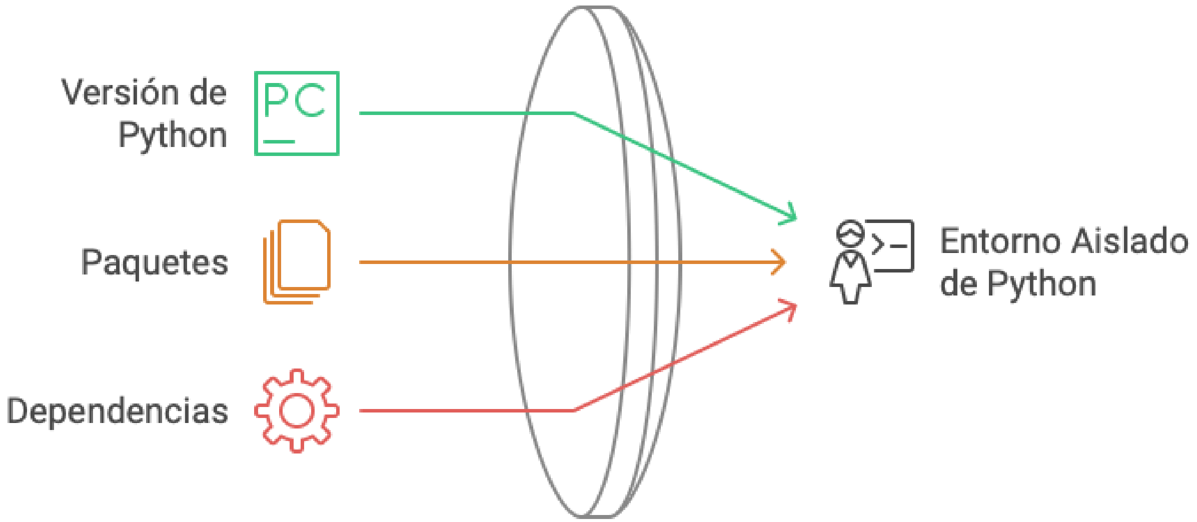
anyoneAI

Hello!

