Airflow tutorial Documentation

Tania Allard

Jun 10, 2019

Table of Contents

This tutorial was originally developed for PyCon US 2019.

CHAPTER 1

Setup

This section will guide you through the pre requisites for the workshop. Please make sure to install the libraries before the workshop as the conference WiFi can get quite slow when having too many people downloading and installing things at the same time.

Make sure to follow all the steps as detailed here especially *PyCon attendees* as there are specific details for the PyCon setup that needs to be done in advance.

1.1 Python 3.x

3.7 Preferred

We will be using Python. Installing all of Python's packages individually can be a bit difficult, so we recommend using Anaconda which provides a variety of useful packages/tools.

To download Anaconda, follow the link https://www.anaconda.com/download/ and select Python 3. Following the download, run the installer as per usual on your machine.

If you prefer not using Anaconda then this tutorial can help you with the installation and setup.

If you already have Python installed but not via Anaconda do not worry. Make sure to have either `venv` or `pipenv` installed. Then follow the instructions to set your virtual environment further down.

1.2 Git

Git is a version control software that records changes to a file or set of files. Git is especially helpful for software developers as it allows changes to be tracked (including who and when) when working on a project.

To download Git, go to the following link and choose the correct version for your operating system: https://git-scm.com/downloads.

1.2.1 Windows

Download the git for Windows installer . Make sure to select “use Git from the Windows command prompt” this will ensure that Git is permanently added to your PATH.

Also select “Checkout Windows-style, commit Unix-style line endings” selected and click on “Next”.

This will provide you both git and git bash. We will use the command line quite a lot during the workshop so using git bash is a good option.

1.3 GitHub

GitHub is a web-based service for version control using Git. You will need to set up an account at https://github.com. Basic GitHub accounts are free and you can now also have private repositories.

1.4 Text Editors/IDEs

Text editors are tools with powerful features designed to optimize writing code. There are several text editors that you can choose from. Here are some we recommend:

- VS code: this is your facilitator’s favourite and it is worth trying if you have not checked it yet
- Pycharm
- Atom

We suggest trying several editors before settling on one.

If you decide to go for VSCode make sure to also have the Python extension installed. This will make your life so much easier (and it comes with a lot of nifty features).

1.5 Microsoft Azure

You will need to get an Azure account as we will be using this to deploy the Airflow instance.

Note: If you are doing this tutorial live at PyCon US then your facilitator will provide you with specific instructions to set up your Azure subscription. If you have not received these please let your facilitator know ASAP.

Follow this link to get an Azure free subscription. This will give you 150 dollars in credit so you can get started getting things up and experimenting with Azure and Airflow.

1.6 MySQL

MySQL is one of the most popular databases used/ We need MySQL to follow along with the tutorial. Make sure to install it beforehand.

1.6.1 Mac users

Warning: There are some known issues with MySQL in Mac so we recommend using this approach to install and set MySQL up: https://gist.github.com/nrollr/3f57fc15ded7dddddcc4e82fe137b58e.

Also, note that you will need to make sure that OpenSSL is on your path to make sure this is added accordingly: If using `zsh`:

```
echo 'export PATH="/usr/local/opt/OpenSSL/bin:$PATH"' >> ~/.zshrc
```

If using `bash`:

```
echo 'export PATH="/usr/local/opt/openssl/bin:$PATH"' >> ~/.bashrc
```

make sure to reload using `source ~/.bashrc` or `source ~/.zshrc`

Troubleshooting

Later on, during the setup,, you will be installing `mysqlclient`. If during the process you get compilation errors try the following:

```
env LDFLAGS="-I/usr/local/opt/openssl/include -L/usr/local/opt/openssl/lib" pip␣
↪install mysqlclient
```

if you want to be safe before installing the library we recommend you set the following env variables:

```
export LDFLAGS="-L/usr/local/opt/openssl/lib"
export CPPFLAGS="-I/usr/local/opt/openssl/include"
```

1.6.2 Windows users

Download and install MySQL from the official website https://dev.mysql.com/downloads/installer/ and execute it. For additional configuration and pre-requisites make sure to visit the official MySQL docs.

1.6.3 Linux users

You can install the Python and MySQL headers and libraries like so:

Debian/Ubuntu:

```
sudo apt-get install python3-dev default-libmysqlclient-dev
```

Red Hat / Centos

```
sudo yum install python3-devel mysql-devel
```

After installation you need to start the service with:

```
systemctl start mysql
```

To ensure that the database launches after a reboot:

```
systemctl enable mysql
```

You should now be able to start the mysql shell through `/usr/bin/mysql -u root -p` you will be asked for the password you set during installation.

1.7 Creating a virtual environment

You will need to create a virtual environment to make sure that you have the right packages and setup needed to follow along the tutorial. Follow the instructions that best suit your installation.

1.7.1 Anaconda

If you are using Anaconda first you will need to make a directory for the tutorial, for example `mkdir airflow-tutorial`. Once created make sure to change into it using `cd airflow-tutorial`.

Next, make a copy of this environment.yaml and install the

dependencies via `conda env create -f environment.yml`. Once all the dependencies are installed you can activate your environment through the following commands

```
source activate airflow-env # Mac
activate airflow-env        # Windows and Linux
```

To exit the environment you can use

```
deactivate airflow-env
```

1.7.2 pipenv

Create a directory for the tutorial, for example:

```
mkdir airflow-tutorial
```

and change your working directory to this newly created one `cd airflow-tutorial`.

Once then make a copy of this Pipfile in your new directory and install via `pipenv install`. This will install the dependencies you need. This might take a while so you can make yourself a brew in the meantime.

Once all the dependencies are installed you can run `pipenv shell` which will start a session with the correct virtual environment activated. To exit the shell session using `exit`.

1.7.3 virtualenv

Create a directory for the tutorial, for example :

```
mkdir airflow-tutorial
```

and change directories into it (`cd airflow-tutorial`). Now you need to run venv

```
python3 -m venv env/airflow # Mac and Linux
python -m venv env/airflow  # Windows
```

this will create a virtual Python environment in the `env/airflow` folder. Before installing the required packages you need to activate your virtual environment:

```
source env/bin/activate # Mac and Linux
.\env\Scripts\activate  # Windows
```

Make a copy of this requirements file in your new directory. Now you can install the packages using via pip `pip install -r requirements.txt`

To leave the virtual environment run `deactivate`

1.8 Twitter and twitter developer account

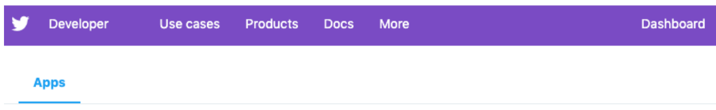
This tutorial uses the Twitter API for some examples and to build some of the pipelines included.

Please make sure to follow the next steps to get you all set up.

1. Create an account at https://twitter.com/.
2. **Next, you will need to apply for a developer account, head to https://developer.twitter.com/en/apply.** You will need to provide detailed information about what you want to use the API for. Make sure to complete all the steps and confirm your email address so that you can be notified about the status of your application.

   Warning: Before completing the application read the PyCon attendees section below *Twitter developer app*

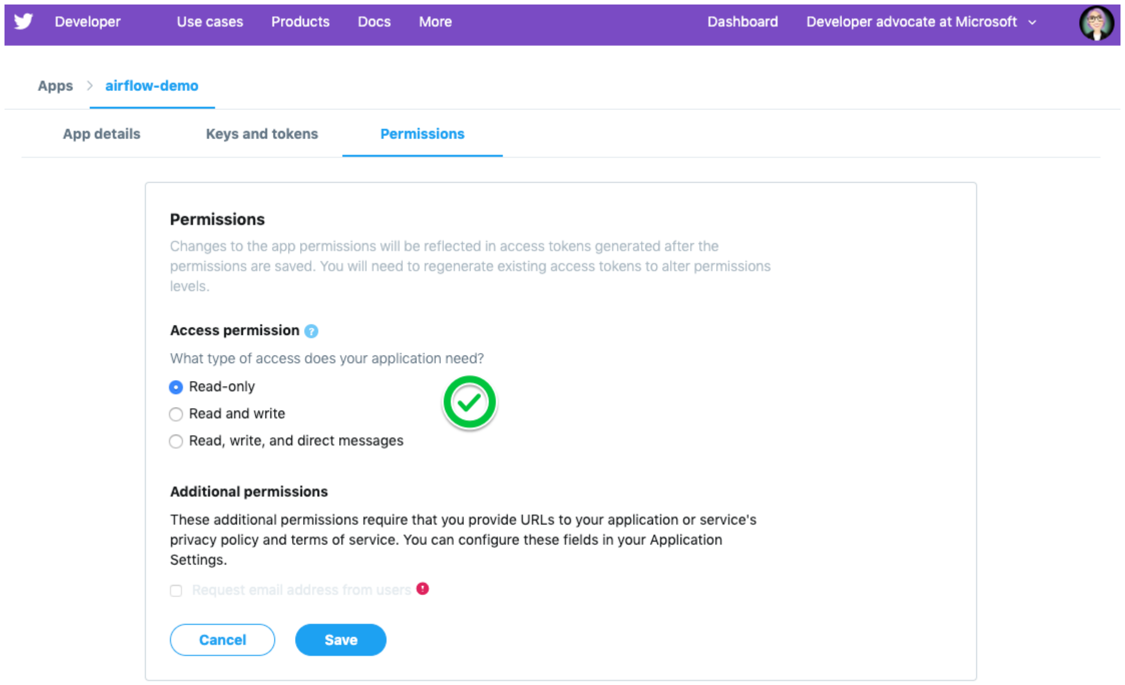
3. Once your application has been approved you will need to go to https://developer.twitter.com/en/apps login with your details (they should be the same as your Twitter account ones).



