



PIPELINES IN PYTHON

Ordina



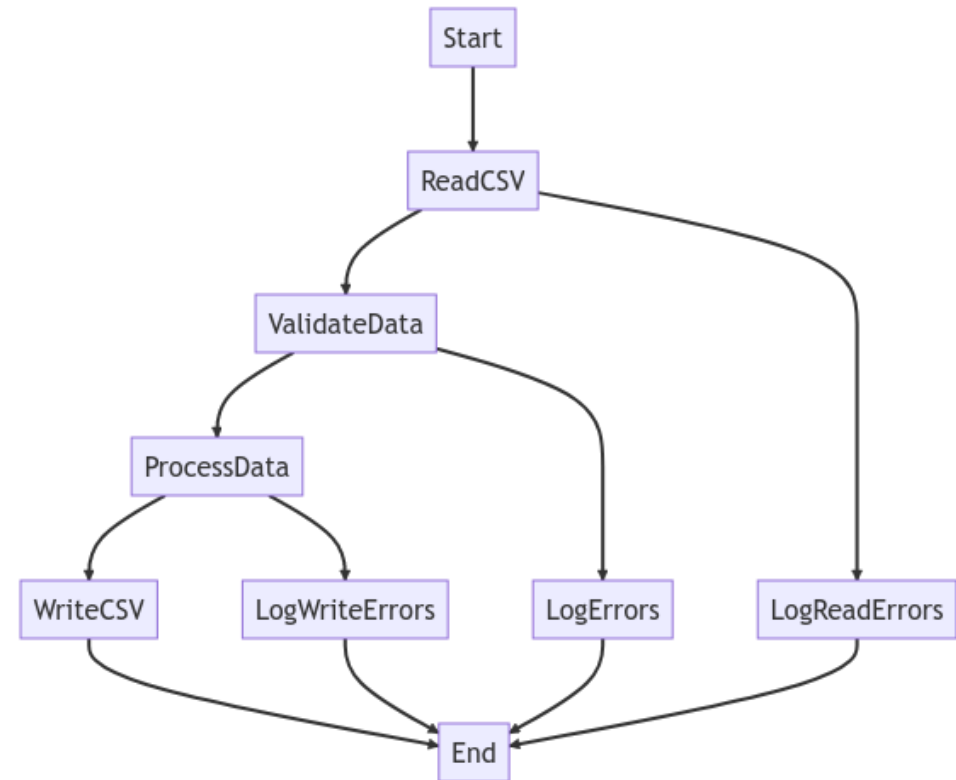
CONTENT

WHAT IS A DATA PIPELINE IN PYTHON?

A data pipeline with Python is a series of data processing steps that transform raw data into actionable insights. This includes the

- Collect,
- Clean up,
- Validate and
- Convert

of data to make it suitable for analysis and reporting. Data pipelines in Python can be simple and consist of a few steps - or they can be complex and include several steps and tools. Both are possible.



DIFFERENT PIPELINE TOOLKITS & FRAMEWORKS

Pipeline frameworks & libraries

- [ActionChain](#) - A workflow system for simple linear success/failure workflows.
- [Adage](#) - Small package to describe workflows that are not completely known at definition time.
- [AiiDA](#) - workflow manager with a strong focus on provenance, performance and extensibility.
- [Airflow](#) - Python-based workflow system created by AirBnb.
- [Anduril](#) - Component-based workflow framework for scientific data analysis.
- [Antha](#) - High-level language for biology.
- [AWE](#) - Workflow and resource management system with CWL support.
- [Balsam](#) - Python-based high throughput task and workflow engine.
- [Bds](#) - Scripting language for data pipelines.
- [BioMake](#) - GNU-Make-like utility for managing builds and complex workflows.
- [BioQueue](#) - Explicit framework with web monitoring and resource estimation.
- [Bioshake](#) - Haskell DSL built on shake with strong typing and EDAM support.
- [Bistro](#) - Library to build and execute typed scientific workflows.
- [Bpipe](#) - Tool for running and managing bioinformatics pipelines.
- [Briefly](#) - Python Meta-programming Library for Job Flow Control.
- [Cluster Flow](#) - Command-line tool which uses common cluster managers to run bioinformatics pipelines.
- [Clusterjob](#) - Automated reproducibility, and hassle-free submission of computational jobs to clusters.
- [Compi](#) - Application framework for portable computational pipelines.
- [Compss](#) - Programming model for distributed infrastructures.
- [Conan2](#) - Light-weight workflow management application.
- [Consecution](#) - A Python pipeline abstraction inspired by Apache Storm topologies.
- [Cosmos](#) - Python library for massively parallel workflows.
- [Coular](#) - Unified interface for constructing and managing workflows on different workflow engines, such as Argo