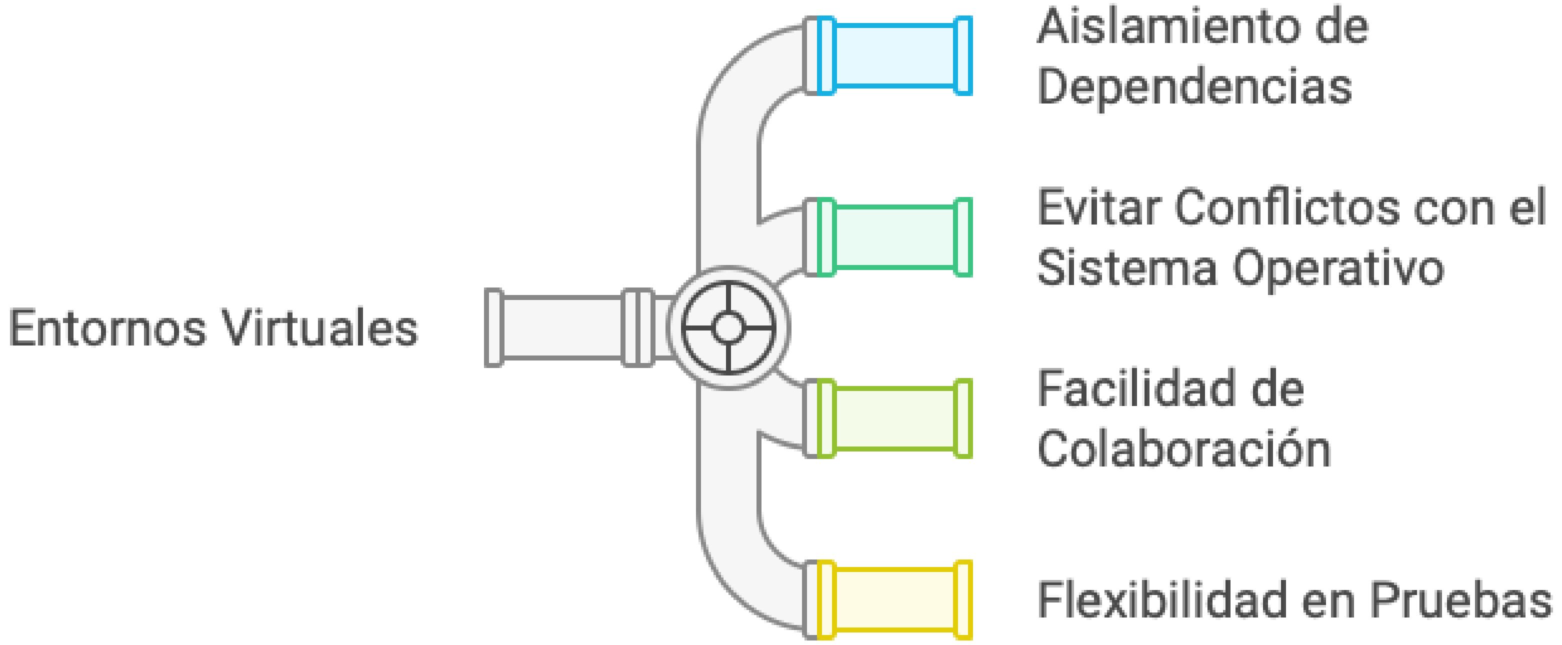
Ambientes Virtuales

Motivación

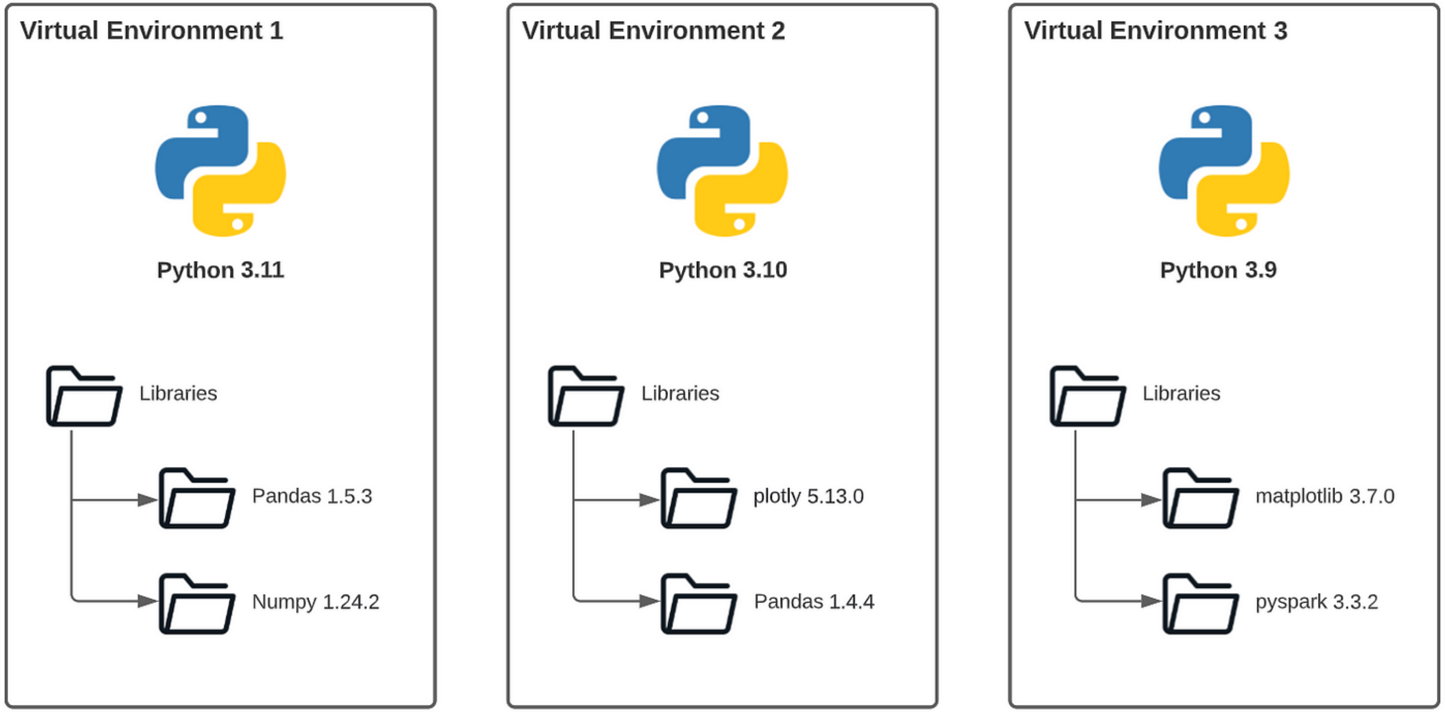
Un ambiente virtual de Python es un entorno aislado en el que se puede ejecutar una versión específica de Python junto con sus paquetes y dependencias, sin interferir con otros proyectos o con las configuraciones del sistema operativo.



