Automation

Automation has played a major role in our society. From the Second Industrial Revolution, when factories were automated using machines, to the Green Revolution in India, where along with other things, automation was adopted in farming to scale up food production, we can see plenty of examples of automation bringing about revolutionary changes.

- When should you automate?
  We should automate whenever the following conditions are met:
  - Repetitive: The process that we are trying to automate is repetitive.
  - Non-unique: The process must be the same every time. If the process changes every time, we cannot automate it.
- What are the advantages of automation?
  - Streamlined process
  - Reduced errors
  - Reduced costs
- What are the types of automation in software engineering?
  - **Schedule-based automation**: Schedule-based automation allows us to automatically trigger a process at a fixed interval of time irrespective of external factors. An example of this would be an Amazon user getting emails daily about the offers applicable on products on that particular day.
  - **Event-based automation**: Event-based automation allows us to automatically trigger a process based on some external factors. An example of this would be an Amazon user getting emails to reset a password whenever the user forgets their password and clicks on “forgot password” while logging into their account.

Apache Airflow

Apache Airflow is a tool for building complex pipelines in the form of directed acyclic graphs. Apache Airflow is a versatile tool using which you can:

- **Create pipelines**: Using Apache Airflow, you can create complex pipelines easily. You can create pipelines using tasks and dependencies. These tasks and dependencies when put together in an acyclic manner form directed acyclic graphs.
- **Manage pipelines**: Using Apache Airflow, you can easily manage the pipelines created. This means that using Apache Airflow, you can schedule or trigger your pipeline based on some events. Recall that in software engineering, two types of automation are predominantly used, schedule-based and event-based. Apache Airflow can be used to perform both types of automation. Apache Airflow can backfill. Let’s understand backfill using the following example. There can be the case when you may want to run the DAG for a specified historical period e.g., A data filling DAG is created with start_date

2019-11-21, but another user requires the output data from a month ago i.e., 2019-10-21. In such a case you run your pipeline to get the data for the required date. This process is known as Backfill. You can also manually trigger the pipelines. This is particularly useful when you are debugging our pipeline.

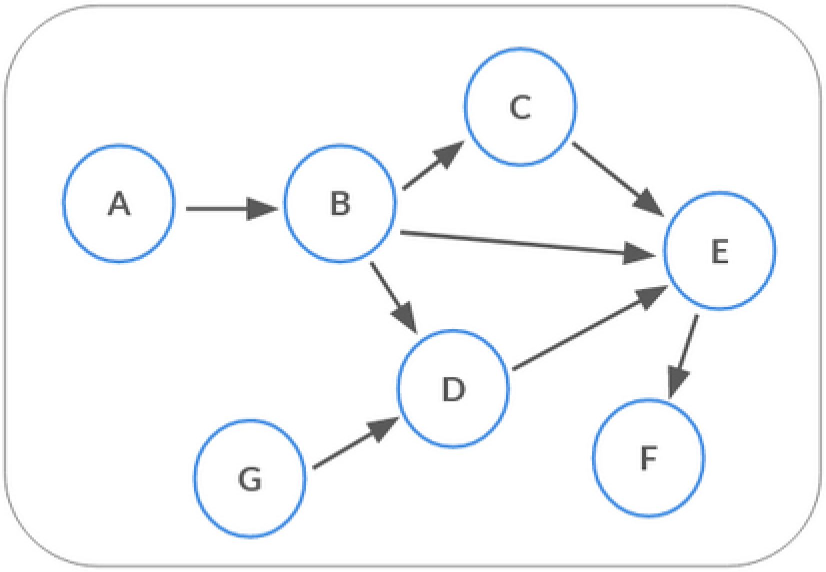
- **Monitor pipelines**: Apache Airflow has inbuilt features that help you in monitoring a pipeline's run. Recall that in order to create a production-level script, you had to include logging and exception handling in our code among other things. In Apache Airflow, you do not have to include these manually. Apache Airflow automatically creates logs and handles runtime errors by retrying failed tasks. Because of the aforementioned reasons, Apache Airflow is a widely adopted tool in the industry and is used by many well-known companies such as Adobe, Airbnb and Walmart.