MAIN PIPELINE FRAMEWORKS

Python provides several frameworks for creating data pipelines, including Apache Airflow, Luigi, and Prefect. With these frameworks, you can easily create, schedule, and manage your data pipelines.

Apache Airflow: A powerful open source platform that allows you to create, plan and monitor workflows in Python.

Luigi: A Python module developed by Spotify that simplifies the construction of complex data pipelines.

Prefect: A modern data pipeline framework with a focus on simplicity, flexibility and scalability.

STEPS IN THE PIPELINE

5 steps for data processing :

Define the data sources: Identify where the data comes from and how it should be collected.

Clean and validate data: Use Python libraries such as Pandas and NumPy to clean, validate and prepare the data.

Transform and enrich data: Use Data transformations and enrichments to improve the quality of the data for analysis.

Store the processed data: Save the processed data in a suitable storage system, such as a database or a Cloud storage.

Analyze and visualize data: Use Python libraries such as Matplotlib, Seaborn and Plotly for Data visualization and analysis.

DATA PIPELINE AND ETL PIPELINE: THE DIFFERENCE

Often the terms Data Pipeline and ETL Pipeline (Extract-Transform-Load) are used synonymously - but this is wrong.

ETL pipelines represent a subcategory of data pipelines. 3 characteristics show this particularly clearly:

ETL pipelines follow a specific sequence. Here, the data is extracted, transformed and stored in a data repository. However, there are also other ways to design data pipelines. In particular, with the introduction of cloudnative tools, the circumstances have changed. In these cases, data is ingested first and then loaded into the cloud data warehouse. Only then are transformations performed.

ETL processes tend to involve batch processing but as already mentioned, the scope of application of data lines is more extensive. They can also integrate the processing of data streams.

Ultimately, although rather rare, **it is not mandatory that data pipelines as a whole system perform data transformations as in ETL pipelines.** Nevertheless, there is hardly any data pipeline that does not employ data transformations to facilitate the data analysis process.

EXTRACT-LOAD-TRANSFORM FOR THE DATA LAKE

In recent years, the ELT process as an alternative variant to the ETL process established.

In the ETL process, the data is first prepared, but this can lead to some information being lost. Originally, this process comes from the data warehousing area, where structured information is of great importance.

This contrasts with the ELT process, where data is first transferred to another infrastructure before being processed. This preserves as much of the original form and content as possible, which is especially important in the field of data science to train accurate machine learning models.

The ELT process is used primarily in the area of Big Data and Data Lakes, as unstructured data can also be processed effectively in this way. ETL and ELT are also generally referred to as "Data Ingestion", which includes data ingestion.

TIPS

Here are 4 helpful tips to improve your data pipeline:

1. Modularize your code: Break your pipeline into smaller, reusable components to make it easier to maintain and debug.
2. Use version control: Track changes to your pipeline code and data using tools such as Git and GitHub.
3. Automate testing: Implement automated Tests to ensure the accuracy and integrity of your data pipeline.
4. Monitor and log: Set up monitoring and logging systems to track the performance and health of your data pipeline.