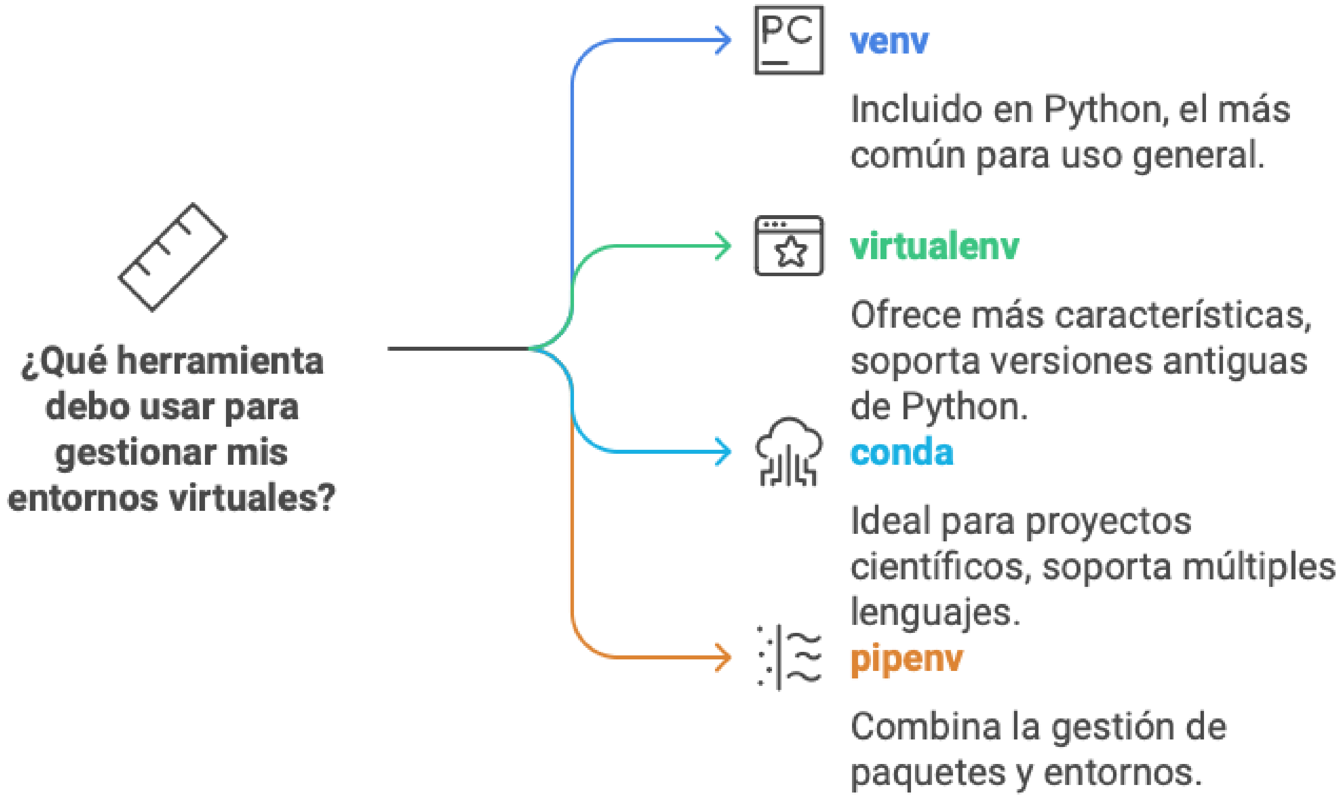
¿ Para qué sirven los ambientes virtuales ?


Entornos Virtuales
Aislamiento de Dependencias
Evitar Conflictos con el Sistema Operativo
Facilidad de Colaboración
Flexibilidad en Pruebas

¿ Para qué sirven los ambientes virtuales ?



Tipos de ambientes virtuales



¿Qué herramienta debo usar para gestionar mis entornos virtuales?

venv

Incluido en Python, el más común para uso general.

virtualenv

Ofrece más características, soporta versiones antiguas de Python.