Directed Acyclic Graphs

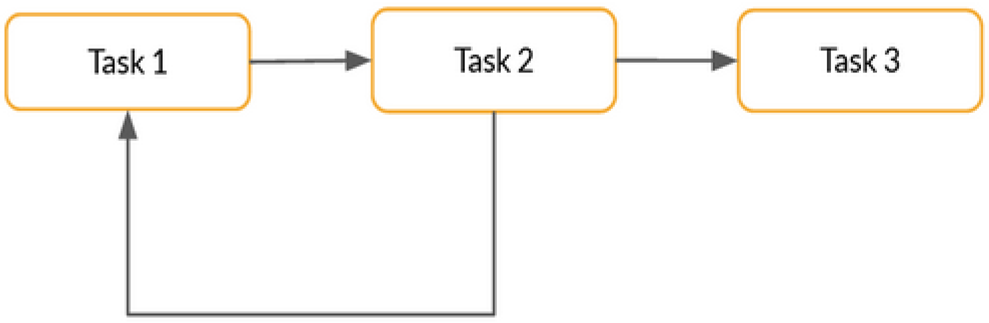
Any flow chart can be a DAG if it has the following three conditions:

- **Tasks**: A DAG is a collection of tasks/nodes. Eeach function in our modularised code will constitute a task. In some places, you will find that people call this a node, which is nothing but its mathematical name. For our use, you will simply call this a task.

  In the following figure, you can see that each box represents a task. You have seven tasks: A, B, C, D, E and F.



- **Direction/Dependency/Relation**: The arrows in this figure represent direction/dependency. As the name suggests, directed acyclic graphs must have a direction in which tasks are executed.
- **Acyclicity**: The most important constraint put on DAGs is acyclicity, meaning that no two tasks must have a cyclic relation. In the figure given above, you can see that tasks are not connected in a cyclic manner.
  Given below is an example of a set of tasks with defined directions. It is not a DAG, as Task-1 and Task-2 are arranged in a cyclic manner.

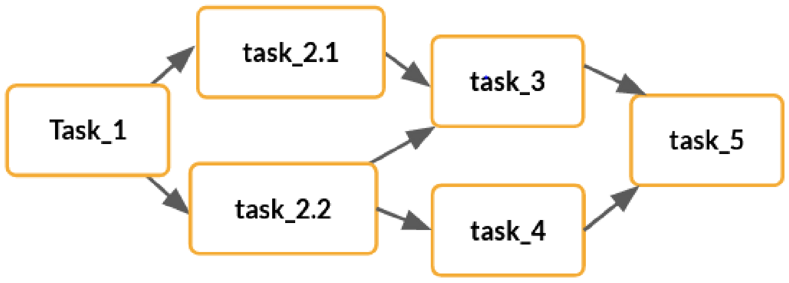


Not a DAG

In this diagram, you can see that Task-1 and Task-2 depend on inputs from each other. This creates a paradox in which we cannot run Task-1 until Task-2 is completed, and vice versa. Thus, we have an issue where we cannot decide the starting point of execution of the DAG. Hence, acyclicity is an important constraint of DAGs.

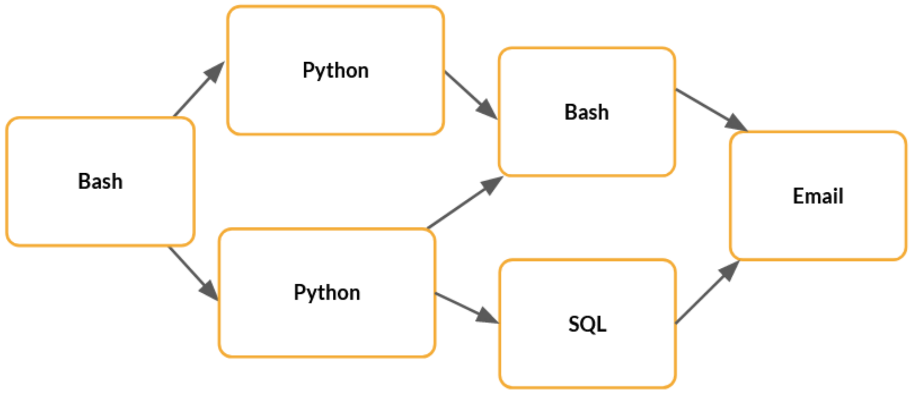
Tasks

- **Tasks are the basic units of execution:** Tasks in Apache Airflow are basic units of execution. Usually, there are multiple tasks in a pipeline, which are executed based on their dependencies once the pipeline is triggered.
- **Tasks are independent**: Any two tasks in Apache Airflow are independent of each other. A major implication of this is that we cannot pass a huge amount of information between two tasks. Practically, this means that we need to write the output of each task to a database or file and read it from there for the next task. This is the standard way of passing a huge amount of data between two tasks.
  A sample DAG is given below.