4. **On your app dashboard click on the create an app button**

   Make sure to give it a descriptive name, something like `airflow-tutorial` or the such

5. Once you complete the details and create your new app you should be able to access it via the main app dashboard. Click on details button next to the app name and head over to permissions. We only need read permissions for the tutorial, so these should look something like this



Permissions

Changes to the app permissions will be reflected in access tokens generated after the permissions are saved. You will need to regenerate existing access tokens to alter permissions levels.

Access permission

What type of access does your application need?

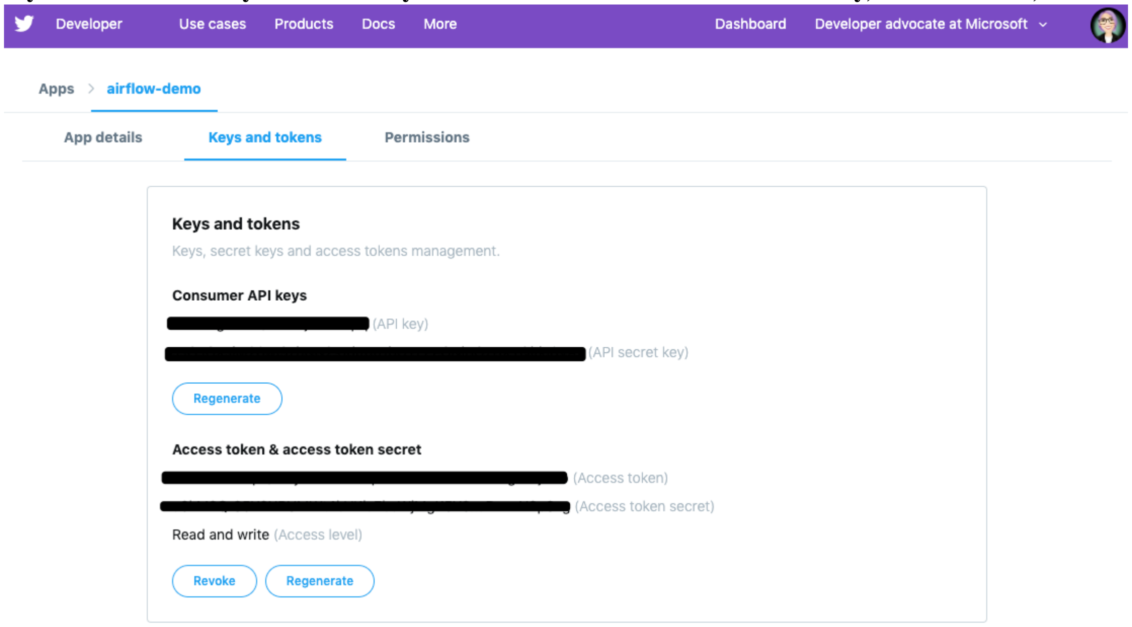
[✓ Read-only]
Read and write
Read, write, and direct messages

Additional permissions

These additional permissions require that you provide URLs to your application or service's privacy policy and terms of service. You can configure these fields in your Application Settings.

Request email address from users

6. **Now if you click on the Keys and tokens you will be able to see a set of an API key, an API secret, an Access token, and an**



Keys and tokens

Keys, secret keys and access tokens management.

Consumer API keys

[redacted] (API key)
[redacted] (API secret key)

Access token & access token secret

[redacted] (Access token)
[redacted] (Access token secret)

Read and write (Access level)

They are only valid for the permissions you specified before. Keep a record of these in a safe place as we will need them for the Airflow pipelines.

1.9 Docker

We are going to use Docker for some bits of the tutorial (this will make it easier to have a local Airflow instance).

Follow the instructions at https://docs.docker.com/v17.12/install/ make sure to read the pre-requisites quite carefully before starting the installation.

1.9.1 PyCon attendees

Twitter developer app

The Twitter team will be expediting your applications to make sure you are all set up for the day .

When filling in your application make sure to add the following details (as written here) to make sure this is processed.

In the what are you planning to use the developer account for:

```
This account is to be used for the Airflow tutorial at PyCon US 2019 lead by Tania␣
↪Allard.
We will be using the Twitter API to collect tweets, setting a database and create ETL␣
↪pipelines as part of the tutorial.
This will be integrated into Airflow and no personally identifiable data will be used␣
↪in the process.
We will not be conducting text analysis, user details analysis or any sort of␣
↪surveillance process as part of the tutorial.
```

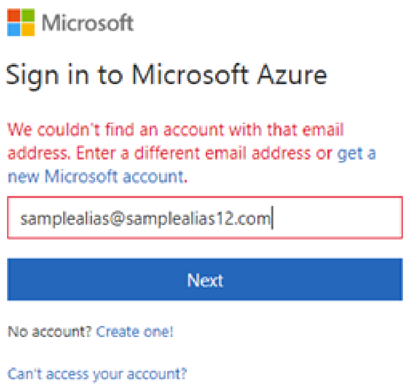
Azure Pass account

As a PyCon attendee, you will be issued with an Azure pass worth 200 dollars with a 90 days validity. You will not need to add credit card details to activate but you will need to follow this process to redeem your credits.

1. Send an email your facilitator at trallard@bitsandchips.me with the subject line `Airflow PyCon- Azure Pass`, they will send you an email with a *unique* code to redeem. Please do not share with anyone, this is a single-use pass and once activated it will be invalid.

2. Go to this site to redeem your pass. We recommend doing this in a private/incognito window. You can then click start and attach your new pass to your existing account.

If you see the following error (see image)


Microsoft

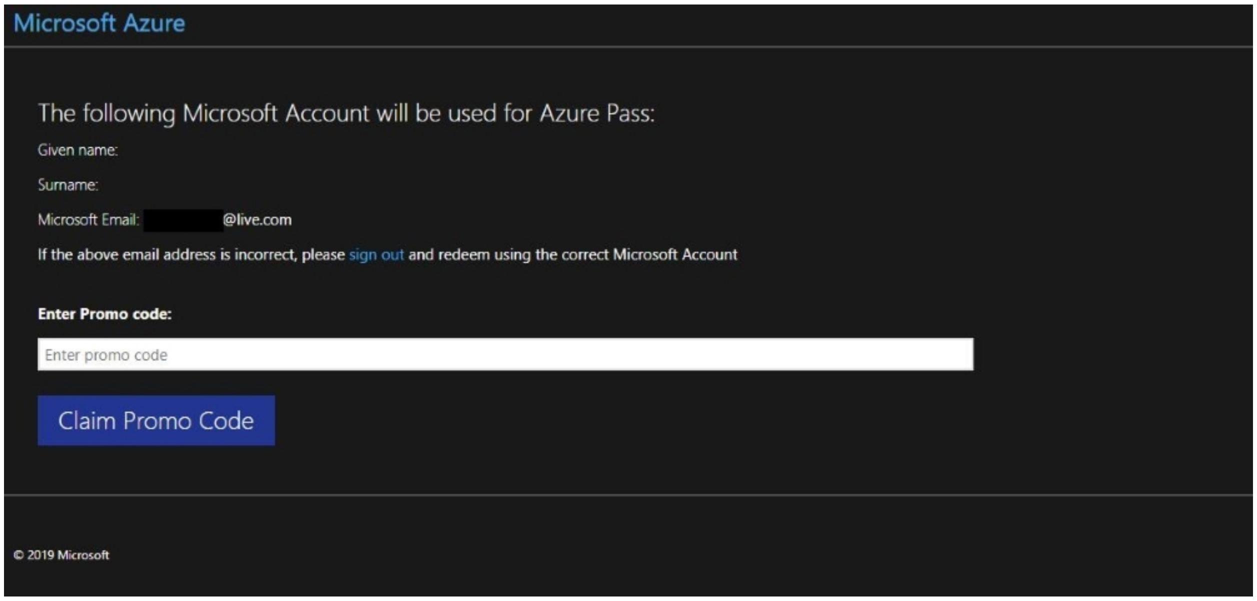
Sign in to Microsoft Azure

We couldn't find an account with that email address. Enter a different email address or get a new Microsoft account.

samplealias@samplealias12.com

you can go to this site to register the email and proceed.

4. Confirm your email address. You will then be asked to add the promo code that you were sent by your instructor. Do not close or refresh the window until you have received a confirmation that this has been successful.