PIPELINE LIBRARIES FOR DATA PROCESSING

Python provides a rich ecosystem of libraries for building data processing pipelines. Here are some important libraries for data manipulation and analysis in Python:

Pandas

A powerful library for data manipulation and analysis. With Pandas, data can be imported in various formats such as CSV, Excel or SQL tables and saved as data frames (DataFrame). Pandas also offers many functions for data manipulation such as filtering, grouping and aggregation.

NumPy

A library for numerical calculations in Python. NumPy offers a variety of functions for numerical calculations such as linear algebra, Fourier transformation and random number generation. NumPy is also the basis for many other libraries used in data science.

PIPELINE LIBRARIES FOR DATA PROCESSING

Dask

A parallel computing library for large-scale data processing. With Dask you can process large data sets in parallel on a cluster of computers. Dask also offers functions for storing and analyzing large data sets in distributed systems.

Scikit-learn

A library for machine learning and data mining in Python. Scikit-learn offers a variety of machine learning algorithms such as regression, classification, clustering and dimensionality reduction. Scikit-learn also offers functions for data modeling, evaluation and selection.

Extract, transform, load (ETL) is a common approach to creating data pipelines. Python is an excellent choice for creating ETL pipelines because of its extensive library support and ease of use. Some popular Python libraries for ETL are Pandas, SQLAlchemy, and PySpark.



CREATE YOUR FIRST ETL PIPELINE WITH PYTHON

APACHE AIRFLOW



Apache
Airflow

WHAT IS AIRFLOW?

Apache Airflow™ is an open-source platform for developing, scheduling, and monitoring batch-oriented workflows. Airflow's extensible Python framework enables you to build workflows connecting with virtually any technology. A web interface helps manage the state of your workflows. Airflow is deployable in many ways, varying from a single process on your laptop to a distributed setup to support even the biggest workflows.


WORKFLOWS AS CODE

The main characteristic of Airflow workflows is that all workflows are defined in Python code. “Workflows as code” serves several purposes:

Dynamic: Airflow pipelines are configured as Python code, allowing for dynamic pipeline generation.

Extensible: The Airflow™ framework contains operators to connect with numerous technologies. All Airflow components are extensible to easily adjust to your environment.

Flexible: Workflow parameterization is built-in leveraging the Jinja templating engine.



```
from datetime import datetime
```

```
from airflow import DAG
```

```
from airflow.decorators import task
```

```
from airflow.operators.bash import BashOperator
```

```
# A DAG represents a workflow, a collection of tasks
```

```
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
```

```
    # Tasks are represented as operators
```

```
    hello = BashOperator(task_id="hello", bash_command="echo hello")
```

```
    @task()
```

```
    def airflow():
```

```
        print("airflow")
```

```
    # Set dependencies between tasks
```

```
    hello >> airflow()
```




Here you see:

A DAG named “demo”, starting on Jan 1st 2022 and running once a day. A DAG is Airflow’s representation of a workflow.

Two tasks, a BashOperator running a Bash script and a Python function defined using the @task decorator

>> between the tasks defines a dependency and controls in which order the tasks will be executed

Airflow evaluates this script and executes the tasks at the set interval and in the defined order. The status of the “demo” DAG is visible in the web interface:

19/09/2023, 17:00:11

25

All Run Types

All Run States

Clear Filters

Auto-refresh

Press **shift** + **/** for Shortcuts

deferred

failed

queued

removed

restarting

running

scheduled

skipped

success

up_for_reschedule

up_for_retry

upstream_failed

no_status

DAG

tutorial /

Run

2023-09-19, 16:28:42 UTC

Clear

Mark state as...