Here, you can see that Task 2.1 and Task 2.2 are dependent on Task 1's output, but due to the independent nature of the tasks, we cannot pass a large amount of information directly between them. You need to write the output of Task 1 in a database or a file, and then read the data from that database or file in Task 2.1 and Task 2.2. This is applied to all tasks, and you write the output of all the tasks to a database or a file and read the input from the database or files.

- **Operators act as templates for defining tasks:**In order to define a task in Airflow, you use operators. Operators are essentially templates that take functions/commands as input. You can construct pipelines with tasks in different languages. This is a major implication of tasks being independent. Tasks being independent means that in a pipeline, we can have one task in, say, Python and another task in, say, Java.



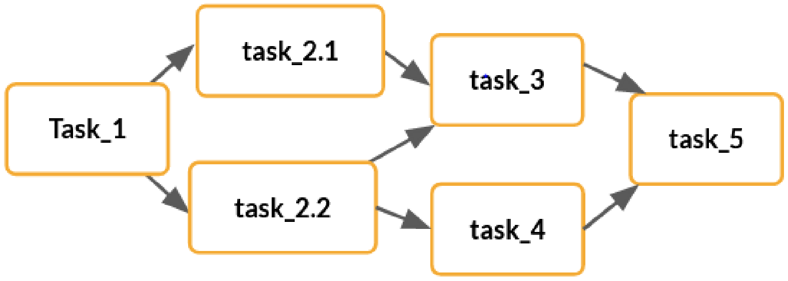
Operators make defining tasks easy. For example, if you want to define a task that creates a directory name 'DIR', you will use the Bash operator to define the task. A code snippet for the same is presented below.

```
bash_task = BashOperator( # Defining that we will be using Bash Command for this task
        task_id="make_directory", #We are giving our task an ID
        bash_command='mkdir "DIR" ' #The bash command that we want the task to execute
)
```

You can clearly see that defining a task in Airflow is simple, and this is facilitated by operators. Operators essentially port the code that you write in any language to Airflow

Relations

There can be three types of relations between tasks.



- **Upstream**: Upstream dependency means that a given task must be completed before the task on which it has an upstream dependency. From the diagram given above, you can see that Task 2.1 needs to be executed before Task 3. This means that Task 2.1 has an upstream dependency on Task 3.
- **Downstream**: Downstream dependency means that a given task can only be executed once the task on which it has a downstream dependency is completed. From the diagram, you can see that Task 3 has a downstream dependency on Task 2.1 and Task 2.2.

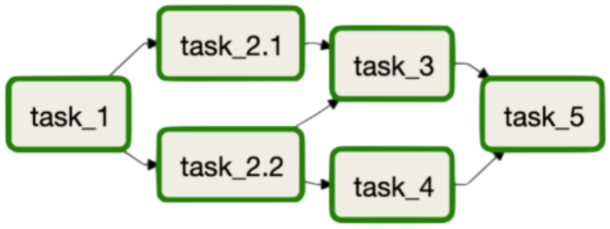
- **Branching**: Branching dependency means that the task will branch out from a given task. From the diagram, you can see that Task 2.2 has a branching dependency on Task 3 and Task 4.

In Airflow, we can use bitwise operators to define relations between tasks.

- Downstream relations are defined using >>.
- Upstream relations are defined using <<.
- Branching relations are defined using [ ].

An example of a pipeline's relations is given below.

```
task_1 >> [task_2_1, task_2_2] >> task_3
task_2_2 >> task_4
[task_3, task_4] >> task_5
```



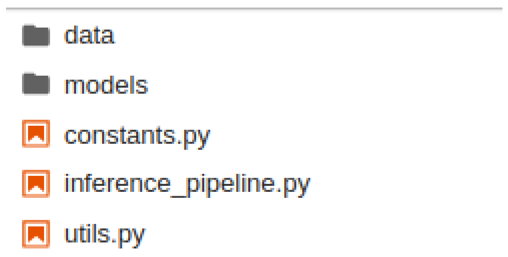
Organising Code

Note that we need to keep all our pipeline-related files in the 'dags' folder, which is present in the 'airflow' folder, else your dag will not be recognised.

We first created a directory named 'heart_disease' inside the dag folder in airflow. We then uploaded data files, saved models, utils and constants files. and organised them.

We did not upload the main file in this case, as the inference_pipeline.py would take care of all the functionalities of the main file and add more functionalities of scheduling, logging and exception handling.

The final structure of the heart_disease folder was



data
models
constants.py
inference_pipeline.py
utils.py

Where

- **data:** Directory contains the inference Data
- **models:** Contains the weights of the models(Ada boost and SVM).
- **constants.py:** Contains all the defined constants
- **utils.py:** Contains all the utility functions
- **inference_pipeline.py:** This is the file where we will define the Airflow pipeline.

