**Leverage dbt Cloud to generate analytics and ML-ready pipelines with SQL and Python with Snowflake**

Updated

Snowflake

Intermediate

**Introduction**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#introduction)

The focus of this workshop will be to demonstrate how we can use both *SQL and python together* in the same workflow to run *both analytics and machine learning models* on dbt Cloud.

All code in today’s workshop can be found on [GitHub](https://github.com/dbt-labs/python-snowpark-formula1/tree/python-formula1).

**What you'll use during the lab**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#what-youll-use-during-the-lab)

* A [Snowflake account](https://trial.snowflake.com/) with ACCOUNTADMIN access
* A [dbt Cloud account](https://www.getdbt.com/signup/" \t "_blank)

**What you'll learn**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#what-youll-learn)

* How to build scalable data transformation pipelines using dbt, and Snowflake using SQL and Python
* How to leverage copying data into Snowflake from a public S3 bucket

**What you need to know**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#what-you-need-to-know)

* Basic to intermediate SQL and python.
* Basic understanding of dbt fundamentals. We recommend the [dbt Fundamentals course](https://courses.getdbt.com/collections" \t "_blank) if you're interested.
* High level machine learning process (encoding, training, testing)
* Simple ML algorithms — we will use logistic regression to keep the focus on the *workflow*, not algorithms!

**What you'll build**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#what-youll-build)

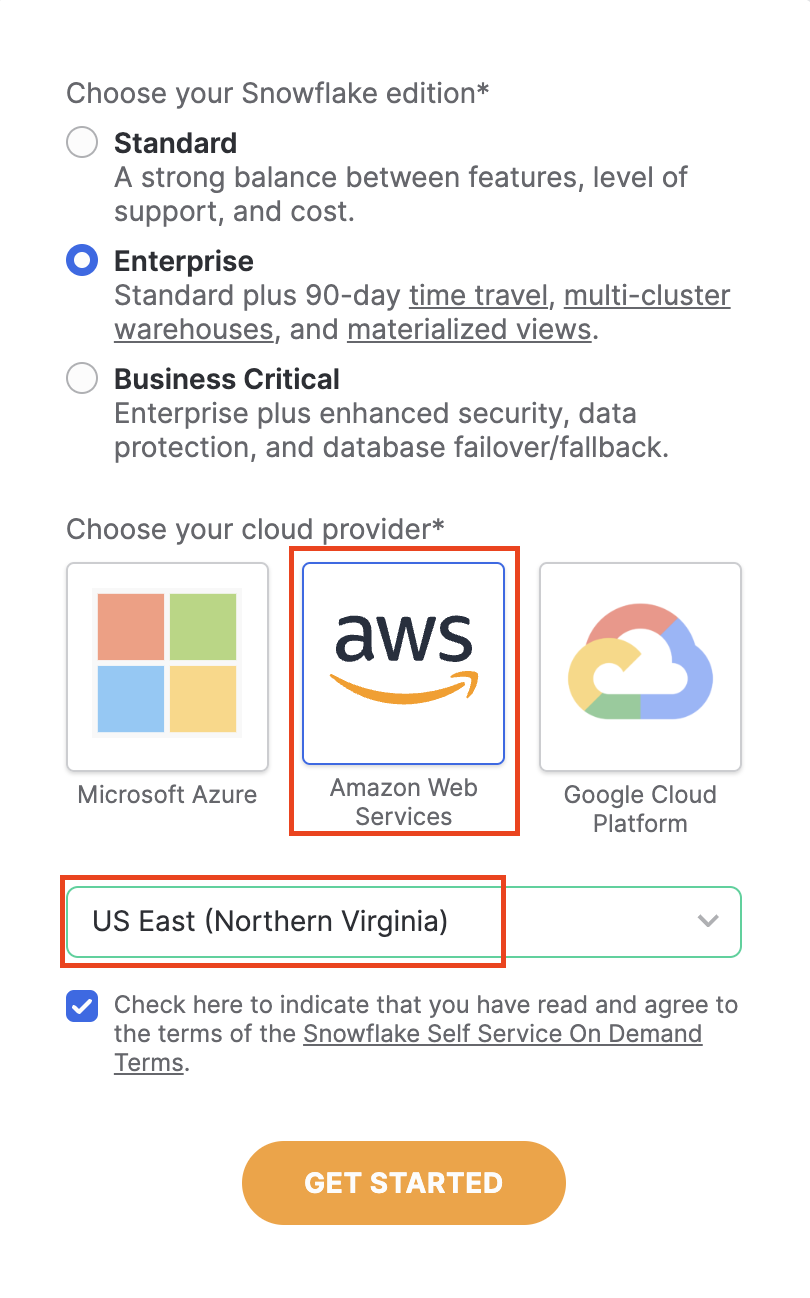
* A set of data analytics and prediction pipelines using Formula 1 data leveraging dbt and Snowflake, making use of best practices like data quality tests and code promotion between environments
* We will create insights for:
  1. Finding the lap time average and rolling average through the years (is it generally trending up or down)?
  2. Which constructor has the fastest pit stops in 2021?
  3. Predicting the position of each driver given using a decade of data (2010 - 2020)

As inputs, we are going to leverage Formula 1 datasets hosted on a dbt Labs public S3 bucket. We will create a Snowflake Stage for our CSV files then use Snowflake’s COPY INTO function to copy the data in from our CSV files into tables. The Formula 1 is available on [Kaggle](https://www.kaggle.com/datasets/rohanrao/formula-1-world-championship-1950-2020). The data is originally compiled from the [Ergast Developer API](http://ergast.com/mrd/" \t "_blank).

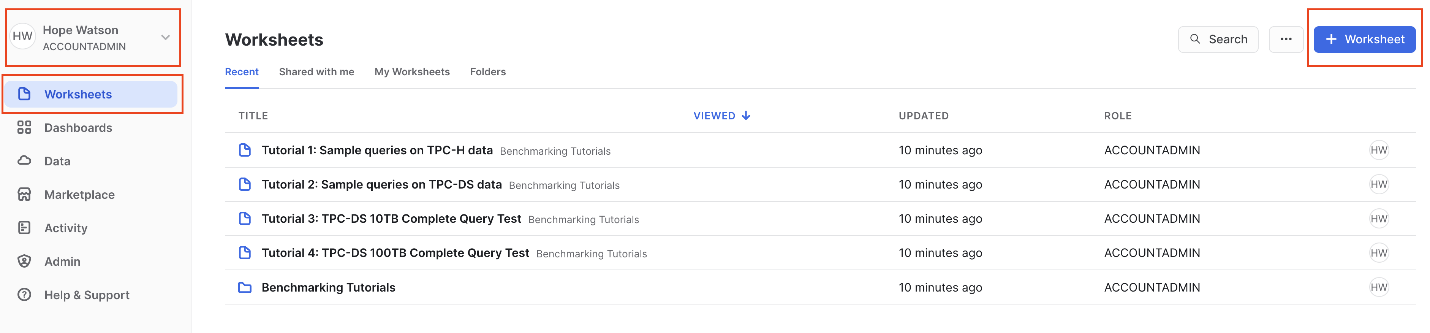
Overall we are going to set up the environments, build scalable pipelines in dbt, establish data tests, and promote code to production.