Microsoft Azure

The following Microsoft Account will be used for Azure Pass:

Given name:

Surname:

Microsoft Email: [redacted]@live.com

If the above email address is incorrect, please sign out and redeem using the correct Microsoft Account

Enter Promo code:

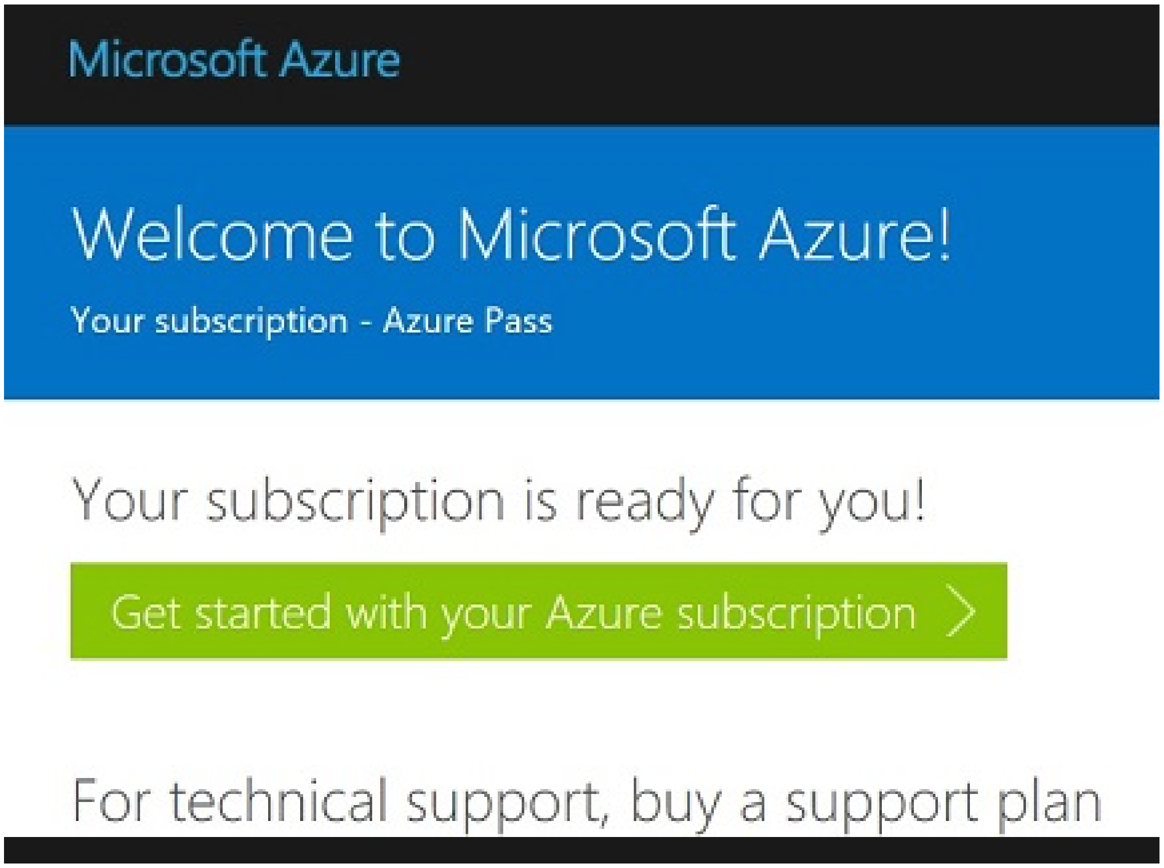
Enter promo code

Claim Promo Code

© 2019 Microsoft

5. Activate your subscription: click on the activate button and fill in the personal details

Again once completed, do not refresh the window until you see this image



Microsoft Azure

Welcome to Microsoft Azure!

Your subscription - Azure Pass

Your subscription is ready for you!

Get started with your Azure subscription >

For technical support, buy a support plan

At this point, your subscription will be ready, click on Get started to go to your Azure portal

CHAPTER 2

About the workshop

We will be taking a look at the basic concepts of data pipelines as well as practical use cases using Python.

2.1 About you:

- Some experience using the command line
- Intermediate Python knowledge / use
- Be able to apply what we learn and adopt to your use cases
- Interested in data and systems
- Aspring or current data engineering
- Some knowledge about systems and databases (enough to be dangerous)

2.2 Our focus for the day

- Greater understanding on how to apply data pipelines using the Python toolset
- Focus on concepts
- Apply knowledge with each library
- Will give you the building blocks

2.3 Keeping on track

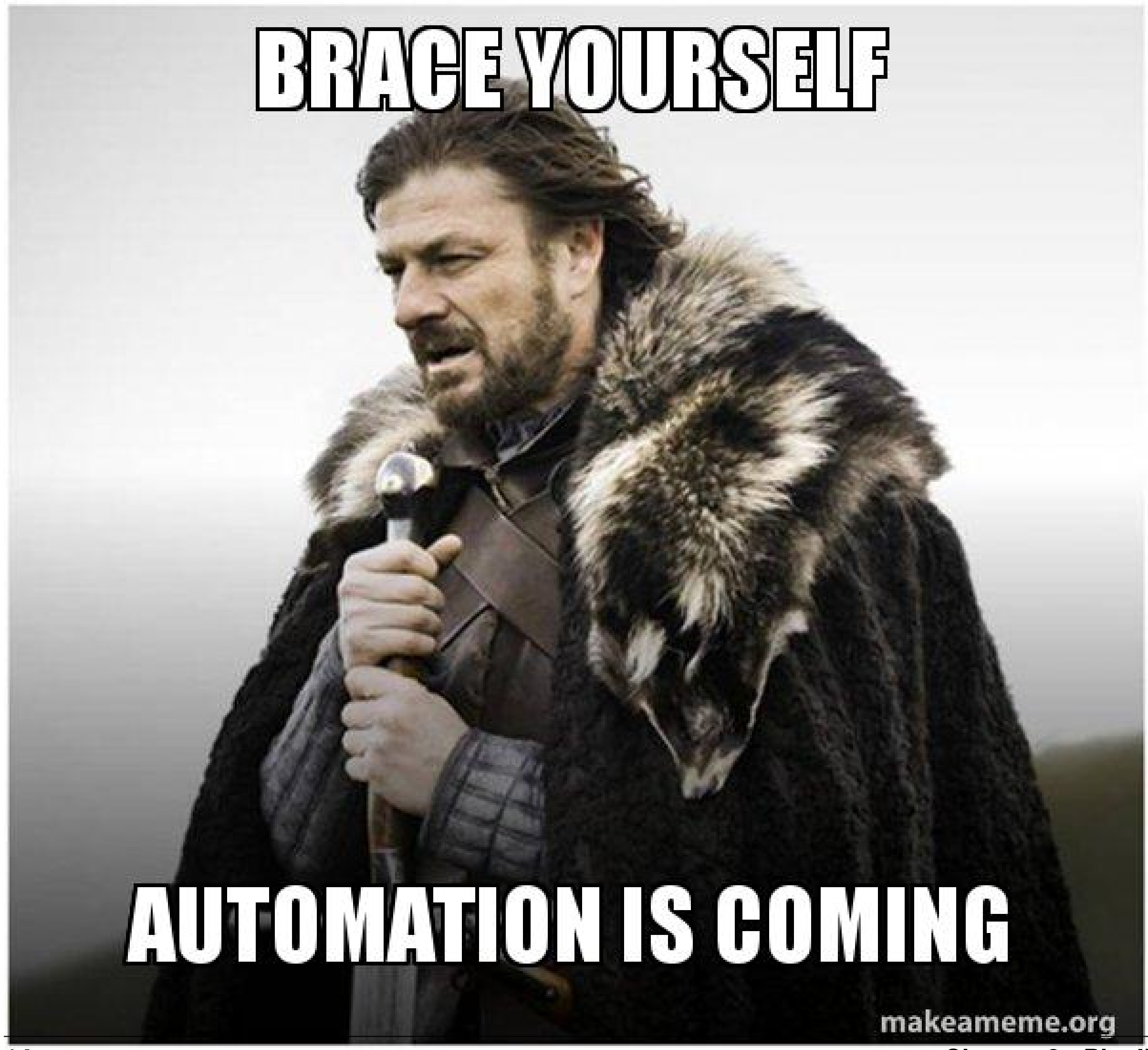
You will find across the tutorial examples. We will use this to identify how folks are doing over the workshop (if following along in person). Place the post it as follows:

Purple postit: all good, task has been completed

Orange postit: I need extra time or need help with the task in hand

CHAPTER 3

Pipelines



Automation helps us speed those manual boring tasks. The ability to automate means you can spend time working on other more thought-intensive projects.

