



EECS E6893 Big Data Analytics

HW4: Data Analytics Pipeline

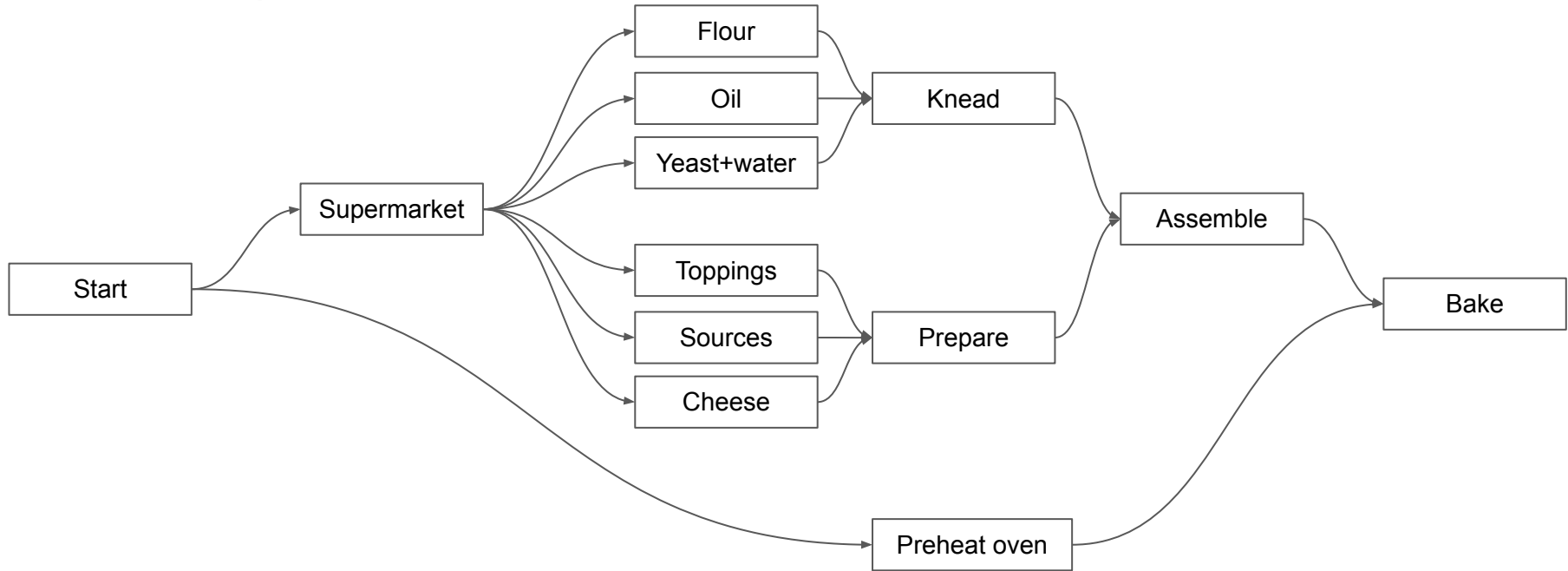
Cong Han, ch3212@columbia.edu

Workflow

- A sequence of tasks involved in moving from the beginning to the end of a working process
- Started on a schedule or triggered by an event

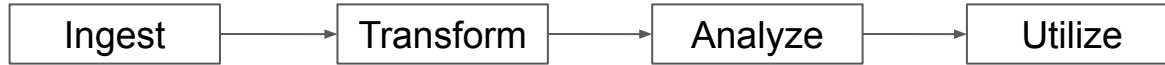
Workflow

- Cook a pizza



Workflow

- Data analytics





- A platform let you create, schedule, monitor and manage workflows

Principles:

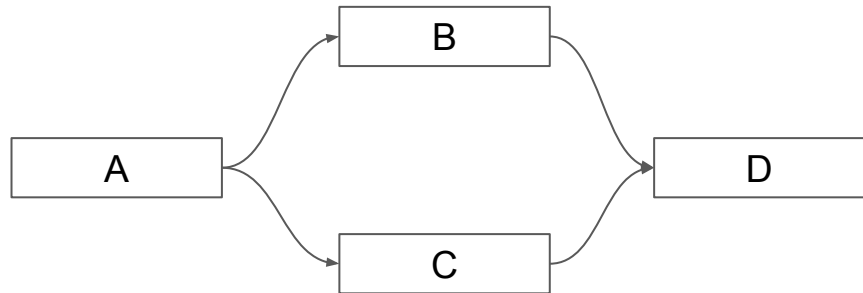
- Scalable
- Dynamic
- Extensible
- Elegant

Features:

- Pure Python
- Useful UI
- Robust Integrations
- Easy to Use
- Open Source

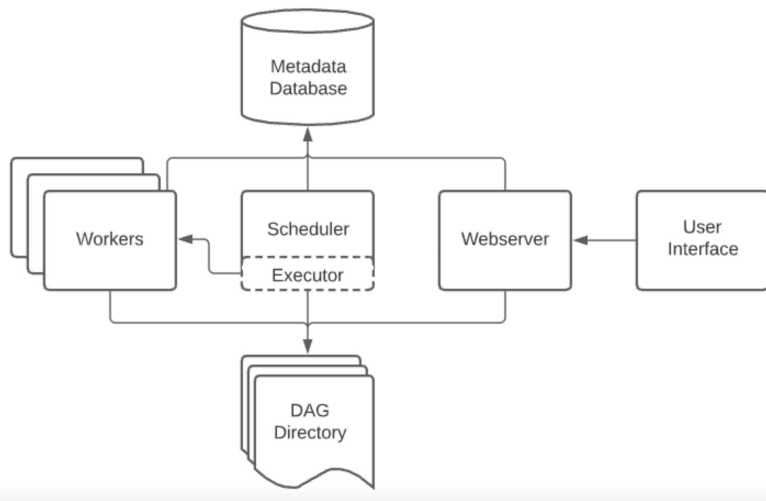
DAG (Directed Acyclic Graph)