**Configure Snowflake**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#configure-snowflake)

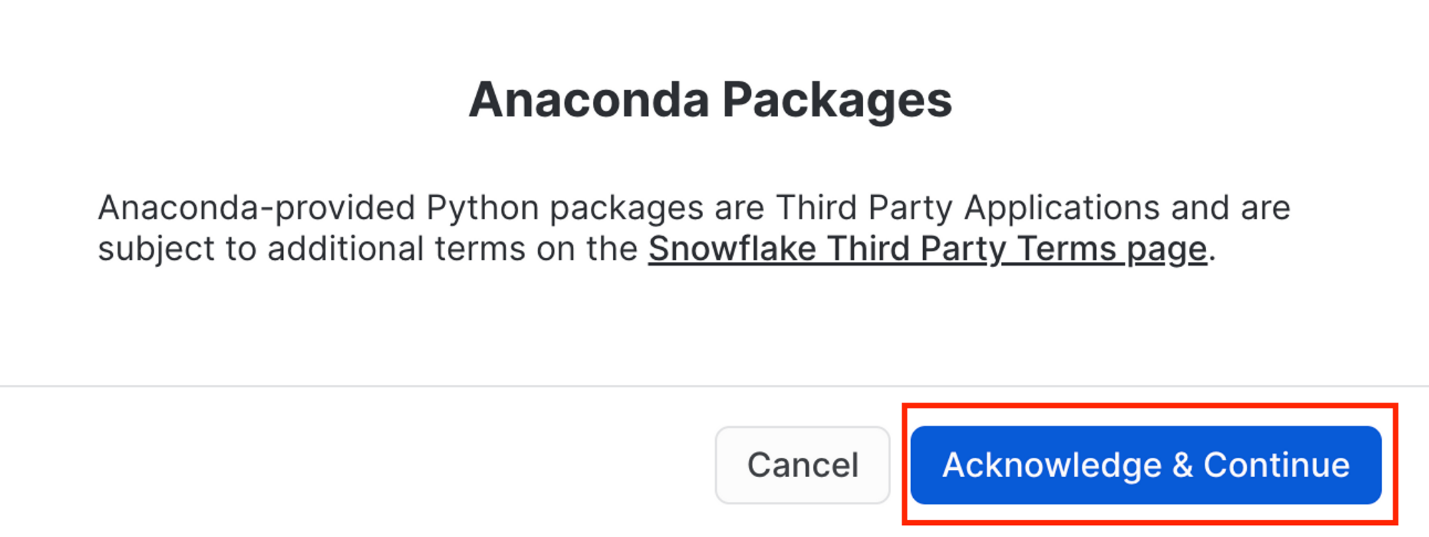
1. Log in to your trial Snowflake account. You can [sign up for a Snowflake Trial Account using this form](https://signup.snowflake.com/) if you don’t have one.
2. Ensure that your account is set up using **AWS** in the **US East (N. Virginia)**. We will be copying the data from a public AWS S3 bucket hosted by dbt Labs in the us-east-1 region. By ensuring our Snowflake environment setup matches our bucket region, we avoid any multi-region data copy and retrieval latency issues.

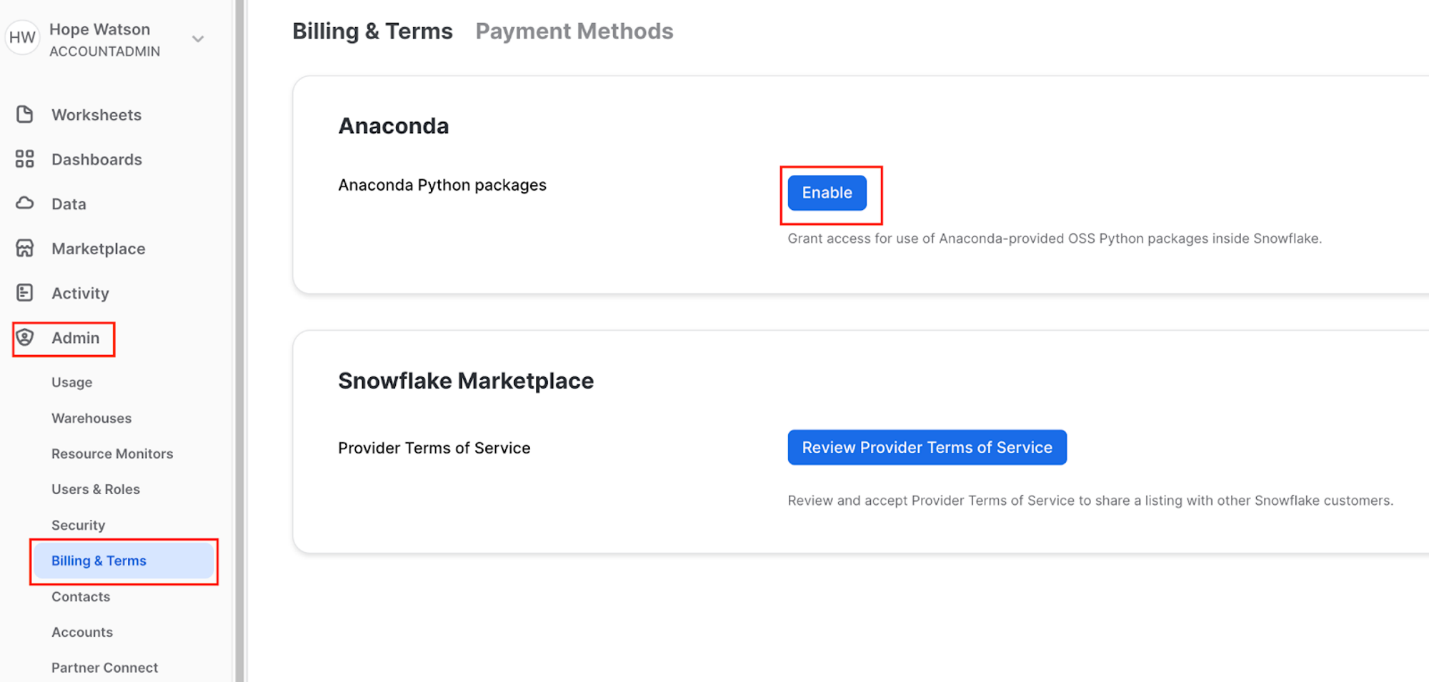
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Snowflake trial

1. After creating your account and verifying it from your sign-up email, Snowflake will direct you back to the UI called Snowsight.
2. When Snowsight first opens, your window should look like the following, with you logged in as the ACCOUNTADMIN with demo worksheets open:

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Snowflake trial demo worksheets

1. Navigate to **Admin > Billing & Terms**. Click **Enable > Acknowledge & Continue** to enable Anaconda Python Packages to run in Snowflake.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Anaconda terms

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Enable Anaconda

1. Finally, create a new Worksheet by selecting **+ Worksheet** in the upper right corner.

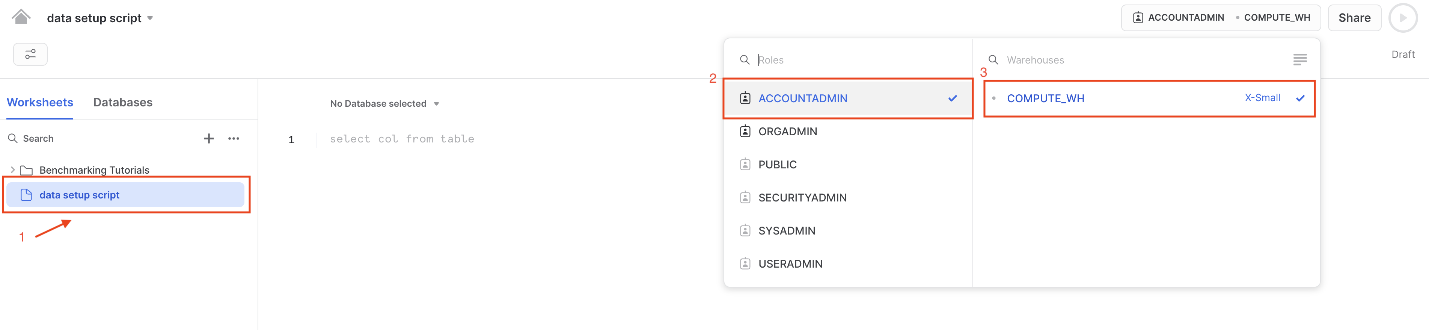
**Connect to data source**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#connect-to-data-source)

We need to obtain our data source by copying our Formula 1 data into Snowflake tables from a public S3 bucket that dbt Labs hosts.