conda

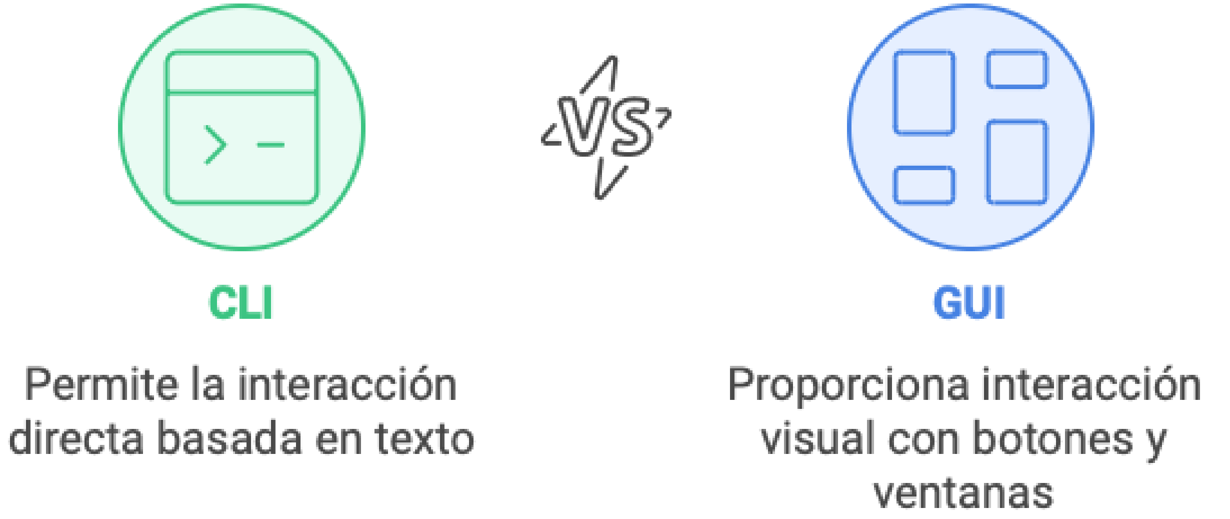
Ideal para proyectos científicos, soporta múltiples lenguajes.

pipenv

Combina la gestión de paquetes y entornos.

CLI (Command Line Interface)

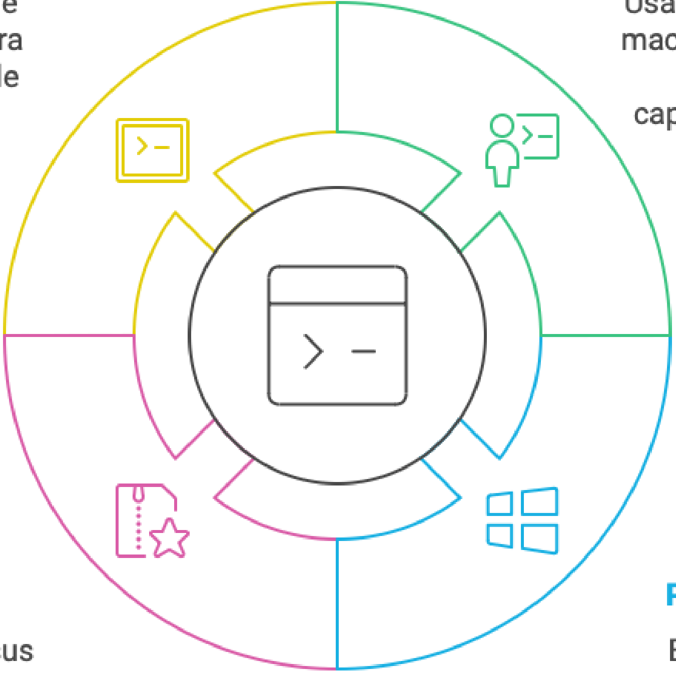
La **CLI (Command Line Interface),** o interfaz de línea de comandos, es un método de interacción entre el usuario y un sistema informático mediante comandos de texto. A diferencia de las interfaces gráficas **(GUI),** donde se usan botones y ventanas, en una **CLI** el usuario escribe directamente instrucciones en un terminal o consola.


CLI
Permite la interacción directa basada en texto
VS
GUI
Proporciona interacción visual con botones y ventanas

anyoneAI

Tipos de CLI

Cada CLI tiene sus propias ventajas y casos de uso.