- In Airflow, workflows are created using DAGs
- A DAG is a collection of tasks that you want to schedule and run, organized in a way that reflects their relationships and dependencies
- The tasks describe what to do, e.g., fetching data, running analysis, triggering other systems, or more
- A DAG ensures that each task is executed at the right time, in the right order, or with the right issue handling
- A DAG is written in Python



Airflow architecture

- **Scheduler:** handles both triggering scheduled workflows, and submitting Tasks to the executor to run.
- **Executor:** handles running tasks.
- **Webserver:** a handy user interface to inspect, trigger and debug the behaviour of DAGs and tasks.
- **A folder of DAG files:** read by the scheduler and executor
- **A metadata database:** used by the scheduler, executor and webserver to store state



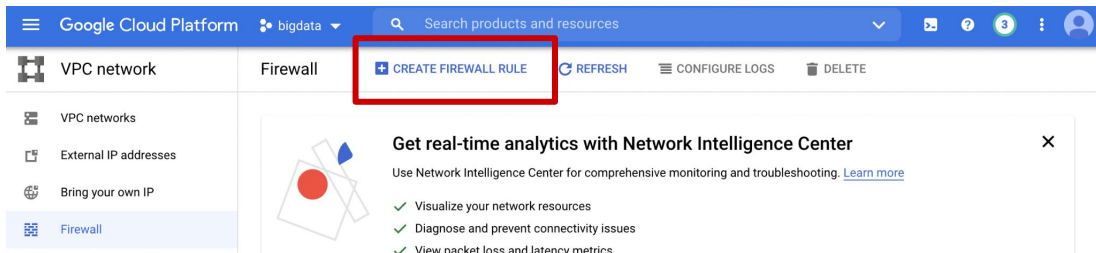
Airflow installation



Three choices


1. **Install and use Airflow in the VM of GCP**
2. Install and use airflow in your local machines
3. Google composer


Set up the firewall



- VPC network → Firewall → Create Firewall rule
- Set service account scope and protocols and ports





Targets 
Specified service account 



Service account scope 
☒ In this project
☐ In another project


Target service account 

Source filter 
Service account 

Service account scope 
☒ In this project
☐ In another project

Source service account 

Second source filter 
None 

Protocols and ports 
☒ Allow all
☐ Specified protocols and ports

Create a VM instance

The screenshot shows the Google Cloud Platform console interface. The top navigation bar includes the Google Cloud Platform logo, a dropdown menu for 'bigdata', a search bar with 'compute engine', and various utility icons. The left sidebar shows the 'Compute Engine' section with a list of resources: Virtual machines (VM instances, Instance templates, Sole-tenant nodes, Machine images, TPUs, Committed use discounts, Migrate for Compute Engi...), and Storage (Disks, Snapshots, Images). The main content area is titled 'VM instances' and features a 'CREATE INSTANCE' button highlighted with a red box. Below this is a table with columns: Status, Name, Zone, Recommendation, and Connect. The table is currently empty. To the right of the table is a 'Select an instance' panel with tabs for PERMISSIONS, LABELS, and MONITORING. A message in the panel states: 'Please select at least one resource.' At the bottom of the main content area, there are two buttons: 'CREATE INSTANCE' and 'TAKE THE QUICKSTART'.

Google Cloud Platform bigdata compute engine

Compute Engine VM instances **CREATE INSTANCE** OPERATIONS HELP ASSISTANT HIDE INFO PANEL

Virtual machines

- VM instances
- Instance templates
- Sole-tenant nodes
- Machine images
- TPUs
- Committed use discounts
- Migrate for Compute Engi...

Storage

- Disks
- Snapshots
- Images

Filter Enter property name or value

Status	Name	Zone	Recommendation	Connect
--------	------	------	----------------	---------

Select an instance

PERMISSIONS LABELS MONITORING

Please select at least one resource.

VM Instances

Compute Engine lets you create VM Instances, which are virtual machines that run on Google's infrastructure. Create micro-VMs or larger instances running Debian, Windows, or other standard images. Create your first VM instance, import it using a migration service, or try the quickstart to build a sample app.

CREATE INSTANCE TAKE THE QUICKSTART

Create a VM instance

Boot disk ?

Type	New balanced persistent disk
Size	50 GB
Image	 Ubuntu 18.04 LTS

CHANGE

Identity and API access ?

Service accounts ?

Service account
Compute Engine default service account ▼

Access scopes ?

