* A DataFrame is a way of storing and manipulating tabular data in Python. DataFrames are often likened to tables with columns and rows that you could find in any [data warehouse](https://docs.getdbt.com/terms/data-warehouse), Google Sheet, or Excel workbook.
* A DataFrame entry in an analytics engineering glossary…what is happening? You’re reading this right. While SQL is the go-to programming language for most analytics engineering work, there are likely inevitable situations where you've found yourself writing some Python and using DataFrames.
* While DataFrames are also used in other languages for data processing, such as R and Scala, the focus of this glossary page will be on Python DataFrames, their use cases, and their relation to analytics engineering work.

DataFrame use cases[​](https://docs.getdbt.com/terms/dataframe#dataframe-use-cases)

You could probably write hundreds of pages on DataFrame use cases and examples, but at their core, DataFrames, *in the context of analytics engineering*, are often used to manipulate data outside of SQL capabilities, work with data during API extraction, and leverage data science and machine learning.

<https://docs.getdbt.com/terms/dataframe>

* You could probably write hundreds of pages on DataFrame use cases and examples, but at their core, DataFrames, *in the context of analytics engineering*, are often used to manipulate data outside of SQL capabilities, work with data during API extraction, and leverage data science and machine learning.

Conclusion[**​**](https://docs.getdbt.com/terms/dataframe#conclusion)

A DataFrame is a tabular data storage format in Python that is widely used across different roles in the data world. Since a DataFrame stores data in rows and columns, similar to how analytics engineers manipulate tables stored in data warehouses, data folks can transform, engineer, and enrich data in DataFrames using Python and Python packages. Analytics engineers may find themselves using DataFrames when they’re extracting data via APIs, enriching data with third-party packages, or experimenting with data science and machine learning models.

What is a Python model?[​](https://docs.getdbt.com/docs/build/python-models#what-is-a-python-model)

A dbt Python model is a function that reads in dbt sources or other models, applies a series of transformations, and returns a transformed dataset. [DataFrame](https://docs.getdbt.com/terms/dataframe) operations define the starting points, the end state, and each step along the way.

This is similar to the role of [CTEs](https://docs.getdbt.com/terms/cte) in dbt SQL models. We use CTEs to pull in upstream datasets, define (and name) a series of meaningful transformations, and end with a final select statement. You can run the compiled version of a dbt SQL model to see the data included in the resulting view or table. When you dbt run, dbt wraps that query in create view, create table, or more complex DDL to save its results in the database.

Instead of a final select statement, each Python model returns a final DataFrame. Each DataFrame operation is "lazily evaluated." In development, you can preview its data, using methods like .show() or .head(). When you run a Python model, the full result of the final DataFrame will be saved as a table in your data warehouse.

dbt Python models have access to almost all of the same configuration options as SQL models. You can test and document them, add tags and meta properties, and grant access to their results to other users. You can select them by their name, file path, configurations, whether they are upstream or downstream of another model, or if they have been modified compared to a previous project state.

Defining a Python model[​](https://docs.getdbt.com/docs/build/python-models#defining-a-python-model)

Each Python model lives in a .py file in your models/ folder. It defines a function named **model()**, which takes two parameters:

* **dbt**: A class compiled by dbt Core, unique to each model, enables you to run your Python code in the context of your dbt project and DAG.
* **session**: A class representing your data platform’s connection to the Python backend. The session is needed to read in tables as DataFrames, and to write DataFrames back to tables. In PySpark, by convention, the SparkSession is named spark, and available globally. For consistency across platforms, we always pass it into the model function as an explicit argument called session.

The model() function must return a single DataFrame. On Snowpark (Snowflake), this can be a Snowpark or pandas DataFrame. Via PySpark (Databricks + BigQuery), this can be a Spark, pandas, or pandas-on-Spark DataFrame. For more about choosing between pandas and native DataFrames, see [DataFrame API + syntax](https://docs.getdbt.com/docs/build/python-models#dataframe-api--syntax).