Commnad Prompt (CMD)

CLI básico de Windows para comandos de archivos y sistema

Bash

Usado en Linux y macOS, conocido por sus capacidades de scripting

Zsh

Popular por sus características interactivas y personalización

PowerShell

Basado en Windows, se integra con .NET para scripting avanzado

Comparativa de comando básicos

Función	CMD	PowerShell	Bash	Zsh
Listar archivos	dir	Get-ChildItem o ls	ls	ls
Cambiar directorio	cd	cd	cd	cd
Copiar archivos	copy	Copy-Item	cp	cp
Mover archivos	move	Move-Item	mv	mv
Eliminar archivos	del	Remove-Item	rm	rm
Crear directorio	mkdir	New-Item -ItemType Directory	mkdir	mkdir
Eliminar directorio	rmdir	Remove-Item	rm -r	rm -r
Ver contenido de un archivo	type	Get-Content	cat	cat
Limpiar pantalla	cls	Clear-Host o cls	clear	clear
Buscar texto en archivos	No disponible	Select-String	grep	grep
Renombrar archivos	rename	Rename-Item	mv	mv
Mostrar IP configuraciones	ipconfig	Get-NetIPAddress	ifconfig (o ip a)	ifconfig (o ip a)
Salir del terminal	exit	exit	exit	exit
Mostrar directorio actual	cd	Get-Location	pwd	pwd

Creación de un ambiente Virtual

1. Crear un ambiente virtual

```
$ python -m venv <nombre_del_entorno>
```

2. Activarlo

```
$ <nombre_del_entorno>\Scripts\actívate.      (Windows)

$ source <nombre_del_entorno>\bin\activate   (macOS/Linux)
```

3. Instalar paquetes (dentro de un ambiente virtual):

```
$ pip install <paquete>
```

4. Desactivarlo

```
$ deactivate
```

Motivation

We'll need to break down our data pipeline into individual workflows that can be orchestrated as needed.

There are several tools that can help us so this such as Airflow, Prefect, Dagster, Luigi, etc.

We'll be creating our workflows using AirFlow for its:

- Wide adoption of the open source platform in industry.
- Python based software development kit (SDK).
- Ability to run locally and scale easily.
- Maturity over the years and part of the apache ecosystem.

Airflow

Apache Airflow is an open-source platform to run any type of workflow.

Airflow is using the Python programming language to define the pipelines. Users can take full advantage of that by using for loop to define pipelines, executing bash commands, using any external modules like pandas, scikit-learn and much more.

Apache Airflow can be used to schedule:

- ELT pipelines that extract data from multiple sources and apply data transformations
- Training machine learning models
- Report generation
- Backups and similar DevOps operations

And much more!

Airflow setup

You can install it with **`pip`** by simply running:

```
$ pip install apache-airflow==2.5.0
```

And check it was correctly installed with:

```
$ airflow version
```

The output should be:

```
2.5.0
```

Airflow setup

Then, you must configure the Airflow home folder in your machine, ~/airflow is the default

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

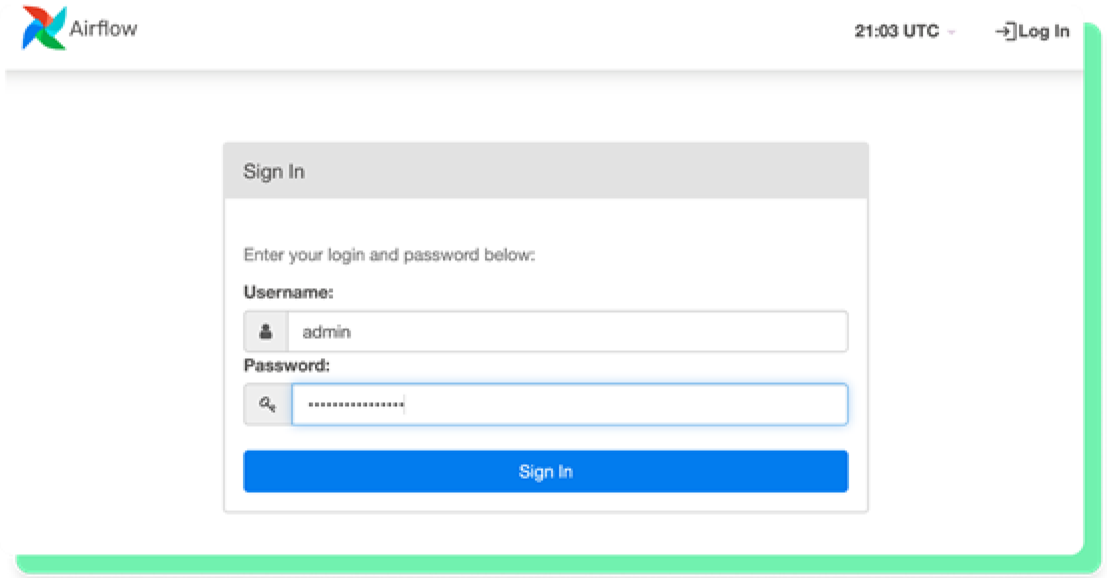
You will need to initialize the database and create an user/password, you can do all this and more simply running:

```
$ airflow standalone
```

Now visit `localhost:8080` in the browser and use the admin account details shown on the terminal to login.