Automation adds monitoring and logging tasks:

| **Easy to automate** | **Difficult to automate** | | —————————————— | —————————————— | | Regularly scheduled reports | One-off or non-scheduled tasks | | Clear success/failure outcomes | Unclear success/failure outcomes | | Input that can be handled via machines | Requires deeper human input |

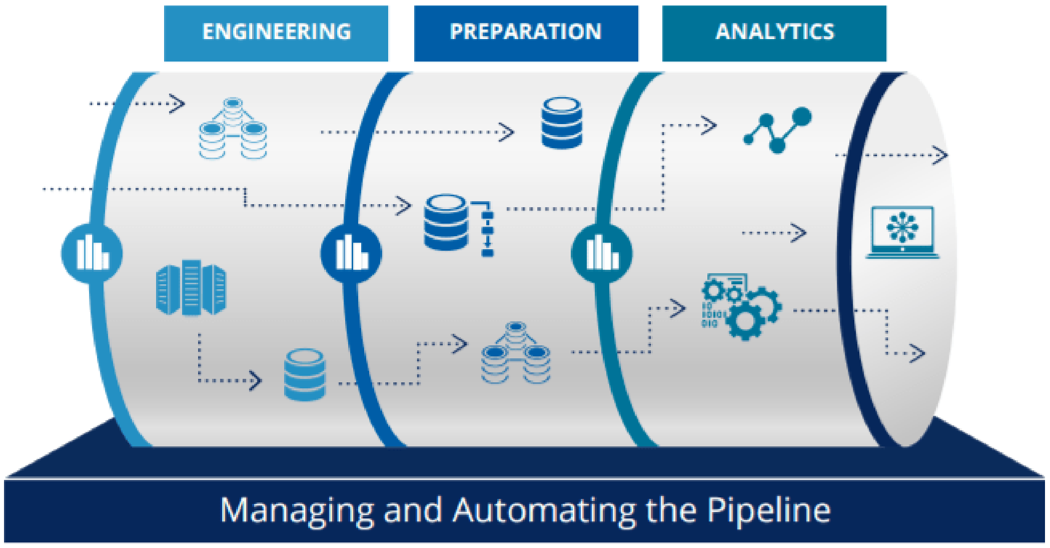
3.1 Steps to automation

Whenever you consider automating a task ask the following questions:

- When should this task begin?
- Does this task have a time limit?
- What are the inputs for this task?
- What is success or failure within this task? (How can we clearly identify the outcomes?)
- If the task fails what should happen?
- What does the task provide or produce? In what way? To whom?
- What (if anything) should happen after the task concludes?

3.2 What is a data pipeline?

Roughly this is how all pipelines look like:



they consist mainly of three distinct parts: data engineering processes, data preparation, and analytics. The upstream steps and quality of data determine in great measure the performance and quality of the subsequent steps.

3.3 Why do pipelines matter?

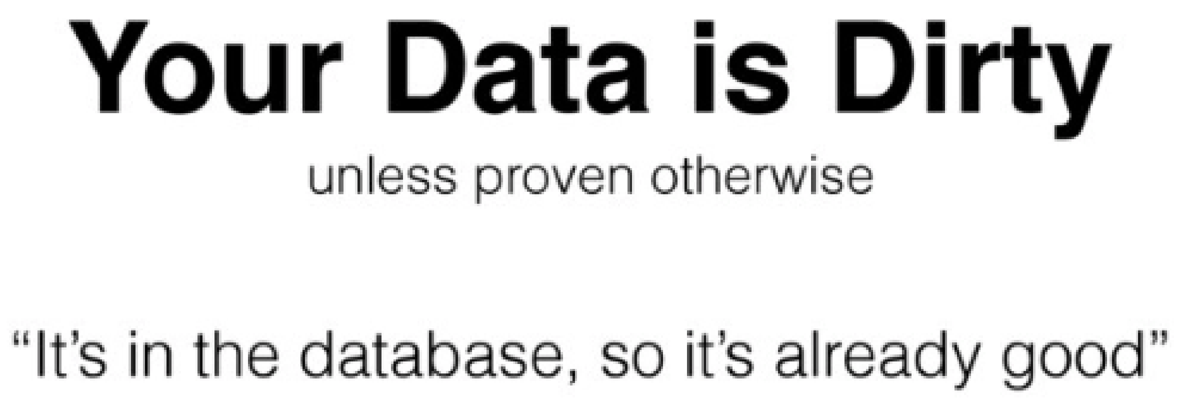
- Analytics and batch processing is mission-critical as they power all data-intensive applications
- The complexity of the data sources and demands increase every day
- A lot of time is invested in writing, monitoring jobs, and troubleshooting issues.

This makes data engineering one of the most critical foundations of the whole analytics cycle.

3.3.1 Good data pipelines are:

- Reproducible: same code, same data, same environment -> same outcome
- Easy to productise: need minimal modifications from R&D to production
- Atomic: broken into smaller well-defined tasks

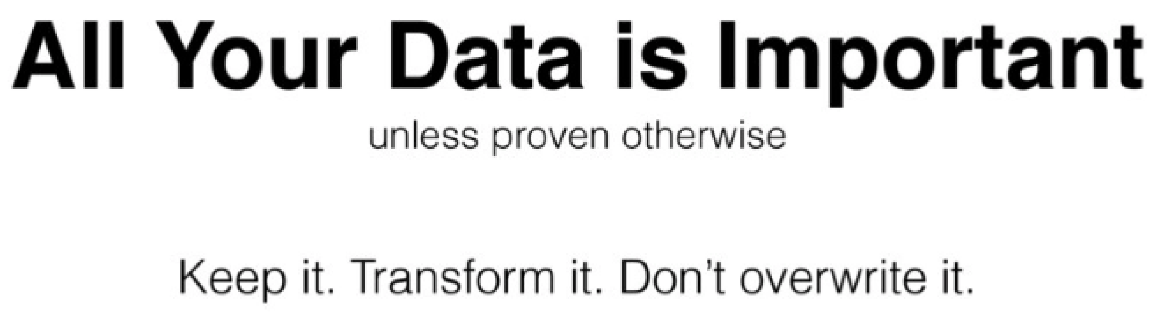
When working with data pipelines always remember these two statements:



Your Data is Dirty

unless proven otherwise

“It’s in the database, so it’s already good”



All Your Data is Important

unless proven otherwise

Keep it. Transform it. Don’t overwrite it.

As your data engineering and data quality demands increase so does the complexity of the processes. So more often than not you will eventually need a workflow manager to help you with the orchestration of such processes.

GNU Make + Unix pipes + Steroids

3.4 Creating your first ETL pipeline in Python

3.4.1 Setting your local database

Let's us access your local MySQL from the command line, type the following on the command line:

```
MySQL -u root -p
```

followed by your MySQL password (this was done as part of your setup)

you should see the following message as well as a change in your prompt:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 276
Server version: 8.0.15 Homebrew

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

To see the existing databases you can use the following command:

```
SHOW DATABASES
```

you can also see the users and the relevant hosts using the following command

```
SELECT user, host FROM mysql.user;
```

We will need to create a new database for the following sections. So let's start by creating a new database called `airflowdb`:

```
 CREATE DATABASE airflowdb CHARACTER SET utf8 COLLATE utf8_unicode_ci;
```

as well as creating a corresponding user:

```
CREATE USER 'airflow'@'localhost' IDENTIFIED BY 'password';
```

make sure to substitute `password` with an actual password. For this tutorial let's assume the password is `python2019`.

Now we need to make sure that the `airflow` user has access to the databases:

```
GRANT ALL PRIVILEGES ON *.* TO 'airflow'@'localhost';
FLUSH PRIVILEGES;
```

If you want to restrict the access of this user to the `airflowdb` database, for example, you can do it via:

```
GRANT ALL PRIVILEGES ON airflowdb.* To 'airflow'@'localhost';
FLUSH PRIVILEGES;
```

so now the output of `SELECT user, host FROM mysql.user;` should look like this:

```
mysql> select user, host FROM mysql.user;
+------------------+-----------+
| user             | host      |
+------------------+-----------+
| airflow          | localhost |
| mysql.infoschema | localhost |
| mysql.session    | localhost |
| mysql.sys        | localhost |
| root             | localhost |
+------------------+-----------+
5 rows in set (0.00 sec)
```

if you need to remove a user you can use the following command:

```
DROP USER '<username>'@'localhost' ;
```

3.4.2 Checking connection to the database from Python

The following snippet will allow you to connect to the created database from Python:

Note that you need the database name, password, and user for this.

```
# Script to check the connection to the database we created earlier airflowdb

# importing the connector from mysqlclient
import mysql.connector as mysql

# connecting to the database using the connect() method
# it takes 3 parameters: user, host, and password

dbconnect = mysql.connect(host="localhost",
                          user="airflow",
                          password="python2019",
                          db="airflowdb")

# print the connection object
print(dbconnect)

# do not forget to close the connection
dbconnect.close()
```

`dbconnect` is a connection object which can be used to execute queries, commit transactions and rollback transactions before closing the connection. We will use it later.

The `dbconnect.close()` method is used to close the connection to database. To perform further transactions, we need to create a new connection.

3.4.3 Streaming Twitter data into the database

We are going to create a Python script that helps us to achieve the following:

1. Create a class to connect to the Twitter API
2. Connect our database and reads the data into the correct columns

We will be using the Tweepy library for this (docs here https://tweepy.readthedocs.io/en/latest/) .