Details

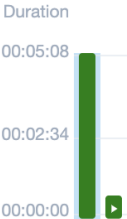
Graph

Gantt

Code

Layout:

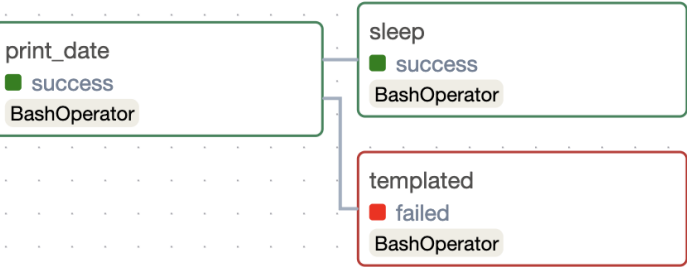
Left -> Right



print_date

sleep

templated



+

-



DAG: demo

Schedule: 0 0 ***

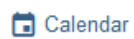
Next Run: 2022-05-29, 00:00:00



Grid



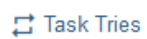
Graph



Calendar



Task Duration



Task Tries



Landing Times



Gantt



Details



Code



Audit Log



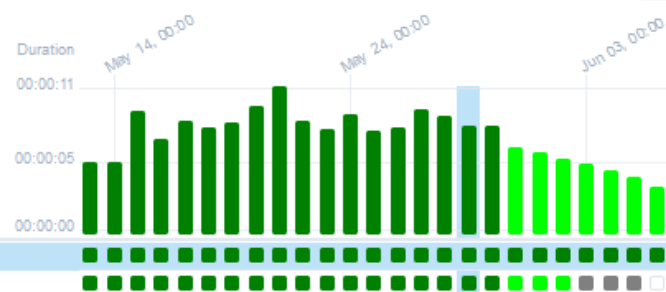
10 / 06 / 2022 , 10 : 12 : 28 AM

25

All Run Types

All Run States

Clear Filters

deferred failed queued running scheduled skipped success up_for_reschedule up_for_retry upstream_failed no_statusAuto-refresh ☒

DAG demo / Run 2022-05-30, 00:00:00 UTC / Task hello

[Task Instance Details](#) [Rendered Template](#) [Log](#) [XCom](#) [List Instances, all runs](#) [Filter Upstream](#)[Details](#) [Logs](#)

Task Actions

[Ignore All Deps](#) [Ignore Task State](#) [Ignore Task Deps](#)

Run

[Past](#) [Future](#) [Upstream](#) [Downstream](#) [Recursive](#) [Failed](#)

Clear

[Past](#) [Future](#) [Upstream](#) [Downstream](#)

Mark Failed

[Past](#) [Future](#) [Upstream](#) [Downstream](#)

Mark Success

Status ■ successTask ID hello [🔗](#)Run ID scheduled__2022-05-29T00:00:00+00:00 [🔗](#)

Operator BashOperator

Duration 00:00:01

Started 2022-10-06, 10:12:26 UTC

Ended 2022-10-06, 10:12:27 UTC

WHY AIRFLOW™ ?

Airflow™ is a batch workflow orchestration platform. The Airflow framework contains operators to connect with many technologies and is easily extensible to connect with a new technology. If your workflows have a clear start and end, and run at regular intervals, they can be programmed as an Airflow DAG.

If you prefer coding over clicking, Airflow is the tool for you. Workflows are defined as Python code which means:

- Workflows can be stored in version control so that you can roll back to previous versions
- Workflows can be developed by multiple people simultaneously
- Tests can be written to validate functionality
- Components are extensible and you can build on a wide collection of existing components

Rich scheduling and execution semantics enable you to easily define complex pipelines, running at regular intervals. Backfilling allows you to (re-)run pipelines on historical data after making changes to your logic. And the ability to rerun partial pipelines after resolving an error helps maximize efficiency.

Airflow's user interface provides:

- In-depth views of two things:
 - Pipelines
 - Tasks
- Overview of your pipelines over time

From the interface, you can inspect logs and manage tasks, for example retrying a task in case of failure.

The open-source nature of Airflow ensures you work on components developed, tested, and used by many other companies around the world. In the active community you can find plenty of helpful resources in the form of blog posts, articles, conferences, books, and more. You can connect with other peers via several channels such as Slack and mailing lists.