Airflow setup

You will see a UI like the following:



Enter your login and password below:

Username:

admin

Password:

················

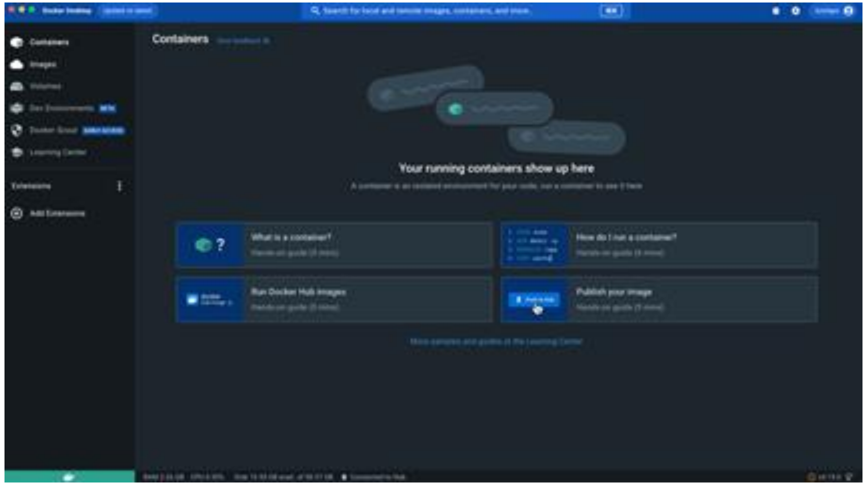
anyoneAI

Airflow setup using Docker for Windows users

Install Docker Desktop from the official site

https://www.docker.com/products/docker-desktop/

Open Docker Desktop, it should looks like that



Your running containers show up here

Airflow setup using Docker for Windows users

Verify the installation of Docker from your terminal

```
$ Docker --version
```

The output should be something like:

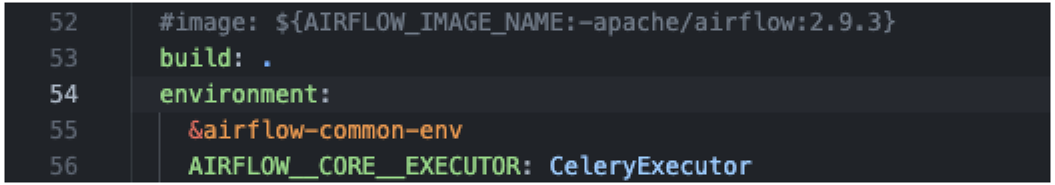
```
Docker versión 23.0.5, build bc447a
```

Airflow setup using Docker for Windows users

Download the *docker-compose.yaml* file using the following command in the terminal:

```
$ curl -LfO 'https://airflow.apache.org/docs/apache-airflow/2.9.3/docker-compose.yaml'
```

Modify the *docker-compose.yaml* file by commenting out the line that select the image and keeping the build line, like this



```
#image: ${AIRFLOW_IMAGE_NAME:-apache/airflow:2.9.3}
build: .
environment:
  &airflow-common-env
  AIRFLOW__CORE__EXECUTOR: CeleryExecutor
```

Airflow setup using Docker for Windows users

Create a requirements.txt file with the necessary libraries.

```
db-sqlite3==0.0.1
matplotlib==3.6.2
pandas==1.5.2
plotly_express==0.4.1
requests==2.26.0
seaborn==0.11.2
SQLAlchemy==1.4.45
nbformat==5.7.3
pytest==7.2.1
```

Airflow setup using Docker for Windows users

Create an *.env* file using with the next variables

```
    AIRFLOW_IMAGE_NAME=apache/airflow:2.9.3
AIRFLOW_UID=50000
```