Let's start with an example to collect some Tweets from your public timeline (for details on the Tweet object visit the API docs)

The first step will be to create a config file (`config.cfg`) with your Twitter API tokens.

```
[twitter]
consumer_key = xxxxxxxxxxxxxxxxxx
consumer_secret = xxxxxxxxxxxxxxxxxx
access_token = xxxxxxxxxxxxxxxxxx
access_token_secret = xxxxxxxxxxxxxxxxxx
```

Now to the coding bits:

```
# Import libraries needed
from configparser import ConfigParser
from pathlib import Path

import tweepy

# Path to the config file with the keys make sure not to commit this file
CONFIG_FILE = Path.cwd() / "config.cfg"

config = ConfigParser()
config.read(CONFIG_FILE)

# Authenticate to Twitter
auth = tweepy.OAuthHandler(
    config.get("twitter", "consumer_key"), config.get("twitter", "consumer_secret")
)
auth.set_access_token(
    config.get("twitter", "access_token"), config.get("twitter", "access_token_secret
↪")
)

# Create Twitter API object
twitter = tweepy.API(auth)

# let's collect some of the tweets in your public timeline
public_tweets = twitter.home_timeline()

for tweet in public_tweets:
    print(tweet.text)
```

3.4.4 Create a new table

Let us create a folder called `etl-basic` and a `stream_twitter.py` script in it.

We are going to create a SQL table to save the tweets to. This table will contain the following:

- username
- tweet content
- time of creation

- retweet count
- unique tweet id

This corresponds to 5 columns and the primary key.

We already know how to connect to a database:

```python
from mysql import connector as mysql

# Details for our MySql connection
DATABASE = {
    "host": "localhost",
    "user": "airflow",
    "password": "python2019",
    "db": "airflowdb",
}

# ----------------------------------------------
#  Database related functions
# ----------------------------------------------


def connect_db(my_database):
    """Connect to a given my_database

    Args:
        my_database(dict): dictionary with the my_database details

    Returns:
        dbconnect: MySql my_database connection object
    """
    try:
        dbconnect = mysql.connect(
            host=my_database.get("host"),
            user=my_database.get("user"),
            password=my_database.get("password"),
            db=my_database.get("db"),
        )
        print("connected")
        return dbconnect
    except mysql.Error as e:
        print(e)
```

Now we need to write a function to create the table

```python
def create_table(my_database, new_table):
    """Create new table in a my_database

    Args:
        my_database (dict): details for the db
        new_table (str): name of the table to create
    """

    dbconnect = connect_db(my_database)

    # create a cursor for the queries
    cursor = dbconnect.cursor()
    cursor.execute("USE airflowdb")
```

(continues on next page)

(continued from previous page)

```
    # here we delete the table, it can be kept or else
    cursor.execute(f"DROP TABLE IF EXISTS {new_table}")

    # these matches the Twitter data
    query = (
        f"CREATE TABLE `{new_table}` ("
        "  `id` INT(11) NOT NULL AUTO_INCREMENT,"
        "  `user` varchar(100) NOT NULL ,"
        "  `created_at` timestamp,"
        "  `tweet` varchar(255) NOT NULL,"
        "  `retweet_count` int(11) ,"
        "  `id_str` varchar(100),"
        "  PRIMARY KEY (`id`))"
    )

    cursor.execute(query)
    dbconnect.close()
    cursor.close()

    return print(f"Created {new_table} table")
```

3.4.5 Collect Tweets

The Twitter Streaming API has rate limits and prohibits too many connection attempts happening too quickly. It also prevents too many connections being made to it using the same authorization keys. Thankfully, tweepy takes care of these details for us, and we can focus on our program.

The main thing that we have to be aware of is the queue of tweets that we're processing. If we take too long to process tweets, they will start to get queued, and Twitter may disconnect us. This means that processing each tweet needs to be extremely fast.

Let's us transform the connection script we created before:

```
# Import libraries needed
import json
import time

from configparser import ConfigParser
from pathlib import Path

import tweepy
from dateutil import parser
from mysql import connector as mysql

# Path to the config file with the keys make sure not to commit this file
CONFIG_FILE = Path.cwd() / "config.cfg"

def connectTwitter():
    config = ConfigParser()
    config.read(CONFIG_FILE)

    #  complete the part to Authenticate to Twitter

```

(continues on next page)

(continued from previous page)

```
    # Create Twitter API object
    twitter = tweepy.API(auth, wait_on_rate_limit=True, wait_on_rate_limit_
↪notify=True)

    print(f" Connected as {twitter.me().screen_name}")

    return twitter
```

The next step is to create a stream listener.

The `StreamListener` class has a method called on_data. This method will automatically figure out what kind of data Twitter sent, and call an appropriate method to deal with the specific data type. It's possible to deal with events like users sending direct messages, tweets being deleted, and more.

```
class customListener(tweepy.StreamListener):
    """We need to create an instance of the Stream Listener
    http://docs.tweepy.org/en/v3.4.0/streaming_how_to.html
    """

    def on_error(self, status_code):
        if status_code == 420:
            # returning False in on_data disconnects the stream
            return False

    def on_status(self, status):
        print(status.text)
        return True

    def on_data(self, data):
        """
        Automatic detection of the kind of data collected from Twitter
        This method reads in tweet data as JSON and extracts the data we want.
        """
        try:
            # parse as json
            raw_data = json.loads(data)

            # extract the relevant data
            if "text" in raw_data:
                user = raw_data["user"]["screen_name"]
                created_at = parser.parse(raw_data["created_at"])
                tweet = raw_data["text"]
                retweet_count = raw_data["retweet_count"]
                id_str = raw_data["id_str"]

            # insert data just collected into MySQL my_database
            populate_table(user, created_at, tweet, retweet_count, id_str)
            print(f"Tweet colleted at: {created_at}")

        except Error as e:
            print(e)
```

In pairs discuss how you would create the `populate_table` function

```
def populate_table(
    user, created_at, tweet, retweet_count, id_str, my_database=DATABASE
):
```

(continues on next page)

(continued from previous page)

```
    """Populate a given table witht he Twitter collected data

    Args:
        user (str): username from the status
        created_at (datetime): when the tweet was created
        tweet (str): text
        retweet_count (int): number of retweets
        id_str (int): unique id for the tweet
    """

    dbconnect = connect_db(DATABASE)

    cursor = dbconnect.cursor()
    cursor.execute("USE airflowdb")

    # add content here

    try:
        # what is missing?
        commit()
        print("commited")

    except mysql.Error as e:
        print(e)
        dbconnect.rollback()

    cursor.close()
    dbconnect.close()

    return
```

For this to be called we need to wrap around a main function:

```
# complete the main function
if __name__ == "__main__":

    create_table(DATABASE, "tweets")
    # first we need to authenticate
    twitter = connectTwitter()

    # next: create stream listener
    myStreamListener = customListener()
    myStream = tweepy.Stream(auth=twitter.auth, listener=myStreamListener, timeout=30)

    # stream tweets using the filter method
    start_stream(myStream, track=["python", "pycon", "jupyter", "#pycon2019"], ␣
↪async=True)
```

Solutions at: https://github.com/trallard/airflow-tutorial/solutions

3.4.6 Extending your data pipeline

So far we have collected some data through streaming (enough to collect some data). And created a database where this data is going to be deposited into.

The next step is to transform the data and prepare it for more downstream processes.

There are different mechanisms to share data between pipeline steps:

- files
- databases
- queues

In each case, we need a way to get data from the current step to the next step.

The next step would be to add some 'transformation' steps to the data set.

Modify your `stream_twitter.py` so that the table contains also: language, follower count and country.

Now let's create a script called `analyse_twitter.py`

This script will do the following:

- Load the table into a pandas dataframe
- Clean the data: lowercase usernames, remove RT
- Create a plot from the data
- Save the plot and a csv with the clean data

```
import os
import os.path
import re
from datetime import datetime
from pathlib import Path

import matplotlib.pyplot as plt
import mysql.connector as mysql
import pandas as pd

# import the previously created functions
from stream_twitter import connect_db

# Details for our MySql connection
DATABASE = {
    "host": "localhost",
    "user": "airflow",
    "password": "python2019",
    "db": "airflowdb",
}

# ----------------------------------------------
#  Database related functions
# ----------------------------------------------


def sql_to_csv(my_database, my_table):

    dbconnect = connect_db(my_database)

    cursor = dbconnect.cursor()

    query = f"SELECT * FROM {table}"
    all_tweets = pd.read_sql_query(query, dbconnect)

    if os.path.exists("./data"):
```

(continues on next page)

(continued from previous page)

```
        all_tweets.to_csv("./data/raw_tweets.csv", index=False)

    else:
        os.mkdir("./data")
        all_tweets.to_csv("./data/raw_tweets.csv", index=False)


def sql_to_df(my_database, my_table):
    dbconnect = connect_db(my_database)

    cursor = dbconnect.cursor()

    query = f"SELECT * FROM {my_table}"

    # store in data frame

    df = pd.read_sql_query(query, dbconnect, index_col="id")

    cursor.close()
    dbconnect.close()

    return df


# ----------------------------------------------
#  Data processing
# ----------------------------------------------


def clean_data(df):

    # Make all usernames lowercase
    clean_df = df.copy()
    clean_df["user"] = df["user"].str.lower()

    # keep only non RT
    clean_df = clean_df[~clean_df["tweet"].str.contains("RT")]

    return clean_df


def create_plots(df):
    x = df["language"].unique()
    fig, ax = plt.subplots()
    countries = df["language"].value_counts()
    plt.bar(range(len(countries)), countries)
    fig.suptitle("Language counts")
    plt.xlabel("languages")
    plt.ylabel("count")
    ax.set_xticklabels(x)

    if os.path.exists("./plots"):
        fig.savefig("./plots/barchart_lang.png")

    else:
        os.mkdir("./plots")
        fig.savefig("./plots/barchart_lang.png")
```