1. When a new Snowflake account is created, there should be a preconfigured warehouse in your account named COMPUTE\_WH.
2. If for any reason your account doesn’t have this warehouse, we can create a warehouse using the following script:

create or replace warehouse COMPUTE\_WH with warehouse\_size=XSMALL

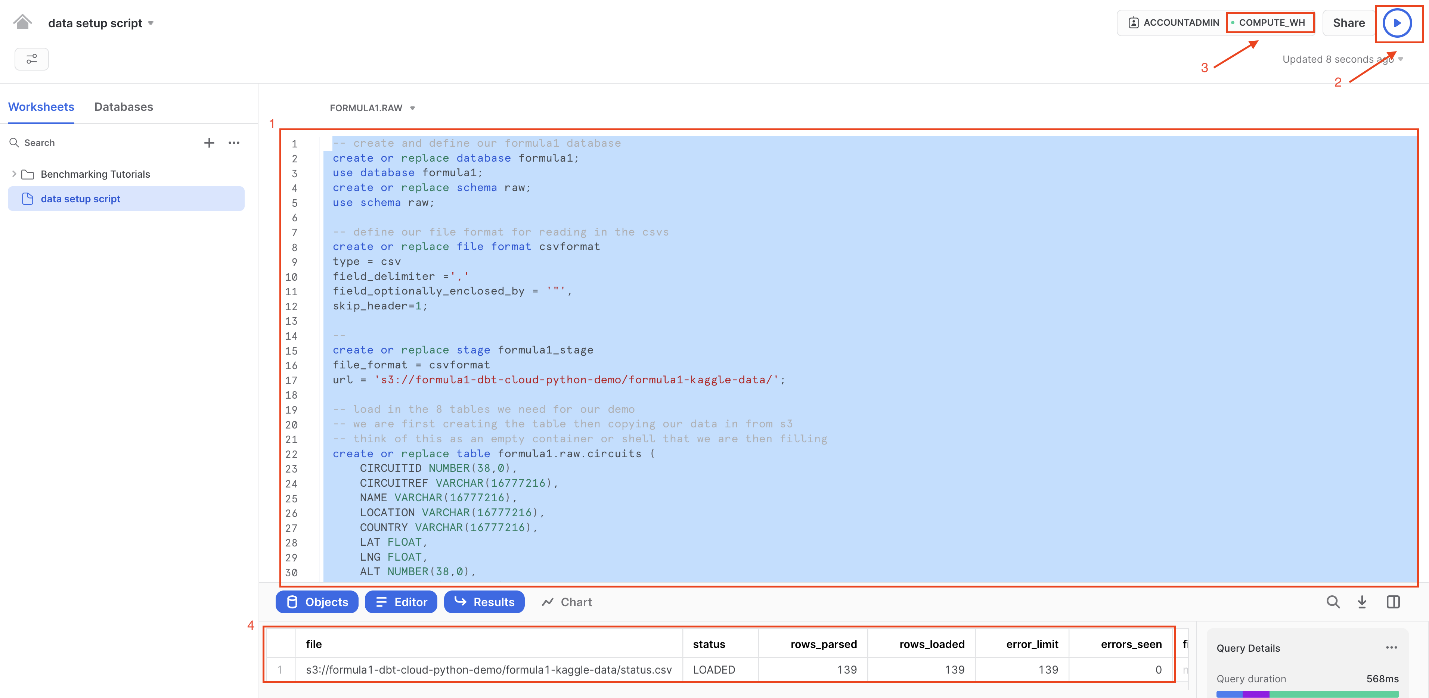
1. Rename the worksheet to data setup script since we will be placing code in this worksheet to ingest the Formula 1 data. Make sure you are still logged in as the **ACCOUNTADMIN** and select the **COMPUTE\_WH** warehouse.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Rename worksheet and select warehouse

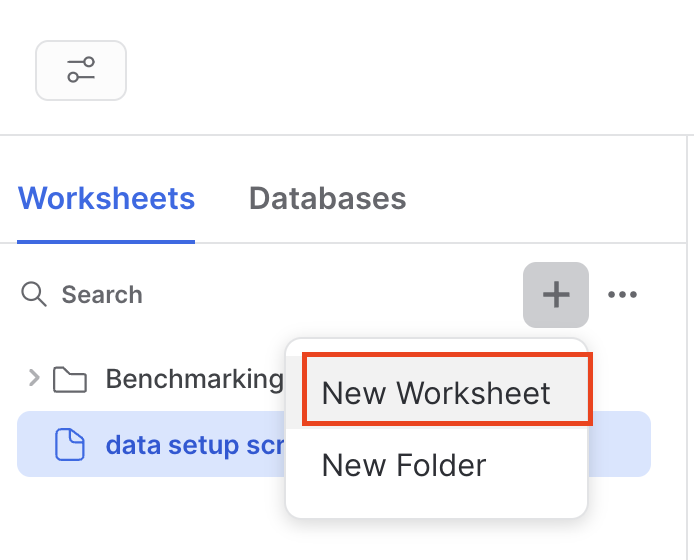
1. Copy the following code into the main body of the Snowflake worksheet. You can also find this setup script under the setup folder in the [Git repository](https://github.com/dbt-labs/python-snowpark-formula1/blob/main/setup/setup_script_s3_to_snowflake.sql). The script is long since it's bring in all of the data we'll need today!

*-- create and define our formula1 database*  
create or replace database formula1;  
use database formula1;   
create or replace schema raw;   
use schema raw;   
  
*-- define our file format for reading in the csvs*   
create or replace file format csvformat  
type = csv  
field\_delimiter =','  
field\_optionally\_enclosed\_by = '"',   
skip\_header=1;   
  
*--*  
create or replace stage formula1\_stage  
file\_format = csvformat   
url = 's3://formula1-dbt-cloud-python-demo/formula1-kaggle-data/';  
  
*-- load in the 8 tables we need for our demo*   
*-- we are first creating the table then copying our data in from s3*  
*-- think of this as an empty container or shell that we are then filling*  
create or replace table formula1.raw.circuits (  
 CIRCUITID NUMBER(38,0),  
 CIRCUITREF VARCHAR(16777216),  
 NAME VARCHAR(16777216),  
 LOCATION VARCHAR(16777216),  
 COUNTRY VARCHAR(16777216),  
 LAT FLOAT,  
 LNG FLOAT,  
 ALT NUMBER(38,0),  
 URL VARCHAR(16777216)  
);  
*-- copy our data from public s3 bucket into our tables*   
copy into circuits   
from @formula1\_stage/circuits.csv  
on\_error='continue';  
  
create or replace table formula1.raw.constructors (  
 CONSTRUCTORID NUMBER(38,0),  
 CONSTRUCTORREF VARCHAR(16777216),  
 NAME VARCHAR(16777216),  
 NATIONALITY VARCHAR(16777216),  
 URL VARCHAR(16777216)  
);  
copy into constructors   
from @formula1\_stage/constructors.csv  
on\_error='continue';  
  
create or replace table formula1.raw.drivers (  
 DRIVERID NUMBER(38,0),  
 DRIVERREF VARCHAR(16777216),  
 NUMBER VARCHAR(16777216),  
 CODE VARCHAR(16777216),  
 FORENAME VARCHAR(16777216),  
 SURNAME VARCHAR(16777216),  
 DOB DATE,  
 NATIONALITY VARCHAR(16777216),  
 URL VARCHAR(16777216)  
);  
copy into drivers   
from @formula1\_stage/drivers.csv  
on\_error='continue';  
  
create or replace table formula1.raw.lap\_times (  
 RACEID NUMBER(38,0),  
 DRIVERID NUMBER(38,0),  
 LAP NUMBER(38,0),  
 POSITION FLOAT,  
 TIME VARCHAR(16777216),  
 MILLISECONDS NUMBER(38,0)  
);  
copy into lap\_times   
from @formula1\_stage/lap\_times.csv  
on\_error='continue';  
  
create or replace table formula1.raw.pit\_stops (  
 RACEID NUMBER(38,0),  
 DRIVERID NUMBER(38,0),  
 STOP NUMBER(38,0),  
 LAP NUMBER(38,0),  
 TIME VARCHAR(16777216),  
 DURATION VARCHAR(16777216),  
 MILLISECONDS NUMBER(38,0)  
);  
copy into pit\_stops   
from @formula1\_stage/pit\_stops.csv  
on\_error='continue';  
  