Create a Dockerfile that contains

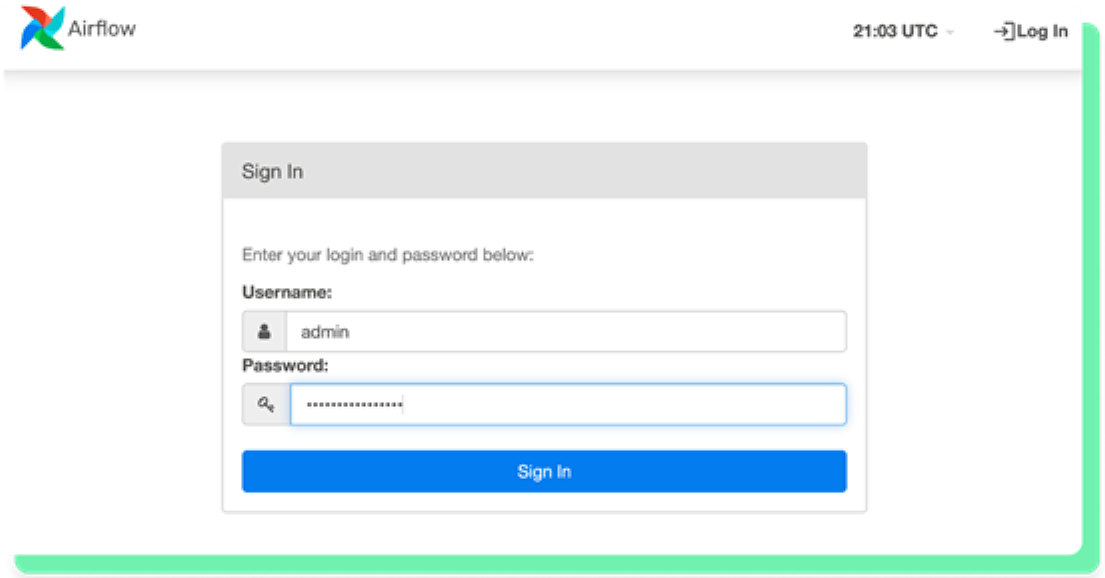
```
FROM apache/airflow:2.9.3
ADD requirements.txt .
RUN pip install apache-airflow==${AIRFLOW_VERSION} -r requirements.txt
```

Run the Docker container using

```
$ docker-compose up
```

Airflow setup

Now visit `localhost:8080` in the browser and use the admin account details shown on the terminal to login. You will see a UI like the following:


Airflow

21:03 UTC

Log In

Sign In

Enter your login and password below:

Username:

admin

Password:

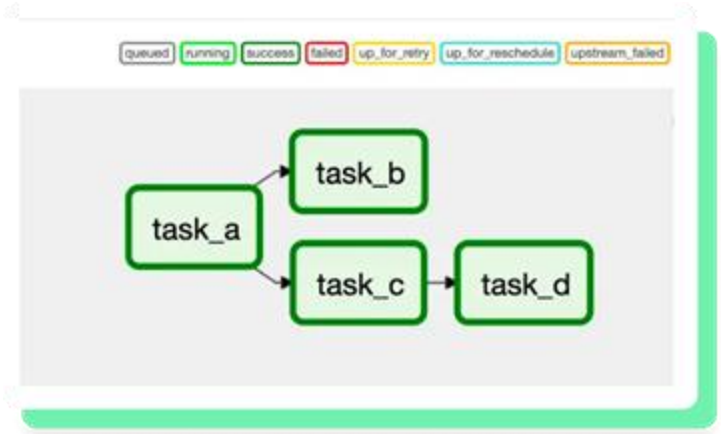
•••••••••••••••••

Sign In

Airflow DAGs

Workflows are defined in Airflow by DAGs (Directed Acyclic Graphs) and are nothing more than a Python file.

A single DAG file may contain multiple DAG definitions, although it is recommended to keep one DAG per file.



Airflow DAGs

Let’s take a look at example DAG:

```
import pendulum

from airflow.models import DAG

with DAG(
   "elt_sales_daily",
   start_date=pendulum.datetime(2021, 1, 1),
   schedule=None,
) as dag:
   ...
```

First of all, DAG is identified by unique **`dag_id`** which has to be unique in whole Airflow deployment.

Additionally, to create a DAG we need to specify:

- **`start_date`**: a date (datetime object) from which the DAG will start running. This helps run a DAG for past dates. It is common to use the `pendulum.datetime()` function to specify this value. If the date is in the future you can still trigger the dag manually.

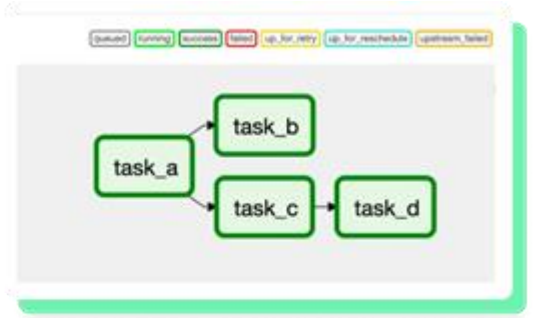
- **`schedule`**: which defines when the DAG should be run. It can be timedelta object for example `timedelta(days=2)` or a `string cron expression * * * * *`. It can be `None` and then the DAG will not be scheduled by Airflow but it can still be triggered manually or externally.