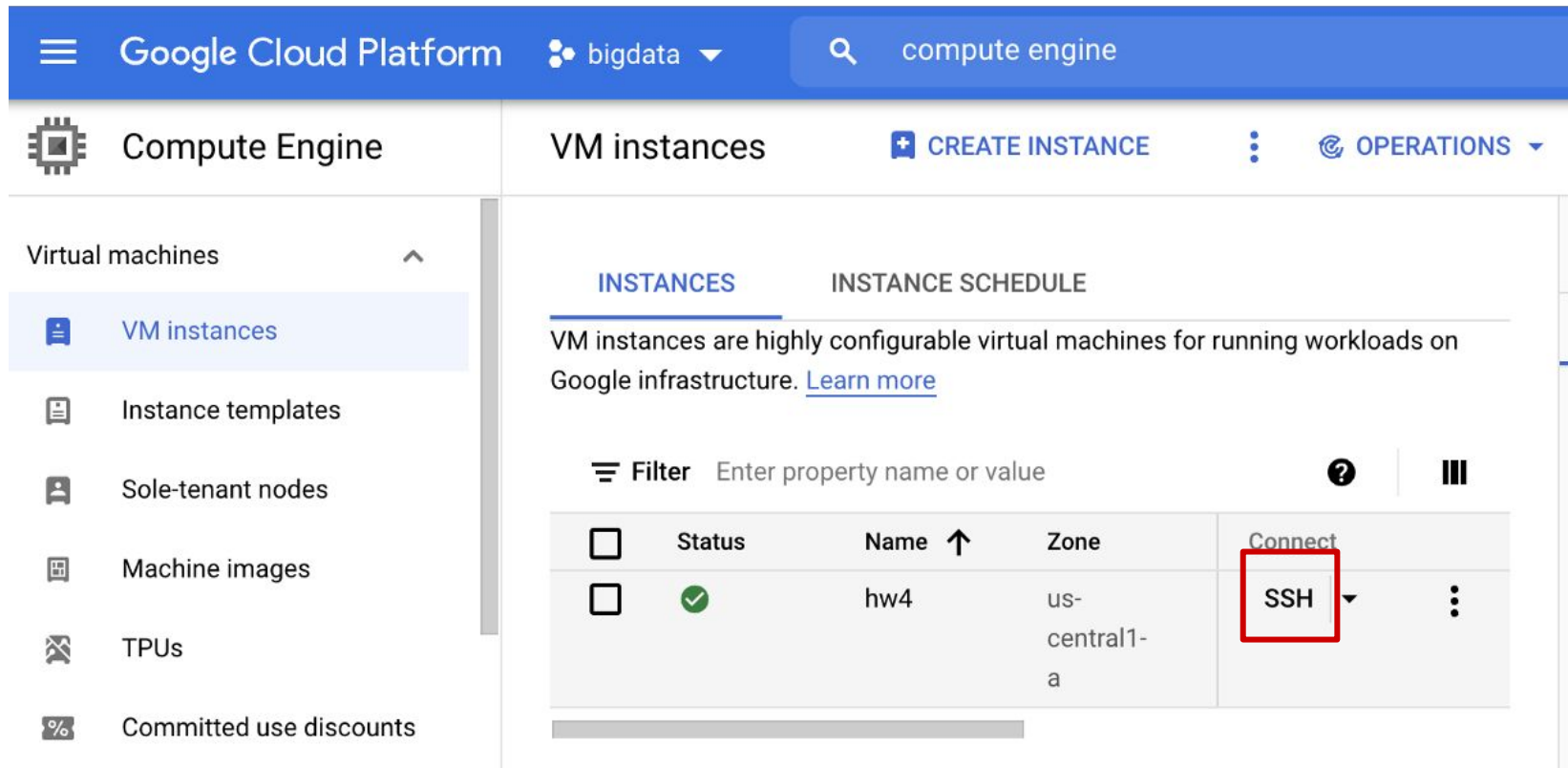
- ☐ Allow default access
- ☒ Allow full access to all Cloud APIs
- ☐ Set access for each API

Firewall ?

Add tags and firewall rules to allow specific network traffic from the Internet

- ☒ Allow HTTP traffic
- ☒ Allow HTTPS traffic

Connect to your VM using SSH



The screenshot shows the Google Cloud Platform console interface. The top navigation bar includes the Google Cloud Platform logo, a 'bigdata' dropdown menu, and a search bar containing 'compute engine'. The left sidebar lists various services, with 'Compute Engine' selected. Under 'Compute Engine', the 'VM instances' section is highlighted. The main content area displays the 'VM instances' page, featuring a 'CREATE INSTANCE' button and an 'OPERATIONS' dropdown. Below this, there are tabs for 'INSTANCES' (selected) and 'INSTANCE SCHEDULE'. A descriptive text states: 'VM instances are highly configurable virtual machines for running workloads on Google infrastructure. [Learn more](#)'. A filter bar is present with the text 'Filter Enter property name or value'. Below the filter, a table lists VM instances. The table has columns for 'Status', 'Name', and 'Zone'. The first instance listed is 'hw4' with a status of '✓' and a zone of 'us-central1-a'. To the right of the table, there is a 'Connect' dropdown menu, which is highlighted with a red box and shows the 'SSH' option selected. A vertical ellipsis menu is also visible next to the 'SSH' option.

Google Cloud Platform bigdata compute engine

Compute Engine VM instances CREATE INSTANCE OPERATIONS

Virtual machines

VM instances

Instance templates

Sole-tenant nodes

Machine images

TPUs

Committed use discounts

INSTANCES INSTANCE SCHEDULE

VM instances are highly configurable virtual machines for running workloads on Google infrastructure. [Learn more](#)

Filter Enter property name or value

Status	Name	Zone	Connect
✓	hw4	us-central1-a	SSH

Connect to your VM using SSH

```
The programs included with the Ubuntu system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*/copyright.
```

```
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by  
applicable law.
```

```
ch3212@hw4:~$ █
```

Install and update packages

1. `sudo apt update`
2. `sudo apt -y upgrade`
3. `sudo apt-get install wget`
4. `sudo apt install -y python3-pip`

Download miniconda and create a virtual environment

1. `mkdir -p ~/miniconda3`
 2. `wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/miniconda3/miniconda.sh`
 3. `bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3`
 4. `rm -rf ~/miniconda3/miniconda.sh`
 5. `~/miniconda3/bin/conda init bash`
 6. `~/miniconda3/bin/conda init zsh`
- # reopen (or we say reconnect) your terminal and create a new environment
7. `conda create --name airflow python=3.8`
- # activate the environment (everytime you open a new terminal, you should run this)
- 8. conda activate airflow**
- # optional but in case you don't like warnings
9. (optional) `pip install virtualenv`
 10. (optional) `pip install kubernetes`

```
(base) ch3212@hw4:~$ conda activate airflow
(airflow) ch3212@hw4:~$ █
```


Install Airflow

Airflow needs a home. `~/airflow` is the default, but you can put it

somewhere else if you prefer (optional)

```
export AIRFLOW_HOME=~/airflow
```

Install Airflow using the constraints file

```
AIRFLOW_VERSION=2.2.1
```

```
PYTHON_VERSION=3.8
```

For example: 3.8

```
CONSTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERSION}/constraints-${PYTHON_VERSION}.txt"
```

For example: <https://raw.githubusercontent.com/apache/airflow/constraints2.2.1/constraints-3.6.txt>

```
pip install "apache-airflow==${AIRFLOW_VERSION}" --constraint "${CONSTRAINT_URL}"
```

run airflow version to check if you install it successfully

```
airflow version
```

The Standalone command will initialise the database, make a user,

and start all components for you.

airflow standalone

Visit localhost:8080 in the browser and use the admin account details

shown on the terminal to login.