create or replace table formula1.raw.races (  
 RACEID NUMBER(38,0),  
 YEAR NUMBER(38,0),  
 ROUND NUMBER(38,0),  
 CIRCUITID NUMBER(38,0),  
 NAME VARCHAR(16777216),  
 DATE DATE,  
 TIME VARCHAR(16777216),  
 URL VARCHAR(16777216),  
 FP1\_DATE VARCHAR(16777216),  
 FP1\_TIME VARCHAR(16777216),  
 FP2\_DATE VARCHAR(16777216),  
 FP2\_TIME VARCHAR(16777216),  
 FP3\_DATE VARCHAR(16777216),  
 FP3\_TIME VARCHAR(16777216),  
 QUALI\_DATE VARCHAR(16777216),  
 QUALI\_TIME VARCHAR(16777216),  
 SPRINT\_DATE VARCHAR(16777216),  
 SPRINT\_TIME VARCHAR(16777216)  
);  
copy into races   
from @formula1\_stage/races.csv  
on\_error='continue';  
  
create or replace table formula1.raw.results (  
 RESULTID NUMBER(38,0),  
 RACEID NUMBER(38,0),  
 DRIVERID NUMBER(38,0),  
 CONSTRUCTORID NUMBER(38,0),  
 NUMBER NUMBER(38,0),  
 GRID NUMBER(38,0),  
 POSITION FLOAT,  
 POSITIONTEXT VARCHAR(16777216),  
 POSITIONORDER NUMBER(38,0),  
 POINTS NUMBER(38,0),  
 LAPS NUMBER(38,0),  
 TIME VARCHAR(16777216),  
 MILLISECONDS NUMBER(38,0),  
 FASTESTLAP NUMBER(38,0),  
 RANK NUMBER(38,0),  
 FASTESTLAPTIME VARCHAR(16777216),  
 FASTESTLAPSPEED FLOAT,  
 STATUSID NUMBER(38,0)  
);  
copy into results   
from @formula1\_stage/results.csv  
on\_error='continue';  
  
create or replace table formula1.raw.status (  
 STATUSID NUMBER(38,0),  
 STATUS VARCHAR(16777216)  
);  
copy into status   
from @formula1\_stage/status.csv  
on\_error='continue';

1. Ensure all the commands are selected before running the query — an easy way to do this is to use Ctrl-a to highlight all of the code in the worksheet. Select **run** (blue triangle icon). Notice how the dot next to your **COMPUTE\_WH** turns from gray to green as you run the query. The **status** table is the final table of all 8 tables loaded in.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Load data from S3 bucket

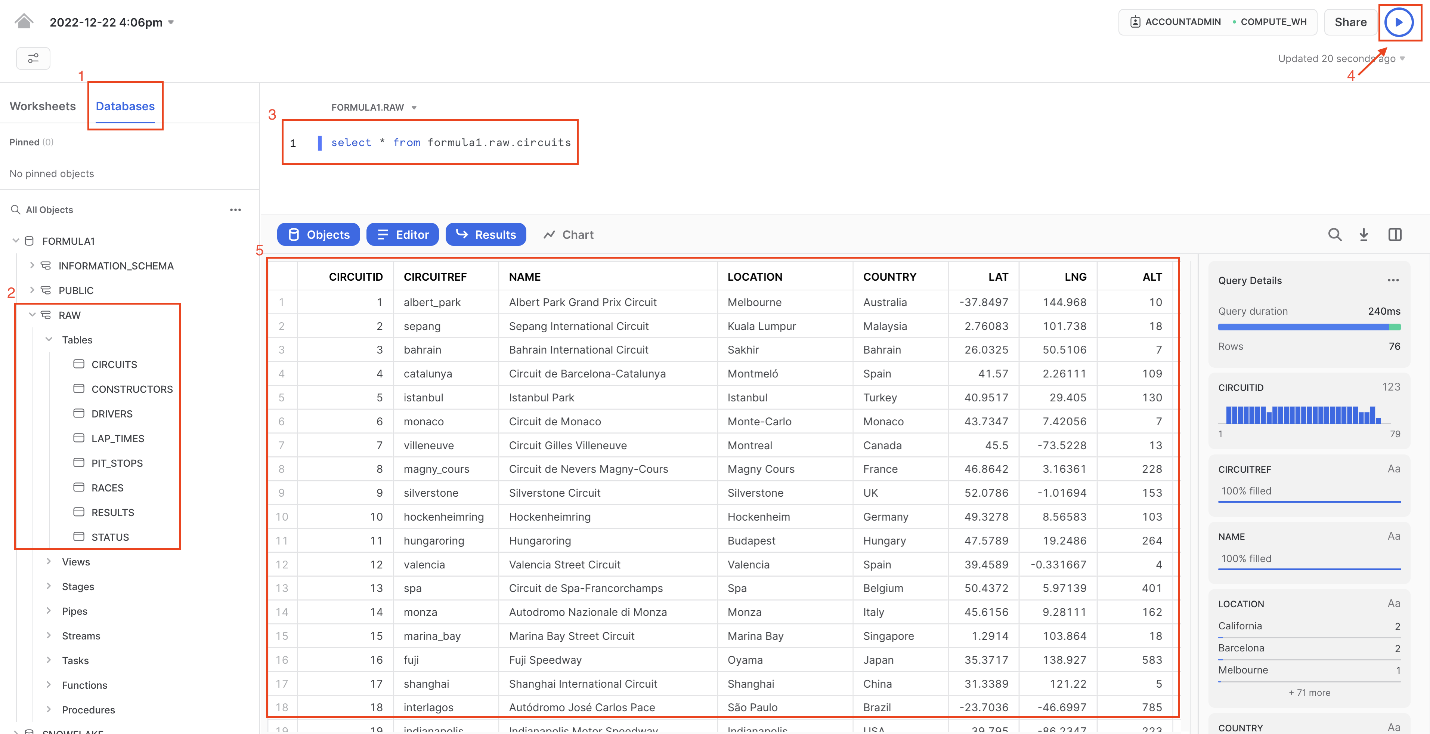
1. Let’s unpack that pretty long query we ran into component parts. We ran this query to load in our 8 Formula 1 tables from a public S3 bucket. To do this, we:
   * Created a new database called formula1 and a schema called raw to place our raw (untransformed) data into.
   * Defined our file format for our CSV files. Importantly, here we use a parameter called field\_optionally\_enclosed\_by = since the string columns in our Formula 1 csv files use quotes. Quotes are used around string values to avoid parsing issues where commas , and new lines /n in data values could cause data loading errors.
   * Created a stage to locate our data we are going to load in. Snowflake Stages are locations where data files are stored. Stages are used to both load and unload data to and from Snowflake locations. Here we are using an external stage, by referencing an S3 bucket.
   * Created our tables for our data to be copied into. These are empty tables with the column name and data type. Think of this as creating an empty container that the data will then fill into.
   * Used the copy into statement for each of our tables. We reference our staged location we created and upon loading errors continue to load in the rest of the data. You should not have data loading errors but if you do, those rows will be skipped and Snowflake will tell you which rows caused errors
2. Now let's take a look at some of our cool Formula 1 data we just loaded up!
   * Create a new worksheet by selecting the **+** then **New Worksheet**.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Create new worksheet to query data

* + Navigate to **Database > Formula1 > RAW > Tables**.
  + Query the data using the following code. There are only 76 rows in the circuits table, so we don’t need to worry about limiting the amount of data we query.

