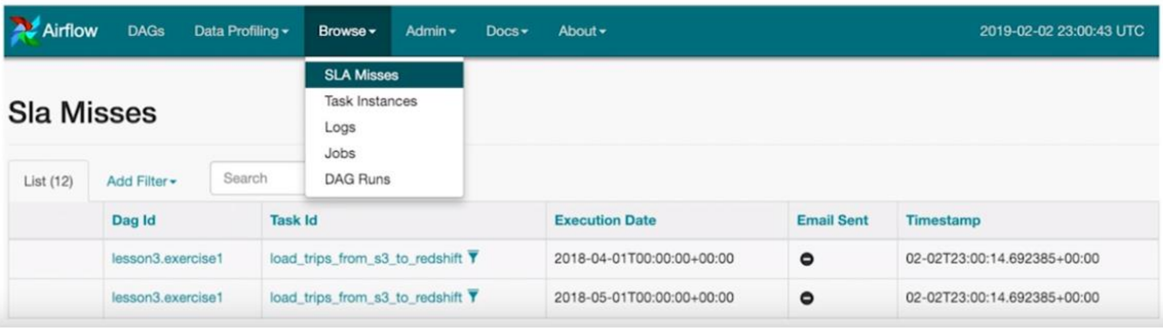
Monitoring:

Airflow can surface metrics and emails to help you stay on top of pipeline issues

Monitoring

DAGs can be configured to have an **SLA** (Service Level Agreement), which is defined as a time by which a DAG must complete.

Airflow can be configured to email a list of DAGs with missed SLAs. It also has a UI to view missed SLAs.

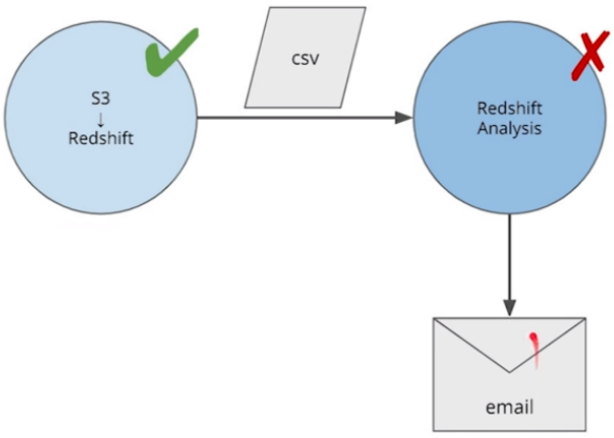


Sla Misses

	Dag Id	Task Id	Execution Date	Email Sent	Timestamp
	lesson3.exercise1	load_trips_from_s3_to_redshift	2018-04-01T00:00:00+00:00	⊖	02-02T23:00:14.692385+00:00
	lesson3.exercise1	load_trips_from_s3_to_redshift	2018-05-01T00:00:00+00:00	⊖	02-02T23:00:14.692385+00:00

Airflow can be configured to send **emails** on DAG and task state changes.

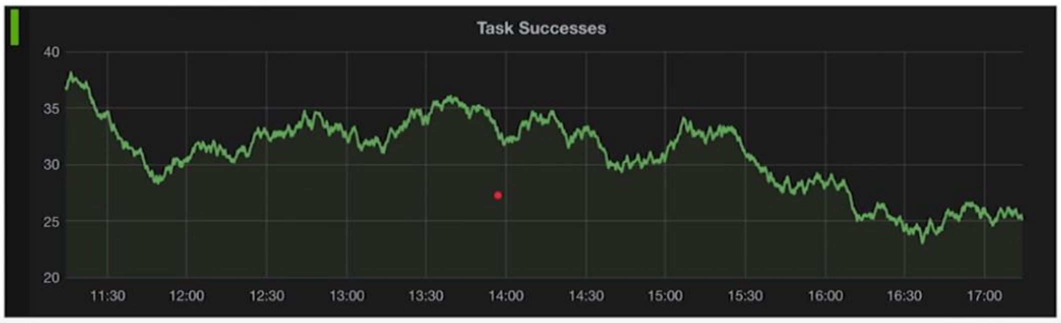
Failure emails can be used to trigger **alerts**.



Monitoring

Airflow can be configured to publish metrics to **statsd**.

Statsd and **Grafana** can be used together to view Airflow system metrics as well as trigger alerts for outages and failures



Data Validation

Definition: 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.

Data Validation in Action

In our bikesharing example, we could have added the following validation steps:

After loading from S3 to Redshift:

- Validate the number of rows in Redshift match the number of records in S3

Once location business analysis is complete:

- Validate that all locations have a daily visit average greater than 0
- Validate that the number of locations in our output table match the number of tables in the input table

Why is this important?

- Data Pipelines provide a set of logical guidelines and a common set of terminology.
- The conceptual framework of data pipelines will help you better organize and execute everyday data engineering tasks.