Airflow as a Platform is highly customizable. By utilizing Public Interface of Airflow you can extend and customize almost every aspect of Airflow

WHY NOT AIRFLOW™ ?

Airflow™ was built for finite batch workflows. While the CLI and REST API do allow triggering workflows, Airflow was not built for infinitely running event-based workflows. Airflow is not a streaming solution. However, a streaming system such as Apache Kafka is often seen working together with Apache Airflow. Kafka can be used for ingestion and processing in real-time, event data is written to a storage location, and Airflow periodically starts a workflow processing a batch of data.

If you prefer clicking over coding, Airflow is probably not the right solution. The web interface aims to make managing workflows as easy as possible and the Airflow framework is continuously improved to make the developer experience as smooth as possible. However, the philosophy of Airflow is to define workflows as code so coding will always be required.

If you don't have pip installed:

```
sudo apt-get update
```

```
sudo apt-get install python3-pip
```

c/user doesn't exist, you can use /home/username

To get airflow in the path folder:

```
sudo nano /etc/profile
```

```
export PATH="$MYPATH:$PATH"
```

^X, Y, enter

```
pip install Flask-Session==0.5.0
```

FUNDAMENTAL CONCEPTS

It's a DAG definition file

One thing to wrap your head around (it may not be very intuitive for everyone at first) is that this Airflow Python script is really just a configuration file specifying the DAG's structure as code. The actual tasks defined here will run in a different context from the context of this script. Different tasks run on different workers at different points in time, which means that this script cannot be used to cross communicate between tasks. Note that for this purpose we have a more advanced feature called XComs.

People sometimes think of the DAG definition file as a place where they can do some actual data processing - that is not the case at all! The script's purpose is to define a DAG object. It needs to evaluate quickly (seconds, not minutes) since the scheduler will execute it periodically to reflect the changes if any.

IMPORTING MODULES

An Airflow pipeline is just a Python script that happens to define an Airflow DAG object. Let's start by importing the libraries we will need.

```
import textwrap
from datetime import datetime, timedelta

# The DAG object; we'll need this to instantiate a DAG
from airflow.models.dag import DAG

# Operators; we need this to operate!
from airflow.operators.bash import BashOperator
```

DEFAULT ARGUMENTS

We're about to create a DAG and some tasks, and we have the choice to explicitly pass a set of arguments to each task's constructor (which would become redundant), or (better!) we can define a dictionary of default parameters that we can use when creating tasks.

Also, note that you could easily define different sets of arguments that would serve different purposes. An example of that would be to have different settings between a production and development environment

```
# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args={
    "depends_on_past": False,
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
    # 'wait_for_downstream': False,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function, # or list of functions
    # 'on_success_callback': some_other_function, # or list of functions
    # 'on_retry_callback': another_function, # or list of functions
    # 'sla_miss_callback': yet_another_function, # or list of functions
    # 'trigger_rule': 'all_success'
},
```

INSTANTIATE A DAG

```
with DAG(
    "tutorial",
    # These args will get passed on to each operator
    # You can override them on a per-task basis during operator initialization
    default_args={
        "depends_on_past": False,
        "email": ["airflow@example.com"],
        "email_on_failure": False,
        "email_on_retry": False,
        "retries": 1,
        "retry_delay": timedelta(minutes=5),
        # 'queue': 'bash_queue',
        # 'pool': 'backfill',
        # 'priority_weight': 10,
        # 'end_date': datetime(2016, 1, 1),
        # 'wait_for_downstream': False,
        # 'sla': timedelta(hours=2),
        # 'execution_timeout': timedelta(seconds=300),
        # 'on_failure_callback': some_function, # or list of functions
        # 'on_success_callback': some_other_function, # or list of functions
        # 'on_retry_callback': another_function, # or list of functions
        # 'sla_miss_callback': yet_another_function, # or list of functions
        # 'trigger_rule': 'all_success'
    },
    description="A simple tutorial DAG",
    schedule=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=["example"],
) as dag:
```

OPERATORS

An operator defines a unit of work for Airflow to complete. Using operators is the classic approach to defining work in Airflow. For some use cases, it's better to use the TaskFlow API to define work in a Pythonic context as described in [Working with TaskFlow](#). For now, using operators helps to visualize task dependencies in our DAG code.