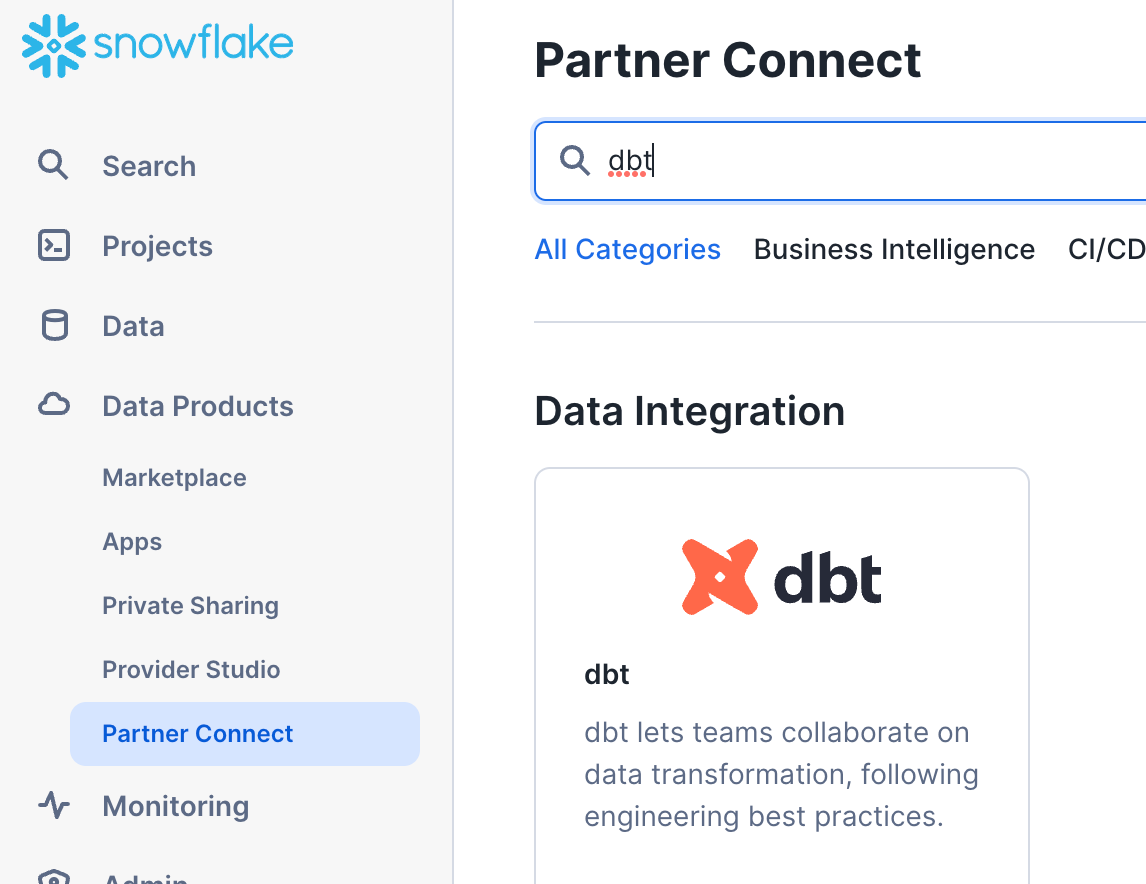
select \* from formula1.raw.circuits

* + Run the query. From here on out, we’ll use the keyboard shortcuts Command-Enter or Control-Enter to run queries and won’t explicitly call out this step.
  + Review the query results, you should see information about Formula 1 circuits, starting with Albert Park in Australia!
  + Finally, ensure you have all 8 tables starting with CIRCUITS and ending with STATUS. Now we are ready to connect into dbt Cloud!

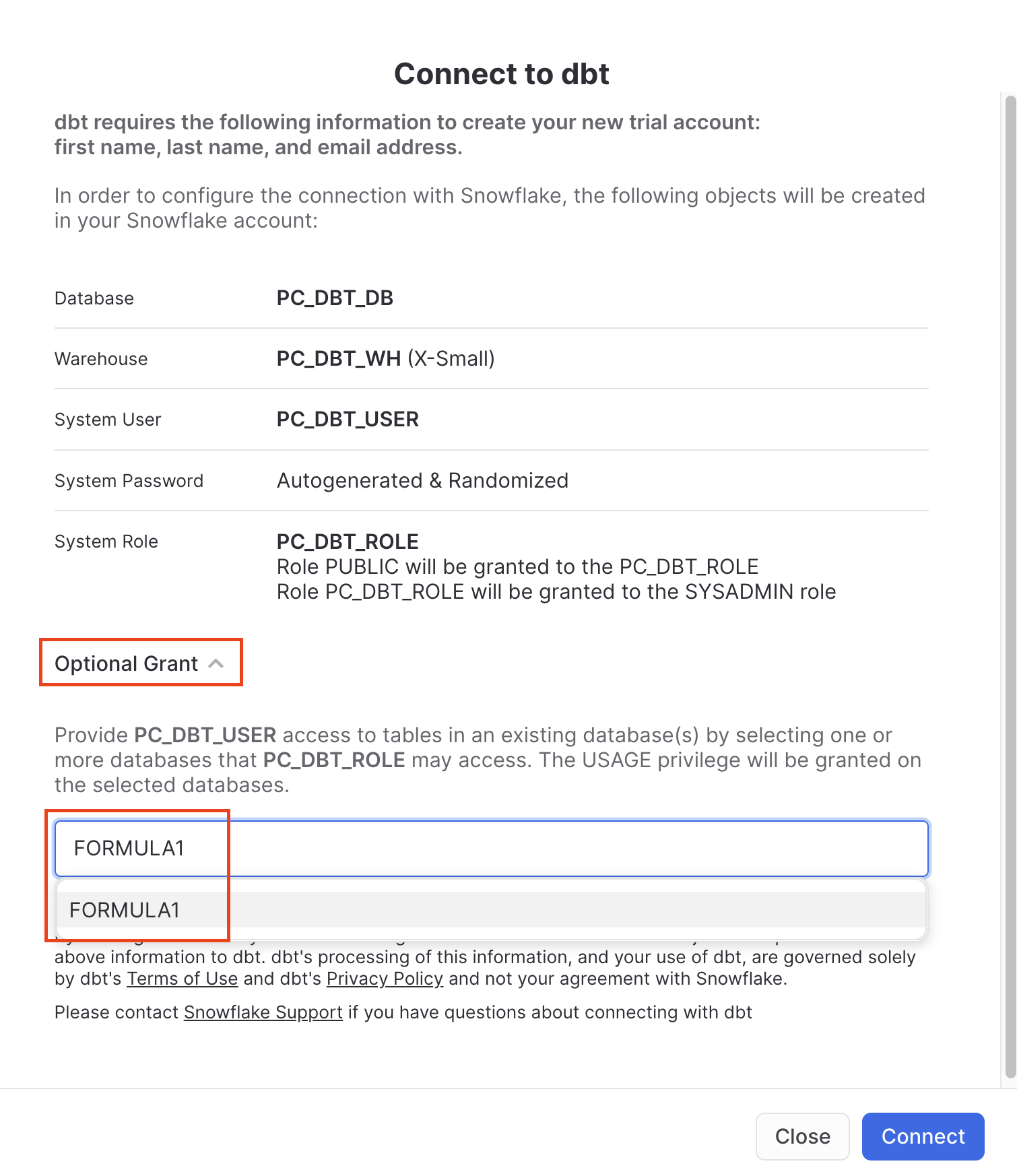
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Query circuits data