Modifying Inference Code

- You modified the functions present in the utils.py file to write their output to CSV files and read their inputs from CSV files. Recall that each time while writing the output to csv files we used **index=False**, this argument does not write the index to the csv files. This is important because, if we do not pass this argument while writing the csv files, we will get new index column each time we write the files
- You removed the **apply_pre_processing function**, as you want to run all the functions independently as tasks. Using this function would mean that a single Airflow task will contain the entire pipeline, which is not desirable.
  Recall that we create an "**inference_pipeline.py**" file. There you will define an Airflow pipeline that will run all the functions in utils.py in the same order but in the form of AIrflow Tasks.
- You created a function **predict_data** that loads the model, passes the data, gets the model's prediction and writes it to a file.
- You modified the **constants.py** file to include all the newly defined constants.

Note that we must import Python files in the following manner:

```
from dir_name.file_name import *
```

Only then will the code work with Airflow, else it will throw an import error saying that the module is not found. In normal Python, “from file_name import *” will work just fine, but that is not the case in Airflow. So, please ensure that whenever you import functions from a custom file in Airflow, you follow this convention.

Defining the Pipeline

You defined some arguments that we will pass while creating an instance of DAG.

```
default_args = {

    'owner': 'airflow',

    'start_date': datetime(2022,7,30),

    'retries' : 1,

    'retry_delay' : timedelta(seconds=30)

}
```

Let’s understand all the default arguments that we have defined in detail:

- **owner**: This is the name of the owner of the pipeline. It can be the name of the company or organisation. We have randomly set this argument to ‘airflow’.
- **start_date**: This is the date from which the scheduling of the pipeline needs to be started. If the start date is a past date, then Airflow will automatically run the pipeline for the previous dates. This is called catchup
- **retries**: This is the number of times a particular task is to be retried once it fails. This argument needs to be a whole number. In our case, we have instructed Airflow to retry failed tasks just once. This is similar to exception handling that you learnt about in the previous modules.
- **retry_delay**: This is the duration to wait before retrying a failed task. We have instructed Airflow to wait five minutes before retrying a failed task.

You learnt how to create an instance of DAG. The following code was used.

```
ML_inference_dag = DAG(

        dag_id = 'Heart_Disease_ML_dag',

        default_args = default_args,

        description = 'Dag to run inferences on predictions of heart disease patients',

        schedule_interval = '@hourly'

)
```

Let's understand all the arguments that we have passed to the DAG class:

- dag_id: It is a unique identifier associated with every DAG in Airflow. It is essentially the name of the DAG. An important point to note is that there should not be any whitespces in the DAG's ID.
  For example: '**Heart Disease ML Dag**' is not a valid dag_id, but '**Heart_Disease_ML_dag**' is. The key takeaway here is that you should use '_' instead of ' ' (space)in your dag_id.
- **default_args:** These are a set of arguments that configure a DAG. Here, we will pass the arguments that we defined earlier in the default_args dictionary.
- **description:** This is a brief description of what your DAG does.
- **schedule_interval:** This specifies the interval at which your DAG will be triggered. We can trigger our DAGs hourly, daily, weekly, monthly or yearly. In our case, we have instructed Airflow to trigger our DAG hourly, as we are getting new inference data hourly.

Defining Tasks and Relation

You learnt how to define a task using PythonOperator.

Let's understand the arguments that we passed to define the task

```
load_task = PythonOperator(
        task_id = 'load_task',
        python_callable = get_inference_data,
        dag = ML_inference_dag)
```

- **task_id**: This is a unique identifier for a task. Essentially, this is the name of the task. Here, we have used 'load_task' as our task_id.
- **python_callable**: This argument takes in the Python function that the task will execute. Since we are using a Python operator, we need to pass in the python function. This is the same function that is present in utils.py.
- **dag**: This is the name of the DAG instance in which we will add this task. Recall that we used the following snippet to instantiate our DAG.

```
ML_inference_dag = DAG(
        dag_id = 'Heart_Disease_ML_dag',
        default_args = default_args,
        description = 'Dag to run inferences on predictions of heart disease patients',
        schedule_interval = '@hourly'
)
```

Ensure that you pass the variables name in the dag argument and not the dag_id.

Code Debugging and Sanity Checks

- We must initialise Airflow's database before running any Airflow command.
- To initialise the database, use the **airflow db** init command. A list of sanity checks that should be performed before finalising the pipeline is given below.