Enable the example_bash_operator dag in the home page

Initialize the database, make a user, and start webserver

Initialize the database, after this you will see a new folder airflow in your
\$AIRFLOW_HOME which contains configuration file airflow.cfg

1.airflow db init

2.airflow users create \

--username cong \

--password 123456 \

--firstname cong \

--lastname han \

--role Admin \

--email ch3212@columbia.edu

3.airflow webserver --port 8080

Airflow UI on your web browser

- Open your browser
- Visit “**http://<External IP>:8080**”
- login

In use by	Internal IP	External IP	Connect
	10.128.0.11 (nic0)	34.121.168.112 ✕	SSH ▾ ⋮



20:23 UTC → Log In

Sign In

Enter your login and password below:

Username:

cong

Password:

.....

Sign In

Start scheduler

Open a new terminal (you can use screen if you prefer to open only one
terminal)

1. conda activate airflow
2. airflow db init
3. airflow scheduler

Airflow examples

Helloworld

Download helloworld.py from Coursework/Files

Open a new terminal

conda activate airflow

Create dags folders

cd airflow

mkdir dags

cd dags

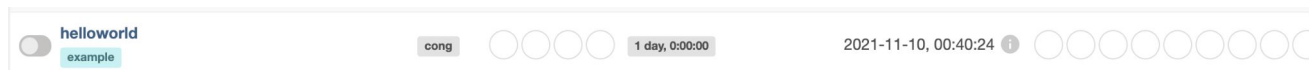
Upload helloworld.py here

Check if the script is correct, no errors if it's correct

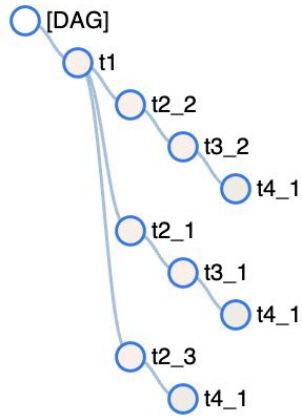
python helloworld.py

Initialize db again and you will see “hello” on the website

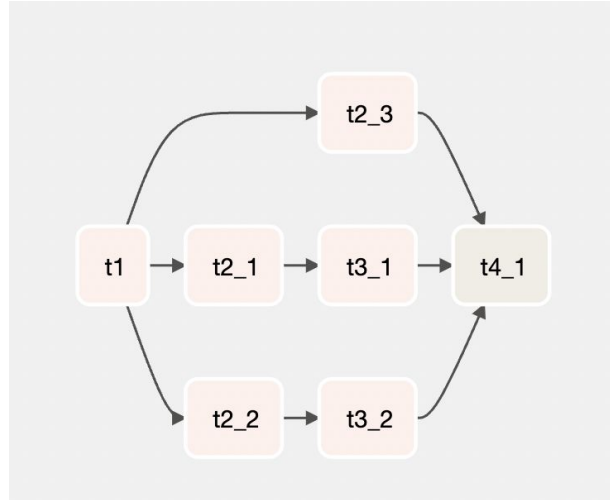
airflow db init



Helloworld



Tree



Graph

Two ways to trigger a DAG

1. Trigger manually
2. Trigger on a schedule

The screenshot displays the Apache Airflow web interface for a DAG named 'helloworld'. The interface includes a top navigation bar with tabs for Tree, Graph, Calendar, Task Duration, Task Tries, Landing Times, Gantt, Details, and Code. Below the navigation bar, there is a section for the DAG's configuration, including a 'Schedule' of '1 day, 0:00:00' and a 'Next Run' of '2021-11-10, 00:43:25'. A 'Trigger DAG' button is highlighted with a red box. Below this, there is a section for the DAG's runs, showing a list of runs with columns for 'Runs', '25', and 'Update'. The status of the DAG is 'No DAG runs yet.' At the bottom, there is a legend for the DAG's tasks, including 'BashOperator' and 'PythonOperator', and a status legend with various colors and labels.

Trigger manually

Scheduler

- The scheduler won't trigger your tasks until the period it covers has ended.
- The scheduler runs your job one `schedule_interval` after the start date, at the end of the interval.

References

<https://airflow.apache.org/docs/apache-airflow/stable/concepts/scheduler.html>

<https://airflow.apache.org/docs/apache-airflow/stable/dag-run.html>

<https://cloud.google.com/composer/docs/triggering-dags>

```
from datetime import datetime, timedelta
from textwrap import dedent
import time

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

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

# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args = {
    'owner': 'cong',
    'depends_on_past': False,
    'email': ['ch3212@columbia.edu'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(seconds=30),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function,
    # 'on_success_callback': some_other_function,
    # 'on_retry_callback': another_function,
    # 'sla_miss_callback': yet_another_function,
    # 'trigger_rule': 'all_success'
}
```

```
with DAG(
    'helloworld',
    default_args=default_args,
    description='A simple toy DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:
```

Tasks

```
with DAG(
    'helloworld',
    default_args=default_args,
    description='A simple toy DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=['example'],
) as dag:

    # task examples of tasks created by instantiating operators
    t1 = PythonOperator(
        task_id='t1',
        python_callable=correct_sleeping_function,
    )

    t2_1 = PythonOperator(
        task_id='t2_1',
        python_callable=correct_sleeping_function,
    )

    t2_2 = PythonOperator(
        task_id='t2_2',
        python_callable=correct_sleeping_function,
        retries=3,
    )

    t2_3 = PythonOperator(
        task_id='t2_3',
        python_callable=correct_sleeping_function,
    )

    t3_1 = PythonOperator(
        task_id='t3_1',
        python_callable=correct_sleeping_function,
    )

    t3_2 = PythonOperator(
        task_id='t3_2',
        python_callable=correct_sleeping_function,
    )

    t4_1 = BashOperator(
        task_id='t4_1',
        bash_command='sleep 2',
        retries=3,
    )
```

```
def correct_sleeping_function():
    """This is a function that will run within the DAG execution"""
    time.sleep(2)
```

Operators

1. **PythonOperator**
2. **BashOperator**
3. `branch_operator`
4. `email_operator`
5. `mysql_operator`
6. `DataprocedureOperator`

...

...