**Configure dbt Cloud**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#configure-dbt-cloud)

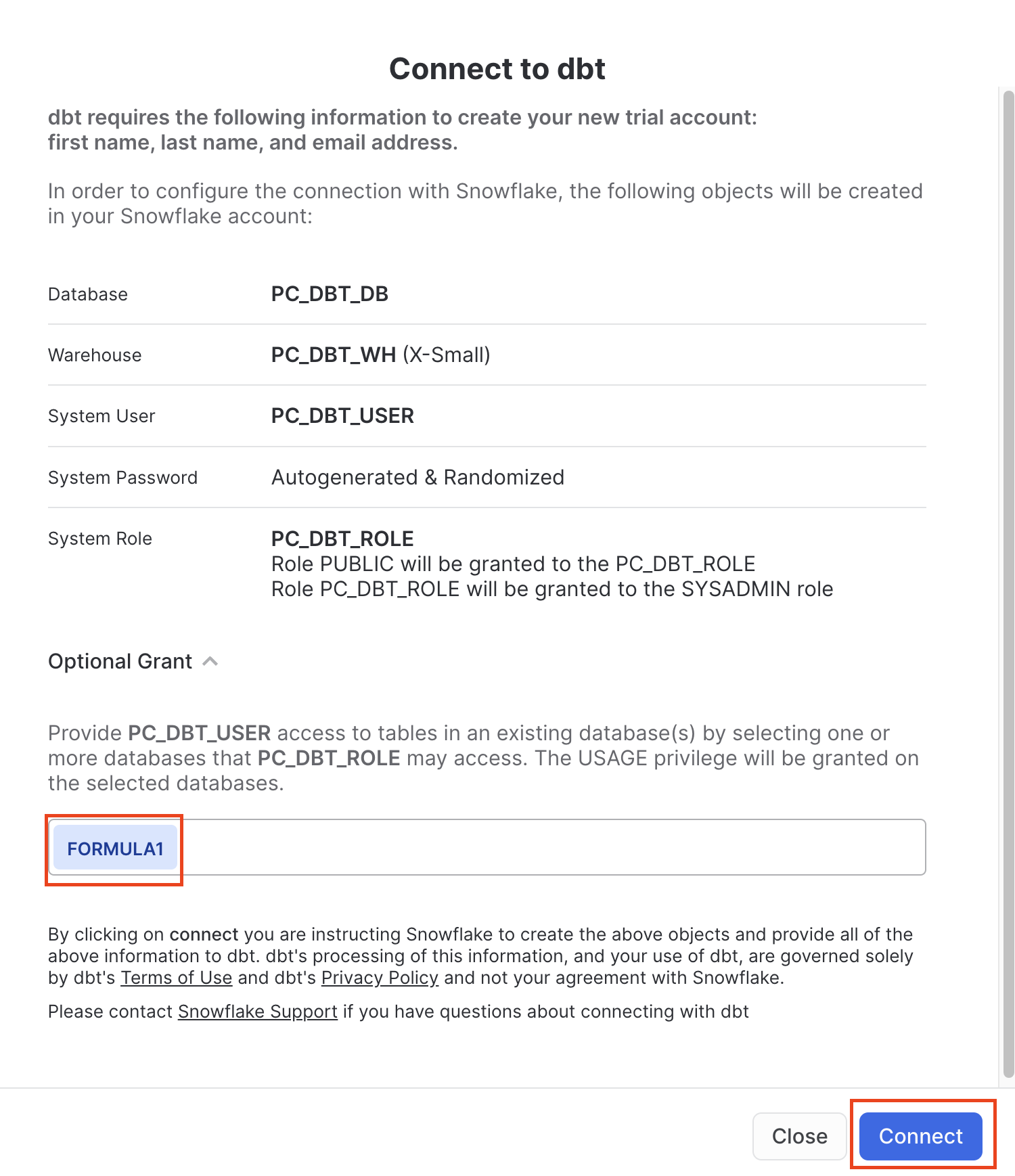
1. We are going to be using [Snowflake Partner Connect](https://docs.snowflake.com/en/user-guide/ecosystem-partner-connect.html) to set up a dbt Cloud account. Using this method will allow you to spin up a fully fledged dbt account with your [Snowflake connection](https://docs.getdbt.com/docs/cloud/connect-data-platform/connect-snowflake), [managed repository](https://docs.getdbt.com/docs/collaborate/git/managed-repository), environments, and credentials already established.
2. Navigate out of your worksheet back by selecting **home**.
3. In Snowsight, confirm that you are using the **ACCOUNTADMIN** role.
4. Navigate to the **Data Products** **> Partner Connect**. Find **dbt** either by using the search bar or navigating the **Data Integration**. Select the **dbt** tile.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Open Partner Connect

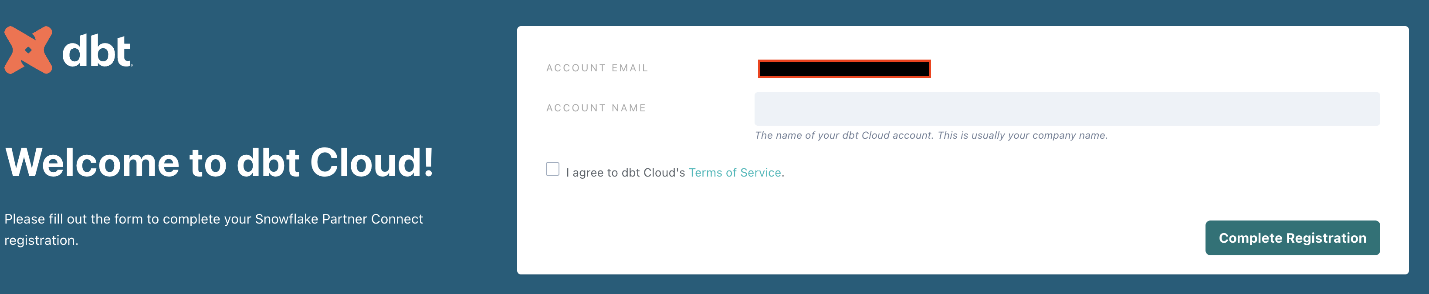
1. You should now see a new window that says **Connect to dbt**. Select **Optional Grant** and add the FORMULA1 database. This will grant access for your new dbt user role to the FORMULA1 database.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Partner Connect Optional Grant

1. Ensure the FORMULA1 is present in your optional grant before clicking **Connect**.  This will create a dedicated dbt user, database, warehouse, and role for your dbt Cloud trial.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Connect to dbt

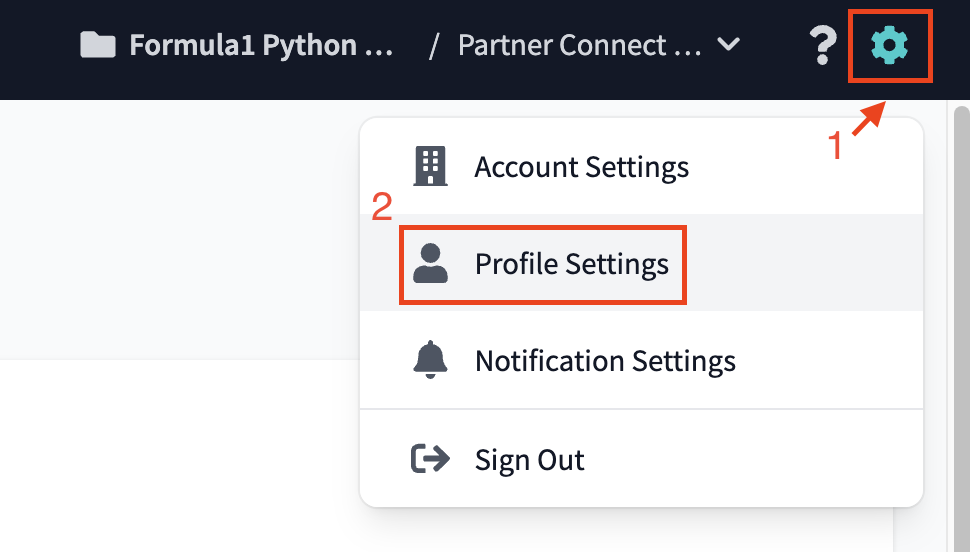
1. When you see the **Your partner account has been created** window, click **Activate**.
2. You should be redirected to a dbt Cloud registration page. Fill out the form. Make sure to save the password somewhere for login in the future.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)dbt Cloud sign up