*snowpark can be used as an alternative to the pandas due to its limited features and other reason for using snowpark are below:*

*Snowpark is the new feature of snowflake warehouse in which an individual can to write code in their preferred language and run that code directly on Snowflake. It supports python, scala and java.*

*So whenever python wants to interact to snowflake in its own native language and syntax then it will have to use snowpark.*

When you dbt run --select python\_model, dbt will prepare and pass in both arguments (dbt and session). All you have to do is define the function. This is how every single Python model should look:

Example:

**Model name: python\_model\_1.py**

def model(dbt, session):

    my\_sql\_model\_df = dbt.ref("chicagocrimedata")

    final\_df = my\_sql\_model\_df

    return final\_df

* *This code created a table in the target schema in which the data from chicagocrimedata has been fetched and which has been equal to the final\_df, so it is saved in it and a new model star\_schema.python (model name) has been created in the snowflake which contains all the data of the model referred as dbt.ref(“chicagocrimedata)*

We can get data from one python model to another sql model with the help of ref function

**Referencing other models**[**​**](https://docs.getdbt.com/docs/build/python-models#referencing-other-models)

Python models participate fully in dbt's directed acyclic graph (DAG) of transformations. Use the dbt.ref() method within a Python model to read data from other models (SQL or Python). If you want to read directly from a raw source table, use dbt.source(). These methods return DataFrames pointing to the upstream source, model, seed, or snapshot.

**def model(dbt, session):**

**# DataFrame representing an upstream model**

**upstream\_model = dbt.ref("upstream\_model\_name")**

**# DataFrame representing an upstream source**

**upstream\_source = dbt.source("upstream\_source\_name", "table\_name")**

**...**

Here is another sql table which will be used to fetch data from python model

**File name: sql\_model.sql**

with abc as

 (select \* from {{ ref("python\_model\_1") }})

  select \* from abc

data is successfully fetched from python model to sql model.

Configuring Python models[​](https://docs.getdbt.com/docs/build/python-models#configuring-python-models)

Just like SQL models, there are three ways to configure Python models:

1. In dbt\_project.yml, where you can configure many models at once
2. In a dedicated .yml file, within the models/ directory
3. Within the model's .py file, using the dbt.config() method

Calling the dbt.config() method will set configurations for your model within your .py file, similar to the {{ config() }} macro in .sql model files:

def model(dbt, session):

    dbt.config(materialized = "table")

    my\_sql\_model\_df = dbt.ref("chicagocrimedata")

    final\_df = my\_sql\_model\_df

    return final\_df

Out of the box, the dbt class supports:

* Returning DataFrames referencing the locations of other resources: dbt.ref() + dbt.source()
* Accessing the database location of the current model:  dbt.this() (also: dbt.this.database, .schema, .identifier)
* Determining if the current model's run is incremental: dbt.is\_incremental

It is possible to extend this context by "getting" them via dbt.config.get() after they are configured in the [model's config](https://docs.getdbt.com/reference/model-configs). This includes inputs such as var, env\_var, and target. If you want to use those values to power conditional logic in your model, we require setting them through a dedicated .yml file config:

Configuring

config file is being configured - dbt\_project.yml

version: 2  
  
models:  
 - name: my\_python\_model  
 config:  
 materialized: table  
 target\_name: "{{ target.name }}"  
 specific\_var: "{{ var('SPECIFIC\_VAR') }}"  
 specific\_env\_var: "{{ env\_var('SPECIFIC\_ENV\_VAR') }}"

use the dbt.config.get() function to access values of configurations that have been set:

python\_model\_1.py

now we will have to configure the python model and get all the configurations set in the config.yml file. and after getting all the config files we can continue with our code.

def model(dbt, session):

target\_name = dbt.config.get("target\_name")

specific\_var = dbt.config.get("specific\_var")

specific\_env\_var = dbt.config.get("specific\_env\_var")

orders\_df = dbt.ref("fct\_orders")

# limit data in dev

if target\_name == "dev":

orders\_df = orders\_df.limit(500)

Materializations[​](https://docs.getdbt.com/docs/build/python-models#materializations)

Python models support these materializations:

* table (default)
* incremental

Python models can't be materialized as view or ephemeral. Python isn't supported for non-model resource types (like tests, macros and snapshots).

import **snowflake.snowpark.functions** as F  
  
**def model(dbt, session):  
 dbt.config(materialized = "incremental")  
 df = dbt.ref("upstream\_table")**  
  
 **if dbt.is\_incremental:**  
  
 *# only new rows compared to max in current table*  
 **max\_from\_this = f"select max(updated\_at) from {dbt.this}"  
 df = df.filter(df.updated\_at >= session.sql(max\_from\_this).collect()[0][0])**  
 *# or only rows from the past 3 days*  
 **df = df.filter(df.updated\_at >= F.dateadd("day", F.lit(-3), F.current\_timestamp()))**  
 ...  
  
 **return df**

* first we have to import the snowpark from snowflake
* then it is necessary that model is defined with parameters dbt and session inside it.
* Dbt.ref is used to get table or model from any other model already defined may be sql or python.
* Df has several functions associated with it and it can be used along with them