PythonOperator:

```
def function():  
    print(123)
```

```
task = PythonOperator(  
    task_id='task_id',  
    python_callable=function,  
)
```

BashOperator:

```
task = BashOperator(  
    task_id='task_id',  
    bash_command='sleep 2',  
)
```

other examples

```
bash_command='python python_code.py'
```

```
bash_command='bash bash_code.sh '
```

(must have a space to
satisfy Jinja template !!)

Dependencies

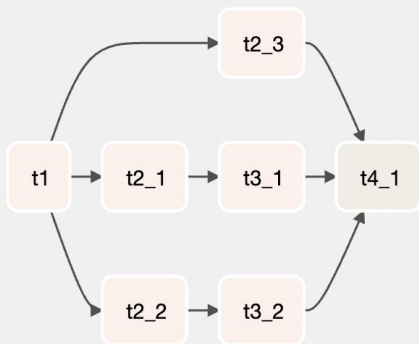
```
# task dependencies
```

```
t1 >> [t2_1, t2_2, t2_3]
```

```
t2_1 >> t3_1
```

```
t2_2 >> t3_2
```

```
[t2_3, t3_1, t3_2] >> t4_1
```



t2_1 will depend on t1 running
successfully to run. The following ways
are equivalent:

```
t1 >> t2_1
```

```
t1 << t2_1
```

```
t1.set_downstream(t2_1)
```

```
t2_1.set_upstream(t1)
```

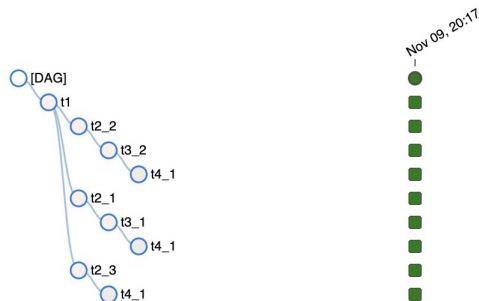
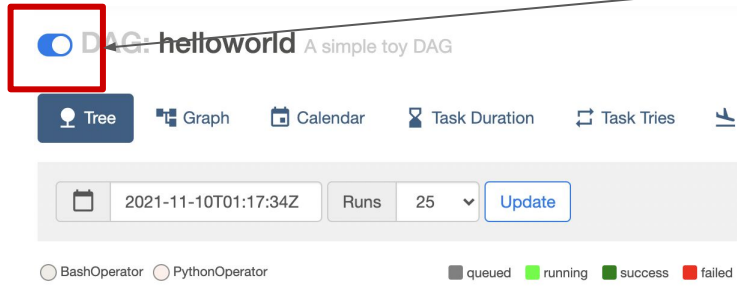
you can write in a chain

```
t1 >> t2_1 >> t3_1 >> t4_1
```

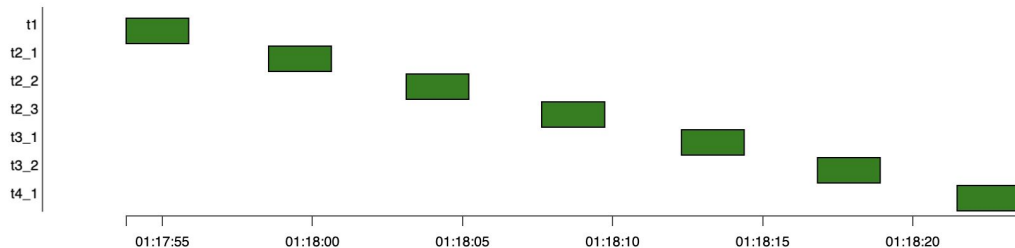
Trigger the dag

```
schedule_interval=timedelta(days=1),  
start_date=datetime(2021, 1, 1),
```

Start scheduling the DAG



Tree

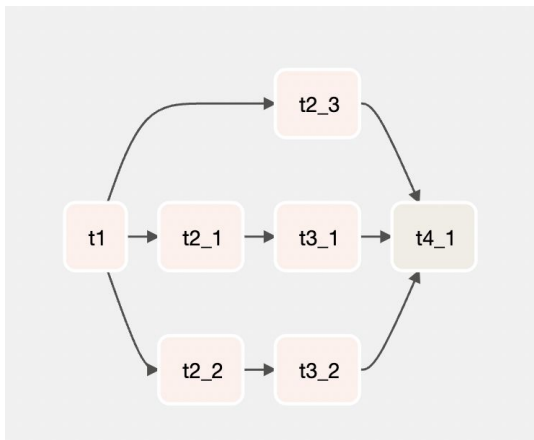


Gantt

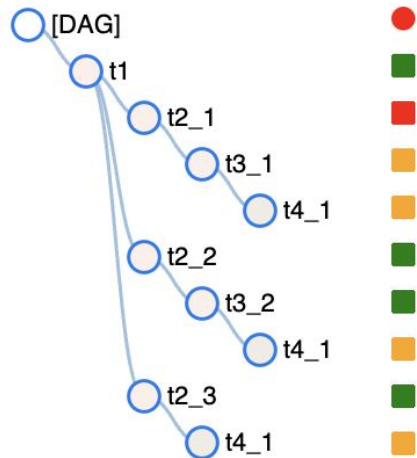
Example 2

```
t1 = PythonOperator(  
    task_id='t1',  
    python_callable=count_function,  
)  
  
t2_1 = PythonOperator(  
    task_id='t2_1',  
    python_callable=wrong_sleeping_function,  
    retries=3,  
)
```