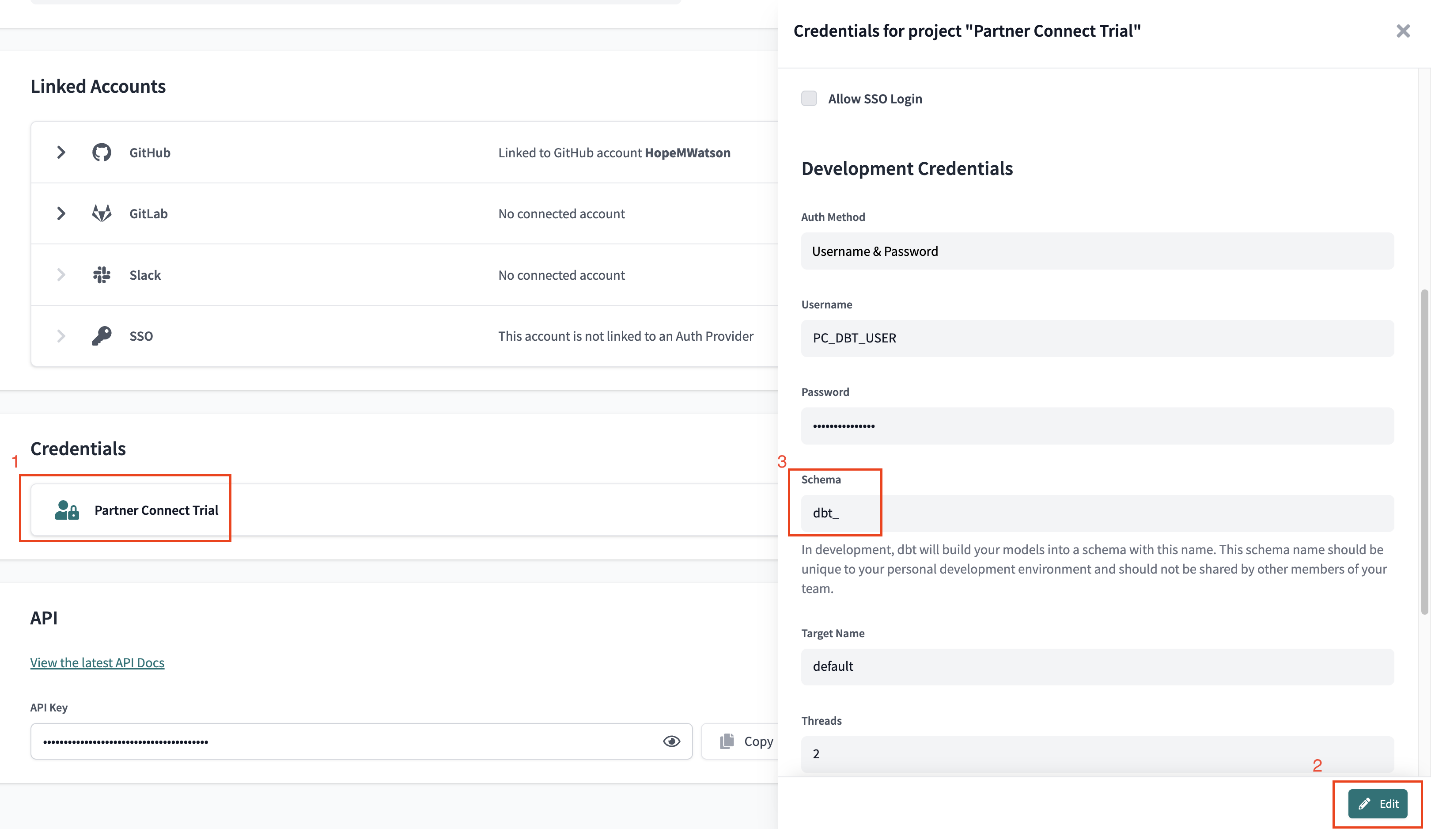
1. Select **Complete Registration**. You should now be redirected to your dbt Cloud account, complete with a connection to your Snowflake account, a deployment and a development environment, and a sample job.
2. To help you version control your dbt project, we have connected it to a [managed repository](https://docs.getdbt.com/docs/collaborate/git/managed-repository), which means that dbt Labs will be hosting your repository for you. This will give you access to a Git workflow without you having to create and host the repository yourself. You will not need to know Git for this workshop; dbt Cloud will help guide you through the workflow. In the future, when you’re developing your own project, [feel free to use your own repository](https://docs.getdbt.com/docs/cloud/git/connect-github). This will allow you to learn more about features like [Slim CI](https://docs.getdbt.com/docs/deploy/continuous-integration) builds after this workshop.

**Change development schema name navigate the IDE**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#change-development-schema-name-navigate-the-ide)

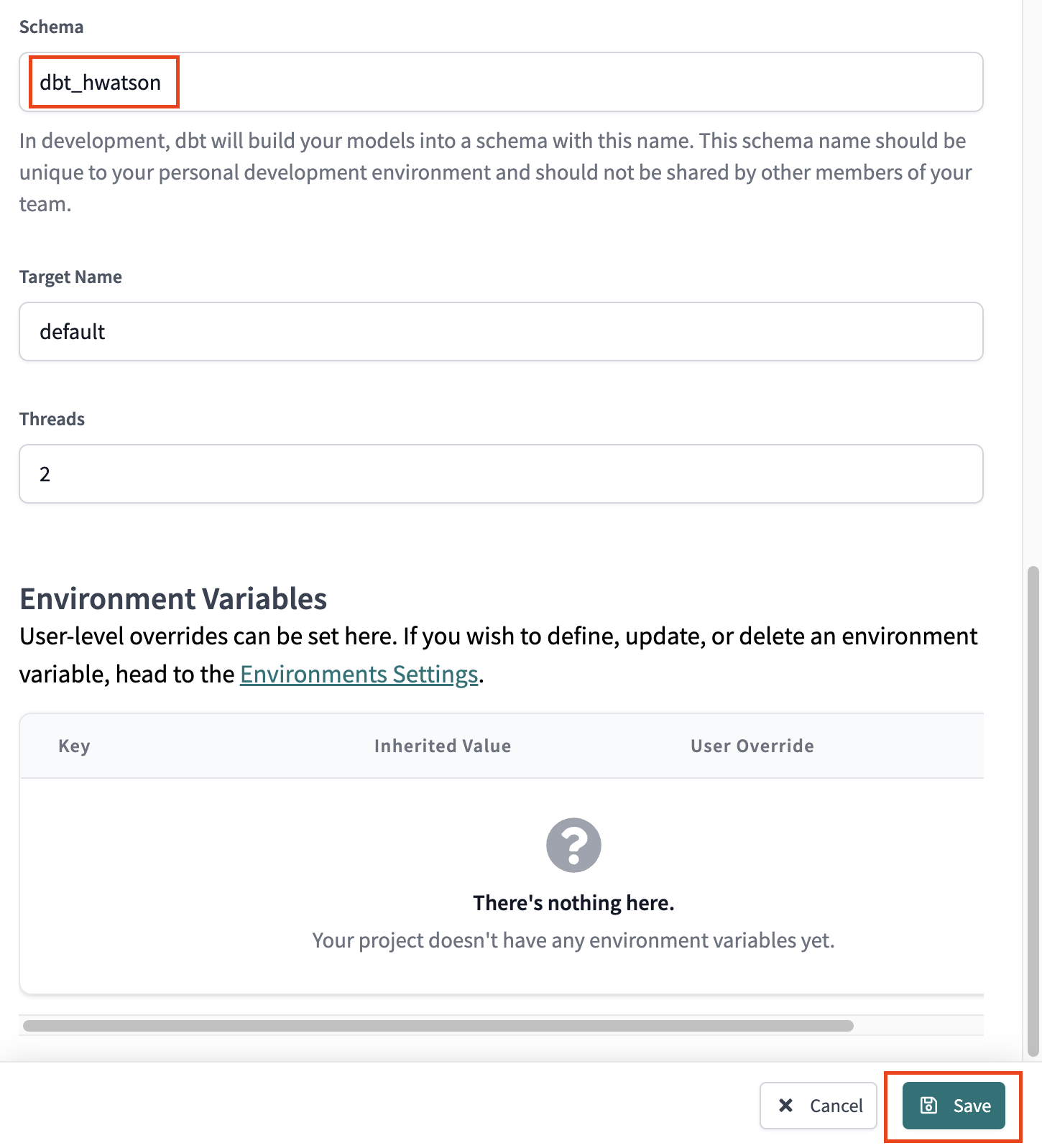
1. First we are going to change the name of our default schema to where our dbt models will build. By default, the name is dbt\_. We will change this to dbt\_<YOUR\_NAME> to create your own personal development schema. To do this, select **Profile Settings** from the gear icon in the upper right.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Settings menu