All operators inherit from the `BaseOperator`, which includes all of the required arguments for running work in Airflow. From here, each operator includes unique arguments for the type of work it's completing. Some of the most popular operators are the `PythonOperator`, the `BashOperator`, and the `KubernetesPodOperator`.


Airflow completes work based on the arguments you pass to your operators. In this tutorial, we use the `BashOperator` to run a few bash scripts.

TASKS

To use an operator in a DAG, you have to instantiate it as a task. Tasks determine how to execute your operator's work within the context of a DAG.

In the following example, we instantiate the `BashOperator` as two separate tasks in order to run two separate bash scripts. The first argument for each instantiation, `task_id`, acts as a unique identifier for the task.

```
t1 = BashOperator(  
    task_id="print_date",  
    bash_command="date",  
)  
  
t2 = BashOperator(  
    task_id="sleep",  
    depends_on_past=False,  
    bash_command="sleep 5",  
    retries=3,  
)
```



Notice how we pass a mix of operator specific arguments (`bash_command`) and an argument common to all operators (`retries`) inherited from `BaseOperator` to the operator's constructor. This is simpler than passing every argument for every constructor call. Also, notice that in the second task we override the `retries` parameter with 3.

The precedence rules for a task are as follows:

1. Explicitly passed arguments
2. Values that exist in the `default_args` dictionary
3. The operator's default value, if one exists

A task must include or inherit the arguments `task_id` and `owner`, otherwise Airflow will raise an exception. A fresh install of Airflow will have a default value of 'airflow' set for `owner`, so you only really need to worry about ensuring `task_id` has a value

TEMPLATING WITH JINJA

Airflow leverages the power of Jinja Templating and provides the pipeline author with a set of built-in parameters and macros. Airflow also provides hooks for the pipeline author to define their own parameters, macros and templates.

This tutorial barely scratches the surface of what you can do with templating in Airflow, but the goal of this section is to let you know this feature exists, get you familiar with double curly brackets, and point to the most common template variable: `{{ ds }}` (today's "date stamp").

```
templated_command = textwrap.dedent(
    """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7)}}"
    {% endfor %}
    """
)

t3 = BashOperator(
    task_id="templated",
    depends_on_past=False,
    bash_command=templated_command,
)
```




Notice that the `templated_command` contains code logic in `{% %}` blocks, references parameters like `{{ ds }}`, and calls a function as in `{{ macros.ds_add(ds, 7) }}`.

Files can also be passed to the `bash_command` argument, like `bash_command='templated_command.sh'`, where the file location is relative to the directory containing the pipeline file (`tutorial.py` in this case). This may be desirable for many reasons, like separating your script's logic and pipeline code, allowing for proper code highlighting in files composed in different languages, and general flexibility in structuring pipelines. It is also possible to define your `template_searchpath` as pointing to any folder locations in the DAG constructor call.

Using that same DAG constructor call, it is possible to define `user_defined_macros` which allow you to specify your own variables. For example, passing `dict(foo='bar')` to this argument allows you to use `{{ foo }}` in your templates. Moreover, specifying `user_defined_filters` allows you to register your own filters. For example, passing `dict(hello=lambda name: 'Hello %s' % name)` to this argument allows you to use `{{ 'world' | hello }}` in your templates