(continues on next page)

(continued from previous page)

```python


def save_df(df):
    today = datetime.today().strftime("%Y-%m-%d")

    if os.path.exists("./data"):
        df.to_csv(f"./data/{today}-clean-df.csv", index=None)

    else:
        os.mkdir("./data")
        df.to_csv(f"./data/{today}-clean-df.csv", index=None)


if __name__ == "__main__":

    df = sql_to_df(DATABASE, "tweets_long")
    print("Database loaded in df")

    clean_df = clean_data(df)

    create_plots(clean_df)

    save_df(clean_df)
```

In pairs discuss:

- How would you run the two scripts together?
- Try and create the pipeline: `stream_twitter.py` -> `analyse_twitter.py`

CHAPTER 4

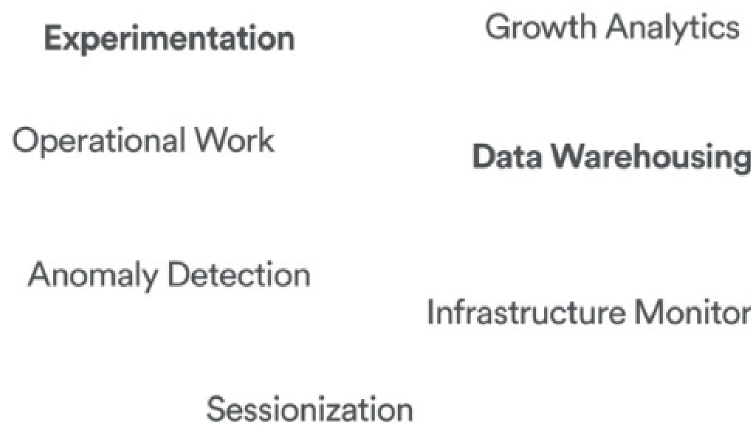
Airflow basics

4.1 What is Airflow?

airflow logo

Airflow is a Workflow engine which means:

- Manage scheduling and running jobs and data pipelines
- Ensures jobs are ordered correctly based on dependencies
- Manage the allocation of scarce resources
- Provides mechanisms for tracking the state of jobs and recovering from failure



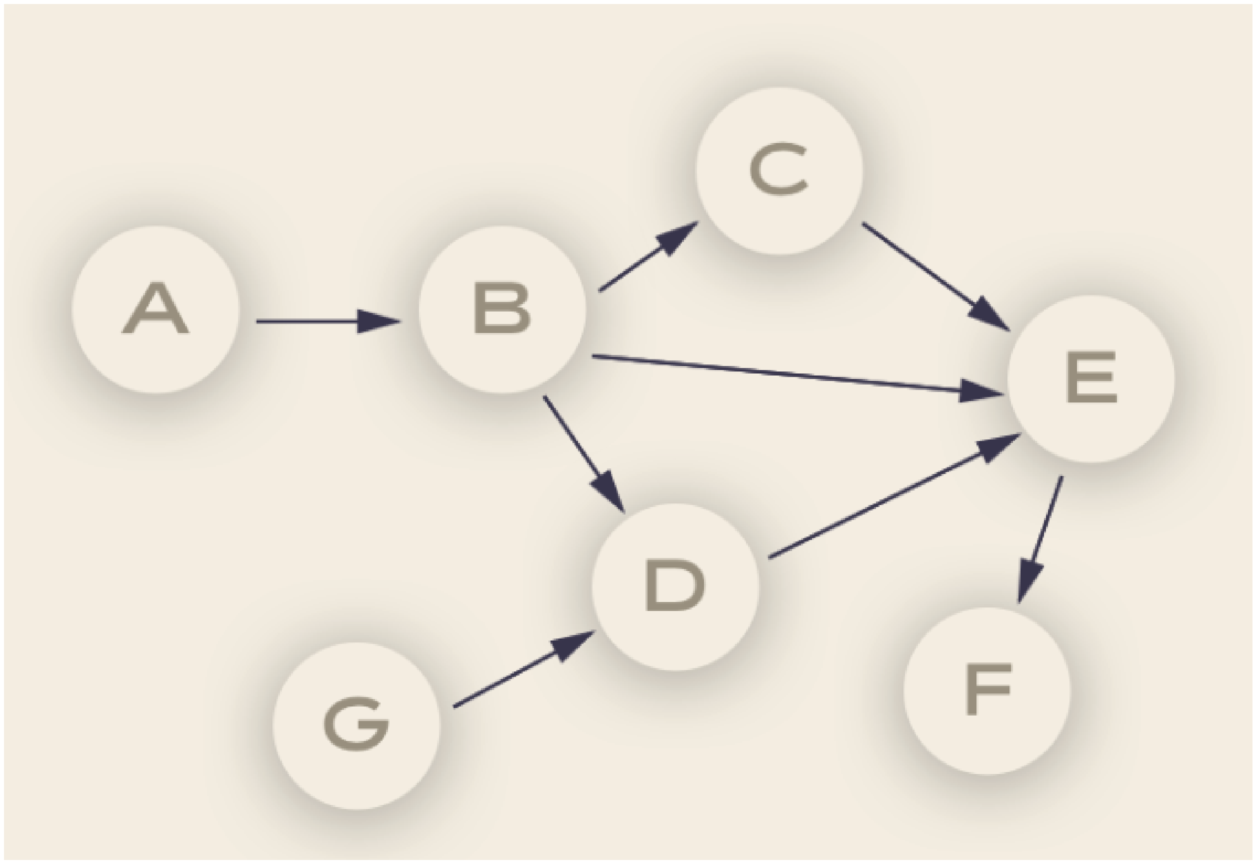
It is highly versatile and can be used across many many domains:

4.2 Basic Airflow concepts

- **Task**: a defined unit of work (these are called operators in Airflow)
- **Task instance**: an individual run of a single task. Task instances also have an indicative state, which could be "running", "success", "failed", "skipped", "up for retry", etc.
- **DAG**: Directed acyclic graph, a set of tasks with explicit execution order, beginning, and end
- **DAG run**: individual execution/run of a DAG

Debunking the DAG

The vertices and edges (the arrows linking the nodes) have an order and direction associated to them



each node in a DAG corresponds to a task, which in turn represents some sort of data processing. For example:

Node A could be the code for pulling data from an API, node B could be the code for anonymizing the data. Node B could be the code for checking that there are no duplicate records, and so on.

These ‘pipelines’ are acyclic since they need a point of completion.

Dependencies

Each of the vertices has a particular direction that shows the relationship between certain nodes. For example, we can only anonymize data once this has been pulled out from the API.

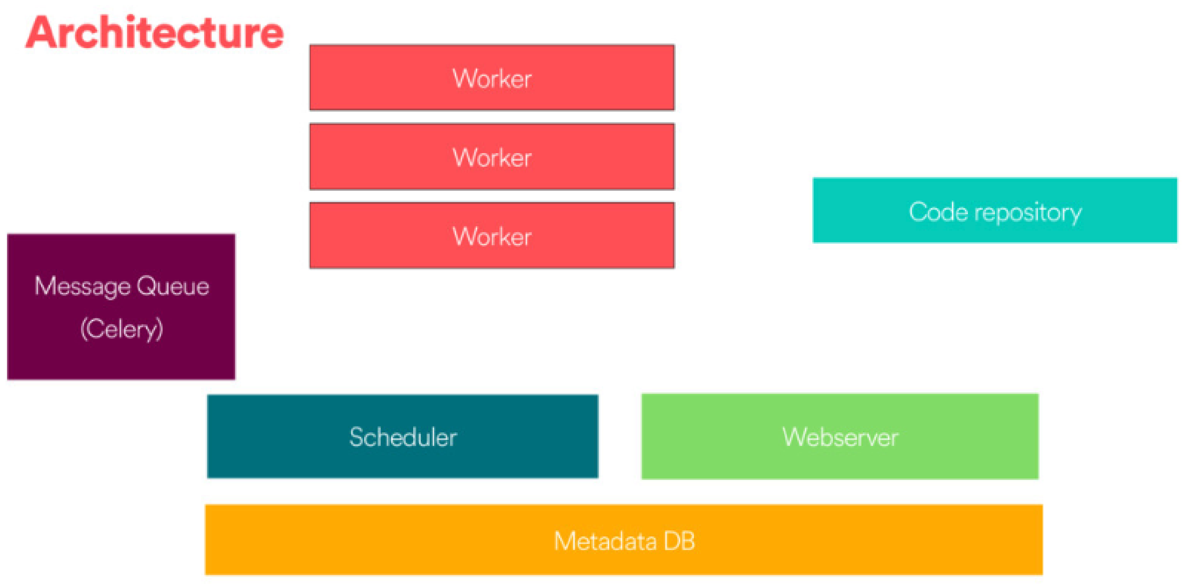
4.3 Idempotency

This is one of the most important characteristics of good ETL architectures.

When we say that something is idempotent it means it will produce the same result regardless of how many times this is run (i.e. the results are reproducible).

Reproducibility is particularly important in data-intensive environments as this ensures that the same inputs will always return the same outputs.