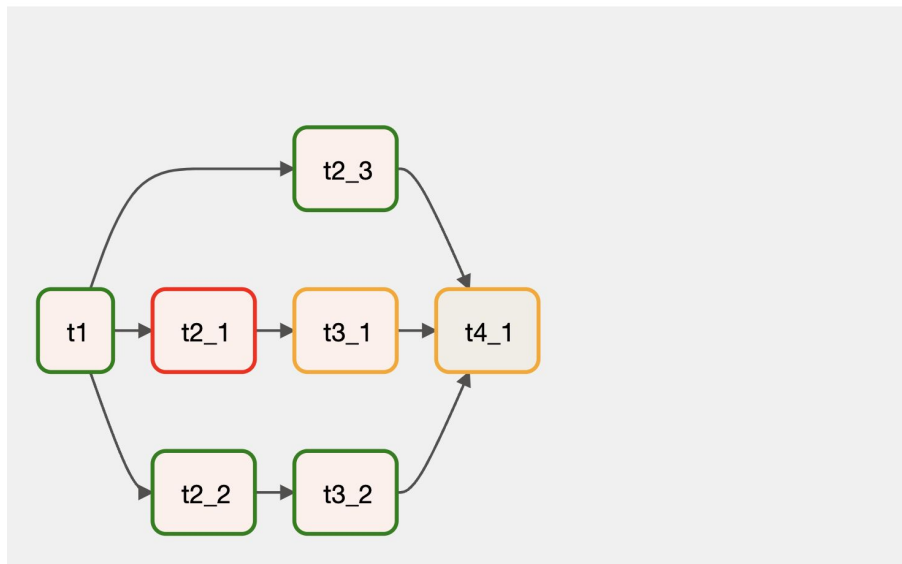
```
count = 0  
  
def count_function():  
    # this task is t1  
    global count  
    count += 1  
    print('count_increase output: {}'.format(count))  
    time.sleep(2)  
  
def wrong_sleeping_function():  
    # this task is t2_1, t1 >> t2_1  
    global count  
    print('wrong sleeping function output: {}'.format(count))  
    assert count == 1  
    time.sleep(2)
```



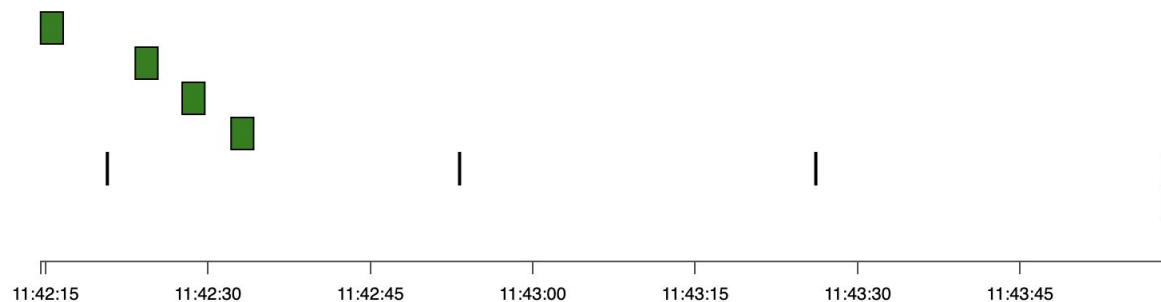
Example 2



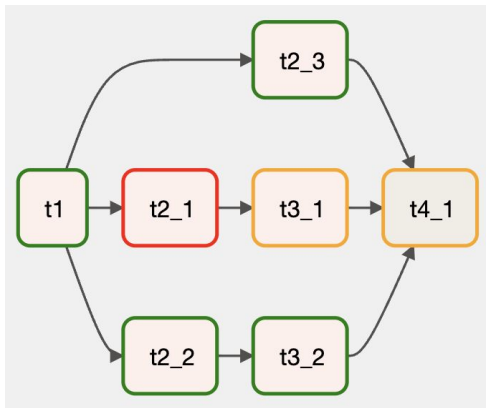
queued running success failed up_for_retry up_for_reschedule upstream_failed



t1
t2_2
t2_3
t3_2
t2_1
t3_1
t4_1



Example 2



```
count = 0

def count_function():
    # this task is t1
    global count
    count += 1
    print('count_increased output: {}'.format(count))
    time.sleep(2)

def wrong_sleeping_function():
    # this task is t2_1, t1 >> t2_1
    global count
    print('wrong sleeping function output: {}'.format(count))
    assert count == 1
    time.sleep(2)
```

```
{logging_mixin.py:109} INFO - count_function output: 1
```

```
...
```

```
{logging_mixin.py:109} INFO - wrong_sleeping_function output: 0
```

```
...
```

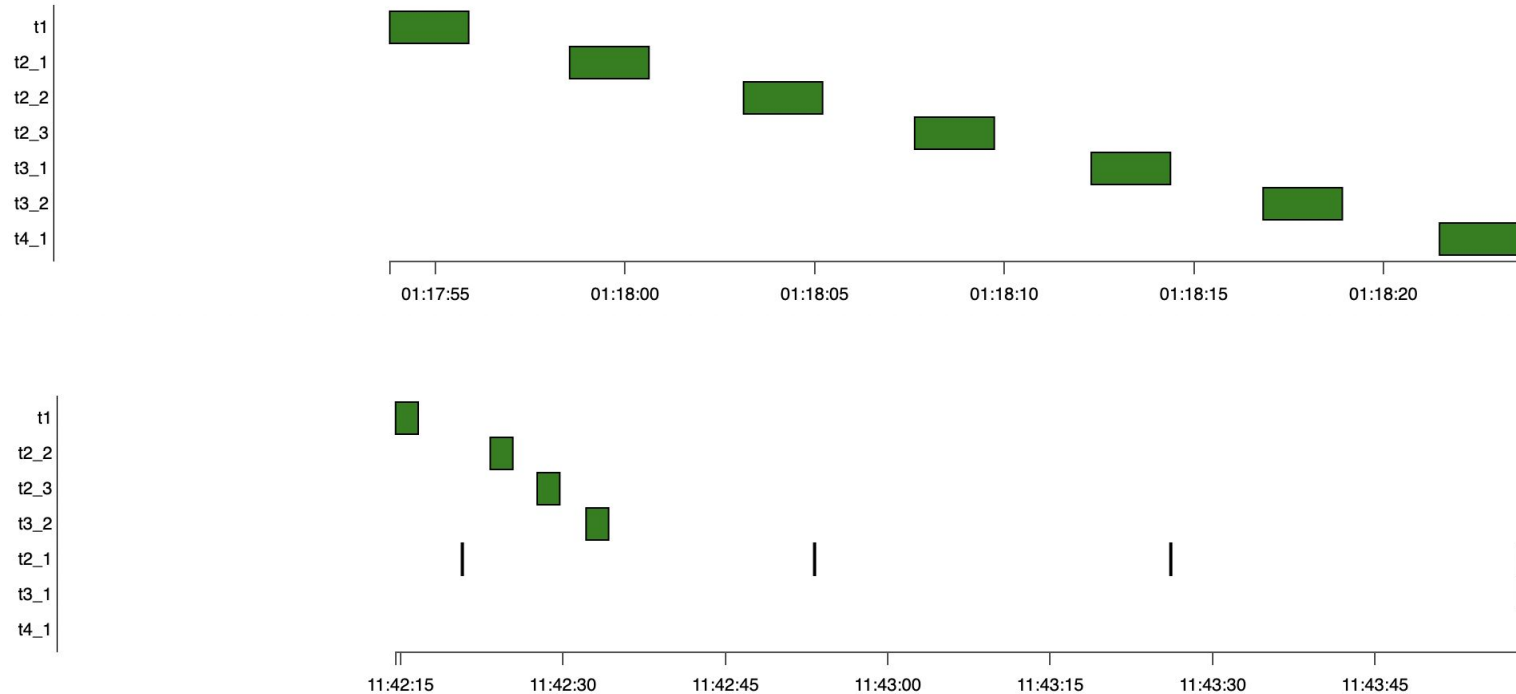
```
assert count == 1
```

```
AssertionError
```