ADDING DAG AND TASKS DOCUMENTATION

We can add documentation for DAG or each single task. DAG documentation only supports markdown so far, while task documentation supports plain text, markdown, reStructuredText, json, and yaml. The DAG documentation can be written as a doc string at the beginning of the DAG file (recommended), or anywhere else in the file. Below you can find some examples on how to implement task and DAG docs, as well as screenshots:

```
t1.doc_md = textwrap.dedent(
    """
    #### Task Documentation
    You can document your task using the attributes `doc_md` (markdown),
    `doc` (plain text), `doc_rst`, `doc_json`, `doc_yaml` which gets
    rendered in the UI's Task Instance Details page.
    ![img](http://montcs.bloomu.edu/~bobmon/Semesters/2012-01/491/import%20soul.png)
    **Image Credit:** Randall Munroe, [XKCD](https://xkcd.com/license.html)
    """
)

dag.doc_md = __doc__ # providing that you have a docstring at the beginning of the DAG; OR
dag.doc_md = """
This is a documentation placed anywhere
""" # otherwise, type it like this
```

SETTING UP DEPENDENCIES

We have tasks t1, t2 and t3 that do not depend on each other. Here's a few ways you can define dependencies between them

Note that when executing your script, Airflow will raise exceptions when it finds cycles in your DAG or when a dependency is referenced more than once

```
t1.set_downstream(t2)

# This means that t2 will depend on t1
# running successfully to run.
# It is equivalent to:
t2.set_upstream(t1)

# The bit shift operator can also be
# used to chain operations:
t1 >> t2

# And the upstream dependency with the
# bit shift operator:
t2 << t1

# Chaining multiple dependencies becomes
# concise with the bit shift operator:
t1 >> t2 >> t3

# A list of tasks can also be set as
# dependencies. These operations
# all have the same effect:
t1.set_downstream([t2, t3])
t1 >> [t2, t3]
[t2, t3] << t1
```

USING TIME ZONES

Creating a time zone aware DAG is quite simple. Just make sure to supply a time zone aware dates using pendulum. Don't try to use standard library timezone as they are known to have limitations and we deliberately disallow using them in DAGs.

TESTING

Time to run some tests. First, let's make sure the pipeline is parsed successfully.

Let's assume we are saving the code from the previous step in `tutorial.py` in the DAGs folder referenced in your `airflow.cfg`. The default location for your DAGs is `~/airflow/dags`.

```
python ~/airflow/dags/tutorial.py
```

If the script does not raise an exception it means that you have not done anything horribly wrong, and that your Airflow environment is somewhat sound

COMMAND LINE METADATA VALIDATION

Let's run a few commands to validate this script further.

```
# initialize the database tables  
airflow db migrate
```

```
# print the list of active DAGs  
airflow dags list
```

```
# prints the list of tasks in the "tutorial" DAG  
airflow tasks list tutorial
```

```
# prints the hierarchy of tasks in the "tutorial" DAG  
airflow tasks list tutorial --tree
```

TESTING

Let's test by running the actual task instances for a specific date. The date specified in this context is called the logical date (also called execution date for historical reasons), which simulates the scheduler running your task or DAG for a specific date and time, even though it physically will run now (or as soon as its dependencies are met).

We said the scheduler runs your task for a specific date and time, not at. This is because each run of a DAG conceptually represents not a specific date and time, but an interval between two times, called a data interval. A DAG run's logical date is the start of its data interval.

TESTING

```
# command layout: command subcommand [dag_id] [task_id] [(optional) date]  
  
# testing print_date  
airflow tasks test tutorial print_date 2015-06-01  
  
# testing sleep  
airflow tasks test tutorial sleep 2015-06-01
```

Now remember what we did with templating earlier? See how this template gets rendered and executed by running this command:

```
# testing templated  
airflow tasks test tutorial templated 2015-06-01
```

This should result in displaying a verbose log of events and ultimately running your bash command and printing the result.

TESTING

Note that the `airflow tasks test` command runs task instances locally, outputs their log to stdout (on screen), does not bother with dependencies, and does not communicate state (running, success, failed, ...) to the database. It simply allows testing a single task instance.