Check	**Airflow Command**
Whether our DAG is showing up in the list of all dags in Airflow. NOTE: *When you run the given command many DAGs will be listed. These DAGs are sample DAGs provided by default in Airflow.*	airflow dags list
Whether the DAG has all the tasks that we defined	airflow task list <dag_id>

Whether the relationship between the tasks defined in the DAG is proper	airflow task list <dag_id> -tree
Whether the individual tasks are running properly	airflow tasks test <dag_id> <task_id> <date>

In order to initialise the Airflow server, we need to run two commands: **airflow scheduler**, which initiates the scheduler in airflow that takes care of scheduling all the pipelines, and **airflow webserver**, which provides a GUI.

Since we are running Airflow on Jarvislabs, we need to ensure that Airflow runs on any one of the available ports. In Jarvislabs, we only have port 6006, 6007, 6008 and 6009 available to us. Thus, we need to specify the port on which we want to run airflow webserver.

For this, the command will be:

```
airflow webserver --port <port number(6006/6007/6008/6009)>
```

Navigating Airflow's UI

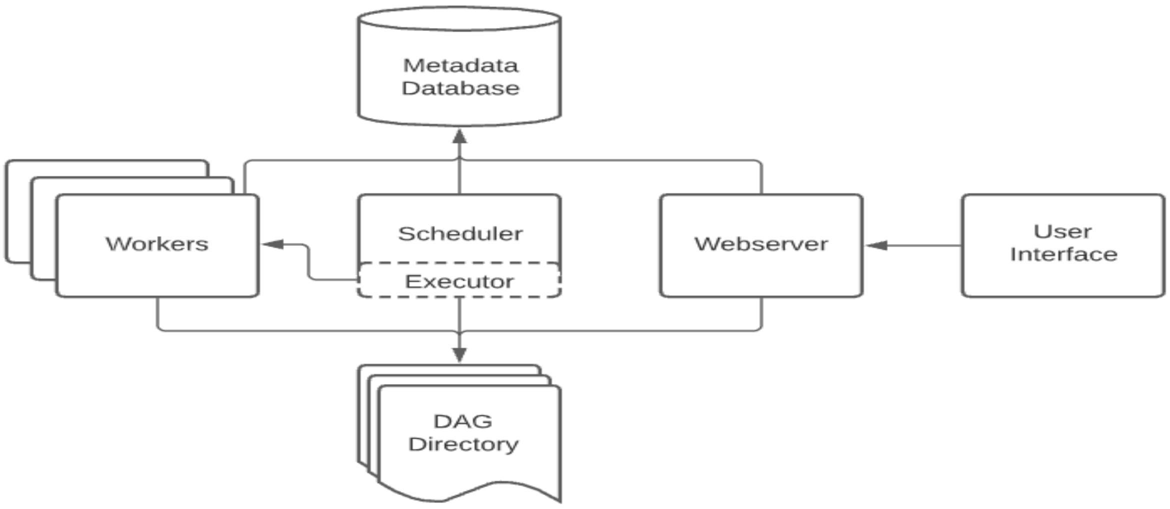
You learnt that by going to the following tabs you can access the mentioned information

- **Grid**: Status of all the executed and running tasks
- **Graph**: View DAG in the form of graph
- **Calendar**: Schedule for the DAG
- **Code**: Code where we defined the tasks and relations. Note that the code that we see is not editable

Recall that we did not add any logging. This is because we know that Airflow automatically logs our runs.

Airflow's Architecture

You learnt how the architecture of Airflow can be related to a factory/assembly line.



In the analogy, we related a scheduler to a manager who is assigned the task of ensuring that things are running smoothly. You learnt that a scheduler is responsible for managing all the tasks in Airflow.

For this, it maintains tabs on when to trigger which pipeline, pipeline configuration, etc. in a database. This is our metadata database. This is similar to the task sheet managed by the manager of a factory. Now, in order to manufacture a part, we need workers. In the case of Airflow, workers are analogous to computational resources.

Our DAG directory is a directory where we store all the files used to create a DAG. Recall that we created our heart_disease directory in a folder called dags in Airflow. This is Airflow's DAG directory. Now, a web server is essentially a way for users to interact with all the information about the pipelines' schedules.

Recall that while executing the commands, we first ran **airflow db init**, then **airflow scheduler** and finally **airflow webserver**. This means that we first initiated our database, as all the information regarding scheduling is stored in it. Then, we initialised our scheduler. Initiating the scheduler before the database would throw an error. This is because what we are doing is equivalent to asking a factory manager to do a set of tasks without telling them what those tasks are. Finally, after initialising the database and scheduler, we initialised the webserver.