Why?

- Airflow Python script is just a configuration file specifying the DAG's structure as code.
- Different tasks run on different workers at different points in time
- Script cannot be used to cross communicate between tasks (Xcoms can)

Why sequential?



Executors

- **SequentialExecutor**
- **LocalExecutor**
- CeleryExecutor
- KubernetesExecutor

Change SequentialExecutor to LocalExecutor

- Postgresql
- Modify the configuration in `~/airflow/airflow.cfg`

```
# Default timezone in case supplied date times are naive
# can be utc (default), system, or any IANA timezone string (e.g. Europe/Amsterdam)
default_timezone = utc

# The executor class that airflow should use. Choices include
# ``SequentialExecutor``, ``LocalExecutor``, ``CeleryExecutor``, ``DaskExecutor``,
# ``KubernetesExecutor``, ``CeleryKubernetesExecutor`` or the
# full import path to the class when using a custom executor.
executor = SequentialExecutor

# The SQLAlchemy connection string to the metadata database.
# SQLAlchemy supports many different database engines.
# More information here:
# http://airflow.apache.org/docs/apache-airflow/stable/howto/set-up-database.html#database-uri
sql_alchemy_conn = sqlite:///home/ch3212/airflow/airflow.db
```

LocalExecutor

1. Install Postgres and Configure your Postgres user:

In your terminal

```
sudo apt-get install postgresql postgresql-contrib
```

```
sudo -u postgres psql
```

Next enter:

```
ALTER USER postgres PASSWORD 'postgres';
```

```
\q
```

In your terminal

```
pip install 'apache-airflow[postgres]
```

LocalExecutor

2. Change your SQL Alchemy Conn inside airflow.cfg:

In your terminal

```
vim ~/airflow/airflow.cfg
```

you will see this line

```
sql_alchemy_conn = sqlite:///.../airflow.db
```

change it to :

```
sql_alchemy_conn = postgresql+psycopg2://postgres:postgres@localhost/postgres
```

Also change

```
Executor = SequentialExecutor
```

To

```
Executor = LocalExecutor
```

LocalExecutor

3. Check for DB connection:

In your terminal

```
airflow db check
```

If Airflow could successfully connect to yours Postgres DB, you will see an INFO #
containing a “Connection Successful” message in it, so now we are good to go.

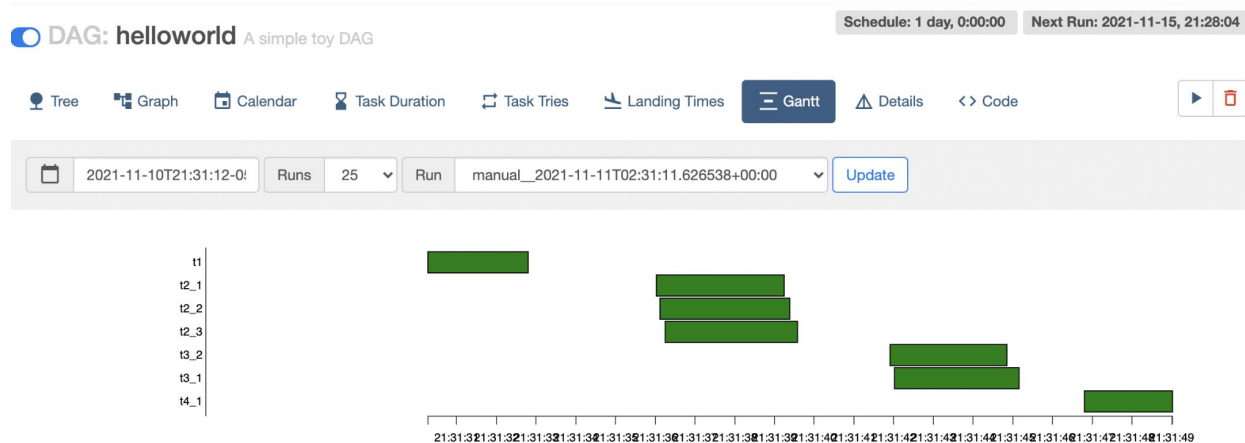
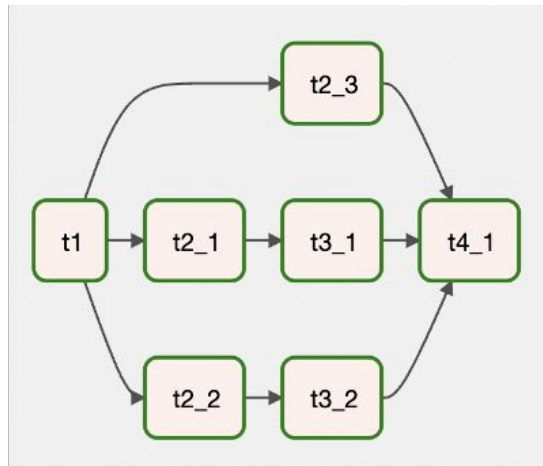
4. Close the webserver and scheduler; Init your Airflow DB:

In your terminal

```
airflow db init
```

5. Create new user and restart the webserver and scheduler as shown in page 18 and 19; and login on the web browser

LocalExecutor



Take home

- DAG
 - Scheduler
 - Executor
 - Database
 - Operator
-
- Cross communication between tasks
 - Schedule a job !! start data and schedule interval

Homework

Three tasks

- Helloworld
- Build workflows
- Written parts

Task 1 Helloworld

Q1.1 Install Airflow (20 pts)

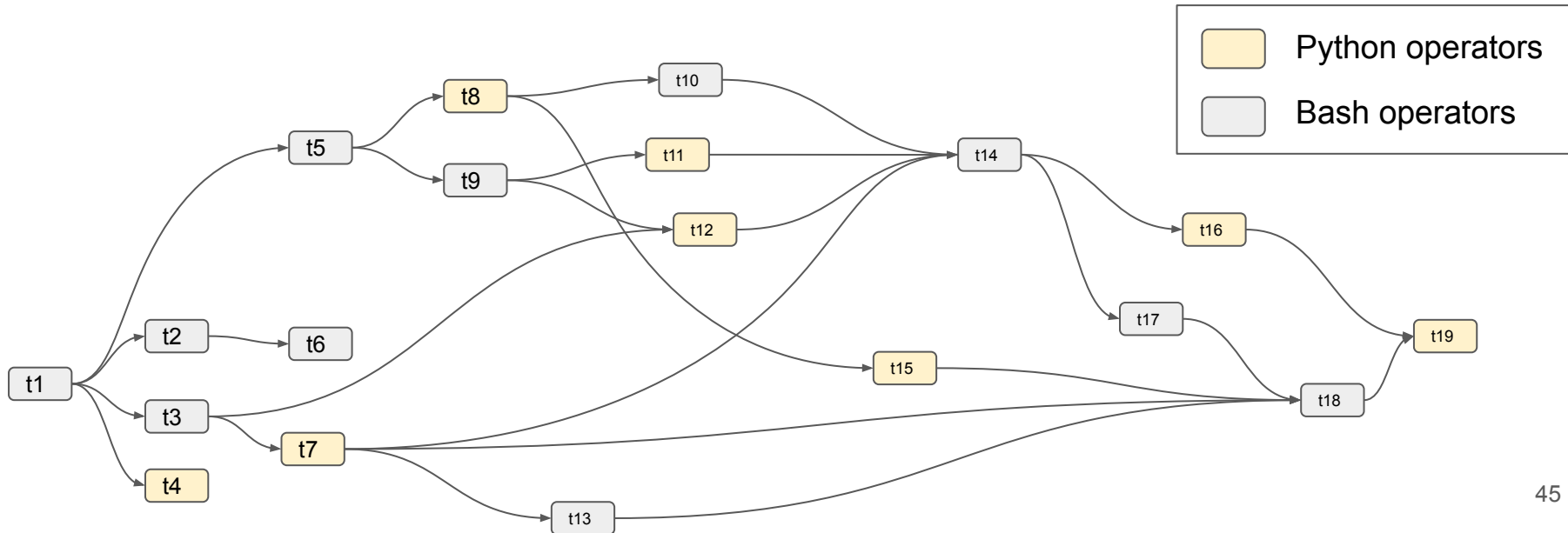
Q1.2 Run helloworld (15 pts)

- SequentialExecutor (5 pts)
- LocalExecutor (5 pts)
- Explore other features and visualizations you can find in the Airflow UI. Choose two features/visualizations to explain their functions and how they help monitor and troubleshoot the pipeline, use helloworld as an example. (5 pts)

Task 2 Build workflows

Q2.1 Implement this DAG (25 pts)

- Tasks and dependencies (10 pts)
- Manually trigger it (10 pts)
- Schedule the first run immediately and running the program every 30 minutes (5 pts)



Task 2 Build workflows

Q2.2 Stock price fetching, prediction, and storage every day (25 pts)

- Schedule fetching the stock price of [AAPL, GOOGL, FB, MSFT, AMZN] at 7:00 AM everyday.
- Preprocess data if you think necessary
- Train/update 5 linear regression models for stock price prediction for these 5 corporates. Each linear model takes the “open price”, “high price”, “low price”, “close price”, “volume” of the corporate in the past ten days as the features and predicts the “high price” for the next day.
- Everyday if you get new data, calculate the relative errors, i.e., (prediction yesterday - actual price today) / actual price today, and update the date today and 5 errors into a table, e.g., a csv file.

```
import yfinance as yf

ticker = 'AAPL'
aapl = yf.Ticker(ticker)
hist = aapl.history(period='max')
print(type(hist))
print(hist.shape)
print(hist)

<class 'pandas.core.frame.DataFrame'>
(10317, 7)
```

	Open	High	Low	Close	Volume \
Date					
1980-12-12	0.100453	0.100890	0.100453	0.100453	469033600
1980-12-15	0.095649	0.095649	0.095213	0.095213	175884800
1980-12-16	0.088661	0.088661	0.088224	0.088224	105728000
1980-12-17	0.090408	0.090845	0.090408	0.090408	86441600
1980-12-18	0.093029	0.093466	0.093029	0.093029	73449600
...
2021-11-04	151.359097	152.207849	150.420465	150.740005	60394600
2021-11-05	151.889999	152.199997	150.059998	151.279999	65414600
2021-11-08	151.410004	151.570007	150.160004	150.440002	55020900
2021-11-09	150.199997	151.429993	150.059998	150.809998	56787900
2021-11-10	150.020004	150.130005	147.850006	147.919998	65076700

Task 3 Written parts

Q3.1 Answer the question (5 pts)

- What are the pros and cons of SequentialExecutor, LocalExecutor, CeleryExecutor, KubernetesExecutor?

Q3.2 Draw the DAG of your group project (10 pts)

- Formulate it into at least 5 tasks
- Task names (functions) and their dependencies
- How do you schedule your tasks?

References

<https://airflow.apache.org/docs/apache-airflow/stable/index.html>

<https://cloud.google.com/composer/docs>