The same applies to `airflow dags test`, but on a DAG level. It performs a single DAG run of the given DAG id. While it does take task dependencies into account, no state is registered in the database. It is convenient for locally testing a full run of your DAG, given that e.g. if one of your tasks expects data at some location, it is available

BACKFILL

Everything looks like it's running fine so let's run a backfill. backfill will respect your dependencies, emit logs into files and talk to the database to record status. If you do have a webserver up, you will be able to track the progress. airflow webserver will start a web server if you are interested in tracking the progress visually as your backfill progresses.

Note that if you use `depends_on_past=True`, individual task instances will depend on the success of their previous task instance (that is, previous according to the logical date). Task instances with their logical dates equal to `start_date` will disregard this dependency because there would be no past task instances created for them.

You may also want to consider `wait_for_downstream=True` when using `depends_on_past=True`. While `depends_on_past=True` causes a task instance to depend on the success of its previous task instance, `wait_for_downstream=True` will cause a task instance to also wait for all task instances immediately downstream of the previous task instance to succeed.

BACKFILL

The date range in this context is a `start_date` and optionally an `end_date`, which are used to populate the run schedule with task instances from this DAG.

```
# optional, start a web server in debug mode in the background  
# airflow webserver --debug &  
  
# start your backfill on a date range  
airflow dags backfill tutorial \  
    --start-date 2015-06-01 \  
    --end-date 2015-06-07
```

TASKFLOW API

What is the difference between DAGs written using the traditional paradigm and those written using the TaskFlow API in Apache Airflow?

TASKFLOW VS CLASSIC

In Apache Airflow, a Directed Acyclic Graph (DAG) is a collection of tasks that run with a specified dependency structure. The traditional way of writing these DAGs involves defining each task and its dependencies separately.

For example, a simple traditional DAG could look like this:

```
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from datetime import datetime

with DAG('my_dag', start_date=datetime(2022, 1, 1)) as dag:
    task1 = DummyOperator(task_id='task1')
    task2 = DummyOperator(task_id='task2')
    task3 = DummyOperator(task_id='task3')

    task1 >> task2 >> task3
```

In this example, each task is an instance of `DummyOperator` and dependencies are defined using the bitshift operators.

TASKFLOW VS CLASSIC

On the other hand, the TaskFlow API, introduced in Airflow 2.0, allows for a more pythonic way of defining tasks and their dependencies using python functions. Each function represents a task and the dependencies are defined by the function calls.

Here's how the same DAG would look using the TaskFlow API:

```
from airflow.decorators import dag, task
from datetime import datetime

@dag(start_date=datetime(2022, 1, 1))
def my_dag():
    @task
    def task1():
        print('Running task 1')

    @task
    def task2():
        print('Running task 2')

    @task
    def task3():
        print('Running task 3')

    task1() >> task2() >> task3()

my_dag = my_dag()
```

TASKFLOW VS CLASSIC

In this example, each task is a python function decorated with the `@task` decorator and dependencies are defined by the function calls.

The TaskFlow API provides several advantages over the traditional paradigm:

Simplicity: Tasks and dependencies are defined using standard python functions and function calls, making the code easier to write and understand.

Flexibility: Since tasks are just python functions, they can accept arguments and return values, allowing for more complex workflows.

Integration with XCom: The TaskFlow API automatically handles pushing and pulling of XComs (cross-communication), which are used to share data between tasks.

THE 6 STEPS OF ETL PROCESS USING AIRFLOW WITH EXAMPLE AND EXERCISE

<https://tegardp.medium.com/the-6-step-etl-process-using-airflow-with-example-and-exercise-db46715a61f0>

About The Project

Prerequisite

- Basic knowledge of Python, installing packages, and virtual environment.
- Basic knowledge of Airflow

Project Overview

- Perform ETL from multiple data sources and store it to a single data source.
- You can go to my GitHub profile to see the source code and the final project.