4.4 Airflow components



There are 4 main components to Apache Airflow:

4.4.1 Web server

The GUI. This is under the hood a Flask app where you can track the status of your jobs and read logs from a remote file store (e.g. Azure Blobstorage).

4.4.2 Scheduler

This component is responsible for scheduling jobs. This is a multithreaded Python process that uses the DAGb object to decide what tasks need to be run, when and where.

The task state is retrieved and updated from the database accordingly. The web server then uses these saved states to display job information.

4.4.3 Executor

The mechanism that gets the tasks done.

4.4.4 Metadata database

- Powers how the other components interact
- Stores the Airflow states
- All processes read and write from here

4.5 Workflow as a code

One of the main advantages of using a workflow system like Airflow is that all is code, which makes your workflows maintainable, versionable, testable, and collaborative.

Thus your workflows become more explicit and maintainable (atomic tasks).

Not only your code is dynamic but also is your infrastructure.

4.5.1 Defining tasks

Tasks are defined based on the abstraction of `Operators` (see Airflow docs here) which represent a single **idempotent task**.

The best practice is to have atomic operators (i.e. can stand on their own and do not need to share resources among them).

You can choose among;

- `BashOperator`
- `PythonOperator`
- `EmailOperator`
- `SimpleHttpOperator`
- `MySqlOperator` (and other DB)

Examples:

```
t1 = BashOperator(task_id='print_date',
    bash_command='date,
    dag=dag)
```

```
def print_context(ds, **kwargs):
    pprint(kwargs)
    print(ds)
    return 'Whatever you return gets printed in the logs'


run_this = PythonOperator(
    task_id='print_the_context',
    provide_context=True,
    python_callable=print_context,
    dag=dag,
)
```

4.6 Comparing Luigi and Airflow

4.6.1 Luigi

- Created at Spotify (named after the plumber)
- Open sourced in late 2012
- GNU make for data

4.6.2 Airflow

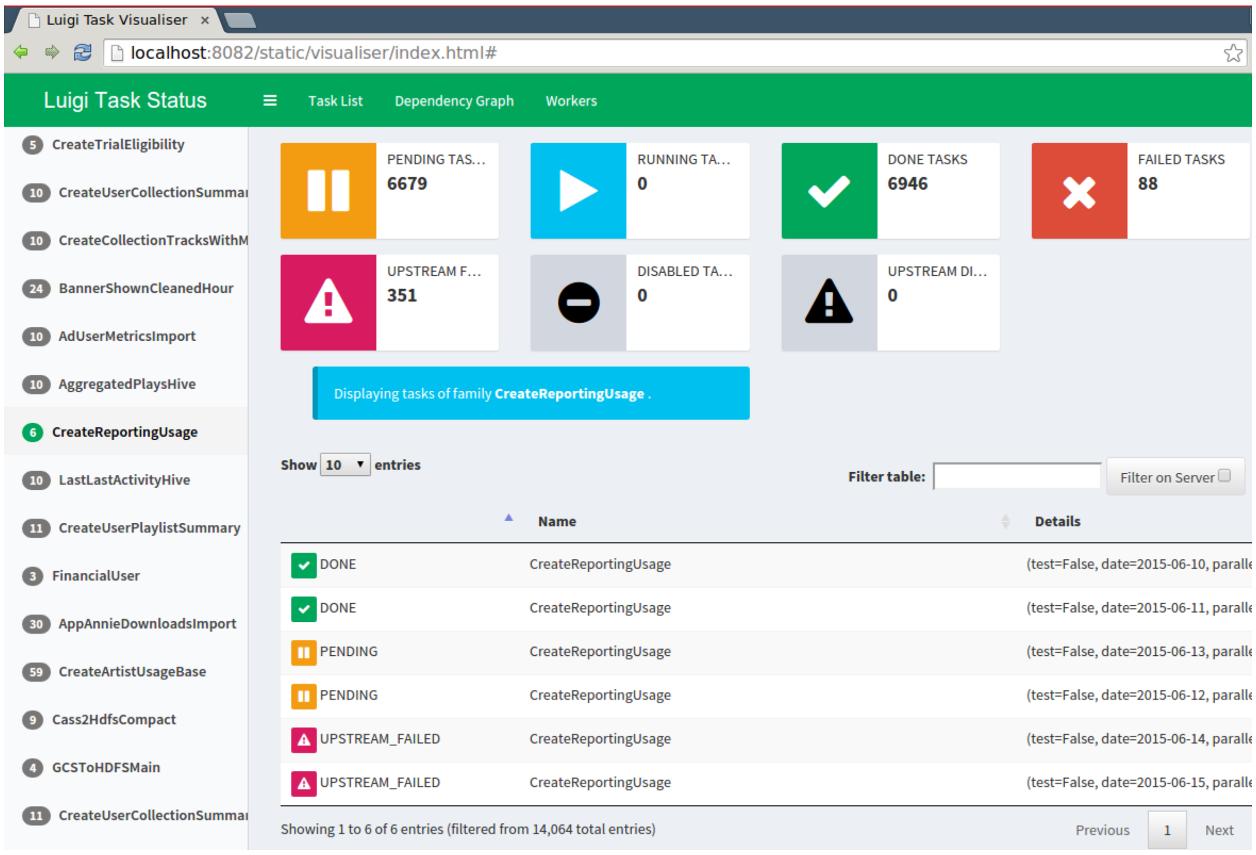
- Airbnb data team
- Open-sourced mud 2015
- Apache incubator mid-2016
- ETL pipelines

4.6.3 Similarities

- Python open source projects for data pipelines
- Integrate with a number of sources (databases, filesystems)
- Tracking failure, retries, success
- Ability to identify the dependencies and execution

4.6.4 Differences

- Scheduler support: Airflow has built-in support using schedulers
- Scalability: Airflow has had stability issues in the past
- Web interfaces



5	CreateTrialEligibility
10	CreateUserCollectionSummar
10	CreateCollectionTracksWithM
24	BannerShownCleanedHour
10	AdUserMetricsImport
10	AggregatedPlaysHive
6	CreateReportingUsage
10	LastLastActivityHive
11	CreateUserPlaylistSummary
3	FinancialUser
30	AppAnnieDownloadsImport
59	CreateArtistUsageBase
9	Cass2HdfsCompact
4	GCSToHDFSMain
11	CreateUserCollectionSummar

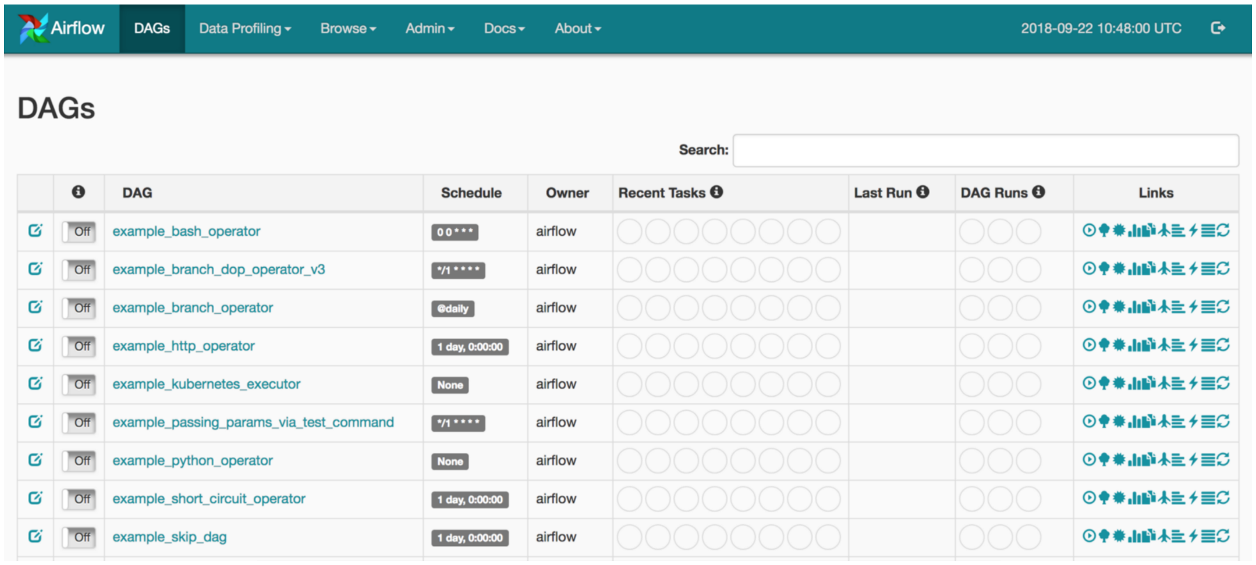
PENDING TAS... 6679
RUNNING TA... 0
DONE TASKS 6946
FAILED TASKS 88
UPSTREAM F... 351
DISABLED TA... 0
UPSTREAM DI... 0

Displaying tasks of family **CreateReportingUsage** .