Airflow Tasks

```
import pendulum
from airflow.models import DAG
from airflow.operators.dummy_operator import
DummyOperator

with DAG(
  "elt_sales_daily",
  start_date=pendulum.datetime(2021, 1, 1),
  schedule=None,
) as dag:
  task_a = DummyOperator(task_id="task_a")
  task_b = DummyOperator(task_id="task_b")
  task_c = DummyOperator(task_id="task_c")
  task_d = DummyOperator(task_id="task_d")

  task_a >> [task_b, task_c]
  task_c >> task_d
```

Task creation and definition happens in the zone marked in **Blue**.

Tasks dependencies inside the **Light Cyan** rectangle.



Operators

An Operator is conceptually a template for a predefined Task, that you can just define declaratively inside your DAG:

Airflow has a very extensive set of operators available, with some built-in to the core or pre-installed providers. Some popular operators from core include:

- **`BashOperator`**: executes a bash command.
- **`PythonOperator`**: calls an arbitrary Python function.
- **`EmailOperator`**: sends an email.

You can also find other operators from a huge set of community provider packages. Some popular operators from here include:

- **`MySqlOperator`**
- **`DockerOperator`**
- **`SlackAPIOperator`**

`See this DAG Example using BashOperator`

XComs

XComs (short for “cross-communications”) are a mechanism that let Tasks talk to each other, as by default Tasks are entirely isolated and may be running on entirely different machines.

An XCom is identified by a key (essentially its name), as well as the **`task_id`** and **`dag_id`** it came from. They can have any (serializable) value, but they are only designed for small amounts of data; do not use them to pass around large values, like dataframes.

XComs are explicitly “pushed” and “pulled” to/from their storage using the xcom_push and xcom_pull methods on Task Instances.

`See this DAG Example using XComs`

TaskFlow

A new way of creating tasks, called TaskFlow, was introduced in the Airflow 2.0 release.

If you write most of your DAGs using plain Python code rather than Operators, then using the TaskFlow API will make it much easier to author clean DAGs without extra boilerplate, all using the **`@task`** decorator like this:

```
@task()
def extract():
  data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'
  order_data_dict = json.loads(data_string)

  return order_data_dict
```

TaskFlow

TaskFlow also takes care of moving inputs and outputs between your Tasks using XComs for you, as well as automatically calculating dependencies.

When you call a TaskFlow function in your DAG file, rather than executing it, you will get an object representing the XCom for the result (an XComArg), that you can then use as inputs to downstream tasks or operators.

```
@task()
def extract():
  data_string = '{"1001": 301.27, "1002": 433.21}'
  order_data_dict = json.loads(data_string)
  return order_data_dict

@task()
def transform(order_data_dict: dict):
  total_order_value = 0
  for value in order_data_dict.values():
    total_order_value += value

  return {"total_order_value": total_order_value}

order_data = extract()
order_summary = transform(order_data)
```

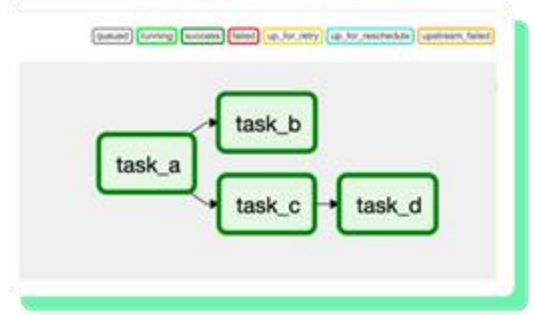
Airflow Tasks

```
import pendulum
from airflow.models import DAG
from airflow.operators.dummy_operator import
DummyOperator

with DAG(
  "elt_sales_daily",
  start_date=pendulum.datetime(2021, 1, 1),
  schedule=None,
) as dag:
  task_a = DummyOperator(task_id="task_a")
  task_b = DummyOperator(task_id="task_b")
  task_c = DummyOperator(task_id="task_c")
  task_d = DummyOperator(task_id="task_d")

  task_a >> [task_b, task_c]
  task_c >> task_d
```

A **Task** is the basic unit of execution in Airflow.

Tasks are arranged into DAGs, and then have upstream and downstream dependencies set between them into order to express the order they should run in.

Every task in a DAG is defined by the Operator (we will dive into more details soon) and has its own **task_id** that has to be unique within a DAG.


queued
running
success
failed
up_for_retry
up_for_reschedule
upstream_failed
task_a
task_b
task_c
task_d