Show 10 entries   Filter table: Filter on Server

	Name	Details
DONE	CreateReportingUsage	(test=False, date=2015-06-10, paralle
DONE	CreateReportingUsage	(test=False, date=2015-06-11, paralle
PENDING	CreateReportingUsage	(test=False, date=2015-06-13, paralle
PENDING	CreateReportingUsage	(test=False, date=2015-06-12, paralle
UPSTREAM_FAILED	CreateReportingUsage	(test=False, date=2015-06-14, paralle
UPSTREAM_FAILED	CreateReportingUsage	(test=False, date=2015-06-15, paralle

Showing 1 to 6 of 6 entries (filtered from 14,064 total entries)   Previous 1 Next



Airflow | DAGs | Data Profiling ▾ | Browse ▾ | Admin ▾ | Docs ▾ | About ▾ | 2018-09-22 10:48:00 UTC

DAGs

Search:

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	Off	example_bash_operator	0 0 * * *	airflow	○○○○○○○○		○○○	
	Off	example_branch_dop_operator_v3	*/1 * * * *	airflow	○○○○○○○○		○○○	
	Off	example_branch_operator	@daily	airflow	○○○○○○○○		○○○	
	Off	example_http_operator	1 day, 0:00:00	airflow	○○○○○○○○		○○○	
	Off	example_kubernetes_executor	None	airflow	○○○○○○○○		○○○	
	Off	example_passing_params_via_test_command	*/1 * * * *	airflow	○○○○○○○○		○○○	
	Off	example_python_operator	None	airflow	○○○○○○○○		○○○	
	Off	example_short_circuit_operator	1 day, 0:00:00	airflow	○○○○○○○○		○○○	
	Off	example_skip_dag	1 day, 0:00:00	airflow	○○○○○○○○		○○○	

| Airflow | Luigi | | ———————————————— | ———————————————————————————— | | Task are defined by`dag_id` defined by user name | Task are defined by task name and parameters | | Task retries based on definitions | Decide if a task is done via input/output | | Task code to the worker | Workers started by Python file where the tasks are defined | | Centralized scheduler (Celery spins up workers) | Centralized scheduler in charge of deduplication sending tasks (Tornado based) |

CHAPTER 5

Airflow 101: working locally and familiarise with the tool

5.1 Pre-requisites

The following prerequisites are needed:

- Libraries detailed in the Setting up section (either via conda or pipenv)
- MySQL installed
- text editor
- command line

5.2 Getting your environment up and running

If you followed the instructions you should have Airflow installed as well as the rest of the packages we will be using.

So let's get our environment up and running:

If you are using conda start your environment via:

```
$ source activate airflow-env
```

If using pipenv then:

```
$ pipenv shell
```

this will start a shell within a virtual environment, to exit the shell you need to type `exit` and this will exit the virtual environment.

5.3 Starting Airflow locally

Airflow home lives in `~/airflow` by default, but you can change the location before installing airflow. You first need to set the `AIRFLOW_HOME` environment variable and then install airflow. For example, using pip:

```
export AIRFLOW_HOME=~/mydir/airflow

# install from PyPI using pip
pip install apache-airflow
```

once you have completed the installation you should see something like this in the `airflow` directory (wherever it lives for you)

```
drwxr-xr-x    - myuser 18 Apr 14:02 .
.rw-r--r--  26k myuser 18 Apr 14:02 ├── airflow.cfg
drwxr-xr-x    - myuser 18 Apr 14:02 ├── logs
drwxr-xr-x    - myuser 18 Apr 14:02 │   └── scheduler
drwxr-xr-x    - myuser 18 Apr 14:02 │       ├── 2019-04-18
lrwxr-xr-x   46 myuser 18 Apr 14:02 │       └── latest -> /Users/myuser/airflow/logs/
↪scheduler/2019-04-18
.rw-r--r-- 2.5k myuser 18 Apr 14:02 └── unittests.cfg
```

We need to create a local dag folder:

```
mkdir ~/airflow/dags
```

As your project evolves, your directory will look something like this:

```
airflow                  # the root directory.
├── dags                 # root folder for all dags. files inside folders are not␣
↪searched for dags.
│   ├── my_dag.py, # my dag (definitions of tasks/operators) including precedence.
│   └── ...
├── logs                 # logs for the various tasks that are run
│   └── my_dag           # DAG specific logs
│   │   ├── src1_s3      # folder for task-specific logs (log files are created by␣
↪date of a run)
│   │   ├── src2_hdfs
│   │   ├── src3_s3
│   │   └── spark_task_etl
├── airflow.db           # SQLite database used by Airflow internally to track the␣
↪status of each DAG.
├── airflow.cfg          # global configuration for Airflow (this can be overridden␣
↪by config inside the file.)
└── ...
```

5.4 Prepare your database

As we mentioned before Airflow uses a database to keep track of the tasks and their statuses. So it is critical to have one set up.

To start the default database we can run `airflow initdb`. This will initialize your database via alembic so that it matches the latest Airflow release.

The default database used is `sqlite` which means you cannot parallelize tasks using this database. Since we have MySQL and MySQL client installed we will set them up so that we can use them with airflow.

Create an airflow database

From the command line:

```
MySQL -u root -p
mysql> CREATE DATABASE airflow CHARACTER SET utf8 COLLATE utf8_unicode_ci;
mysql> GRANT ALL PRIVILEGES ON airflow.* To 'airflow'@'localhost';
mysql> FLUSH PRIVILEGES;
```

and initialize the database:

```
airflow initdb
```

Notice that this will fail with the default `airflow.cfg`

5.5 Update your local configuration

Open your airflow configuration file `~/airflow/airflow.cf` and make the following changes:

```
executor = CeleryExecutor
```

```
# http://docs.celeryproject.org/en/latest/userguide/configuration.html#broker-settings
# needs rabbitmq running
broker_url = amqp://guest:guest@127.0.0.1/


# http://docs.celeryproject.org/en/latest/userguide/configuration.html#task-result-
↪backend-settings
result_backend = db+mysql://airflow:airflow@localhost:3306/airflow

sql_alchemy_conn = mysql://airflow:python2019@localhost:3306/airflow
```

Here we are replacing the default executor (`SequentialExecutor`) with the `CeleryExecutor` so that we can run multiple DAGs in parallel. We also replace the default `sqlite` database with our newly created `airflow` database.

Now we can initialize the database:

```
airflow initdb
```

Let's now start the web server locally:

```
airflow webserver -p 8080
```

we can head over to http://localhost:8080 now and you will see that there are a number of examples DAGS already there.

Take some time to familiarise with the UI and get your local instance set up

Now let's have a look at the connections (http://localhost:8080/admin/connection/) go to `admin > connections`. You should be able to see a number of connections available. For this tutorial, we will use some of the connections including `mysql`.

5.5.1 Commands

Let us go over some of the commands. Back on your command line:

```
airflow list_dags
```

we can list the DAG tasks in a tree view

```
airflow list_tasks tutorial --tree
```

we can tests the dags too, but we will need to set a date parameter so that this executes:

```
airflow test tutorial print_date 2019-05-01
```

(note that you cannot use a future date or you will get an error)

```
airflow test tutorial templated 2019-05-01
```

By using the test commands these are not saved in the database.

Now let's start the scheduler:

```
airflow scheduler
```

Behind the scenes, it monitors and stays in sync with a folder for all DAG objects it contains. The Airflow scheduler is designed to run as a service in an Airflow production environment.

Now with the schedule up and running we can trigger an instance:

```
$ airflow run airflow run example_bash_operator runme_0 2015-01-01
```

This will be stored in the database and you can see the change of the status change straight away.

What would happen for example if we wanted to run or trigger the `tutorial` task?

Let's try from the CLI and see what happens.

```
airflow trigger_dag tutorial
```

5.6 Writing your first DAG

Let's create our first simple DAG. Inside the dag directory (`~/airflow/dags)` create a `simple_dag.py` file.

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator


def print_hello():
    return "Hello world!"


default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2019, 4, 30),
    "email": ["airflow@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
```

(continues on next page)

(continued from previous page)

```
    "retries": 1,
    "retry_delay": timedelta(minutes=2),
}

dag = DAG(
    "hello_world",
    description="Simple tutorial DAG",
    schedule_interval="0 12 * * *",
    default_args=default_args,
    catchup=False,
)

t1 = DummyOperator(task_id="dummy_task", retries=3, dag=dag)

t2 = PythonOperator(task_id="hello_task", python_callable=print_hello, dag=dag)

# sets downstream foe t1
t1 >> t2

# equivalent
# t2.set_upstream(t1)
```

If it is properly setup you should be able to see this straight away on your instance.

5.6.1 Now let's create a DAG from the previous ETL pipeline (kind of)

All hands on - check the solutions

CHAPTER 6

About your facilitator

My name is Tania. I live in Manchester UK where I work as a Cloud Advocate for Microsoft.

Over the years, I have worked as a data engineer, machine learning engineer, and research software engineer. I love data intensive enviroments and I am particularly interested in the tools and workflows to deliver robust, reproducible data insights.

If you have any questions or feedback about this tutorial please, file an issue using the following link: https://github.com/trallard/airflow-tutorial/issues/new.

You can also contact me via the following channels:

- E-mail: trallard@bitsandchips.me
- Twitter: @ixek
- Tania on GitHub

CHAPTER 7

Code of Conduct

All attendees to this workshop are expected to adhere to PyCon's Code of Conduct, in brief: **Be open, considerate, and respectful.**

CHAPTER 8

License