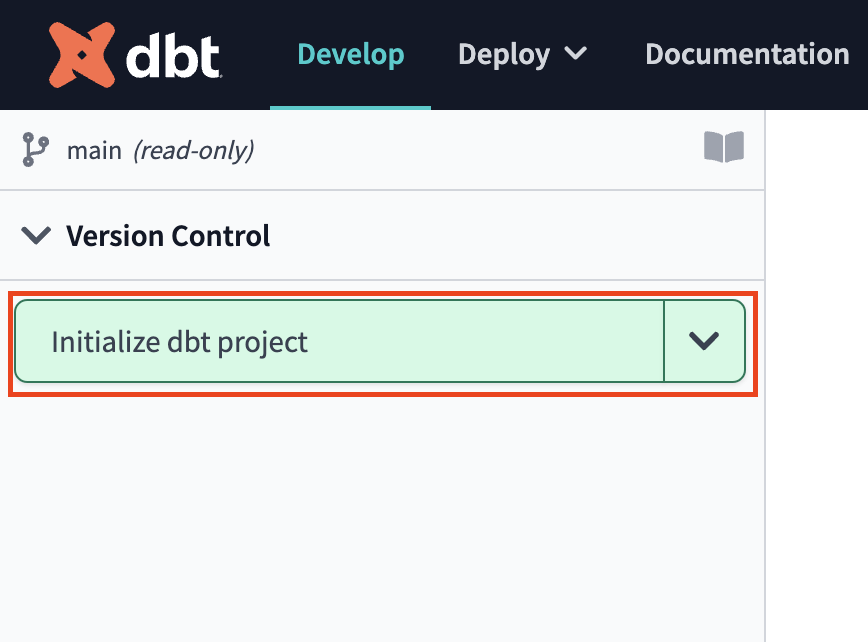
1. Navigate to the **Credentials** menu and select **Partner Connect Trial**, which will expand the credentials menu.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Credentials edit schema name

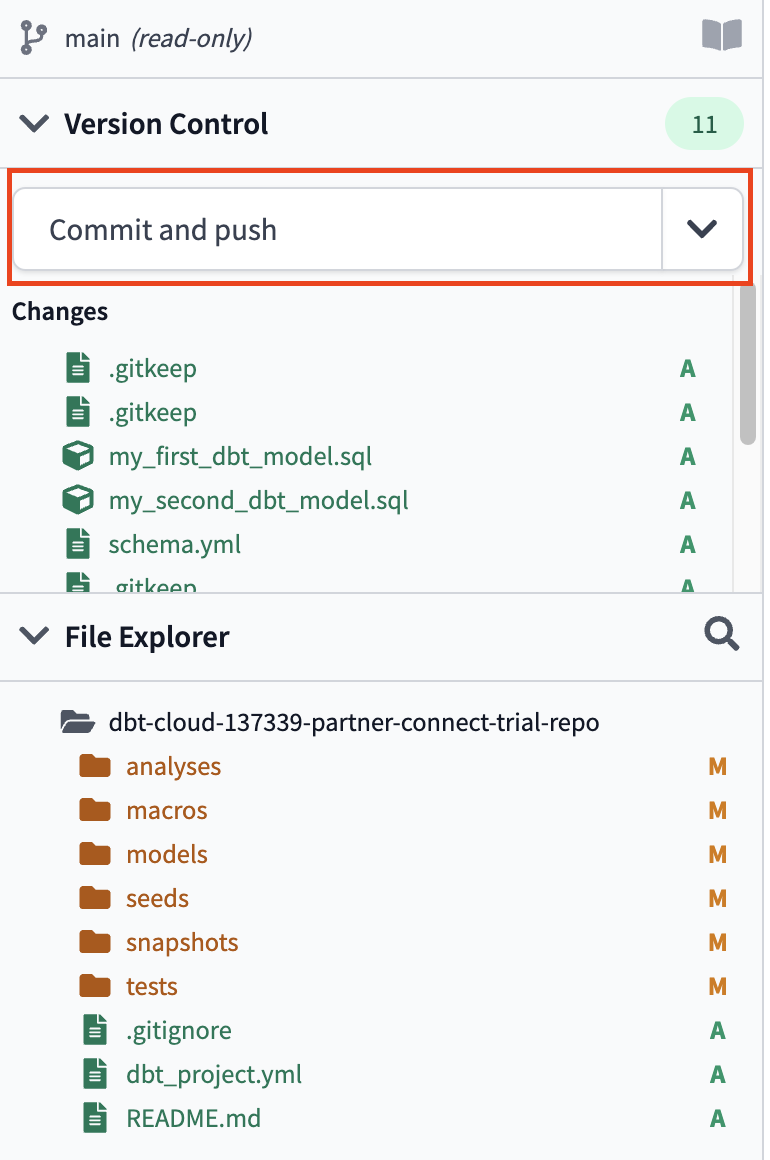
1. Click **Edit** and change the name of your schema from dbt\_ to dbt\_YOUR\_NAME replacing YOUR\_NAME with your initials and name (hwatson is used in the lab screenshots). Be sure to click **Save** for your changes!

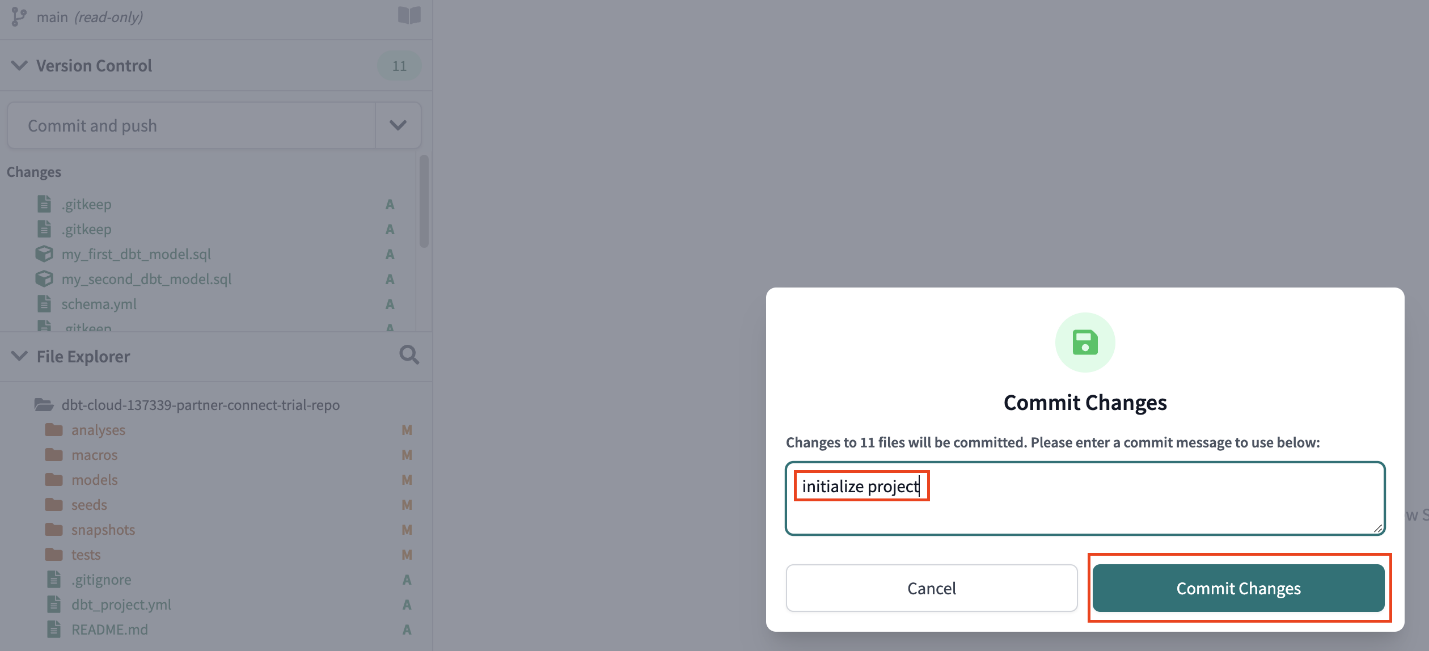
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Save new schema name

1. We now have our own personal development schema, amazing! When we run our first dbt models they will build into this schema.
2. Let’s open up dbt Cloud’s Integrated Development Environment (IDE) and familiarize ourselves. Choose **Develop** at the top of the UI.
3. When the IDE is done loading, click **Initialize dbt project**. The initialization process creates a collection of files and folders necessary to run your dbt project.

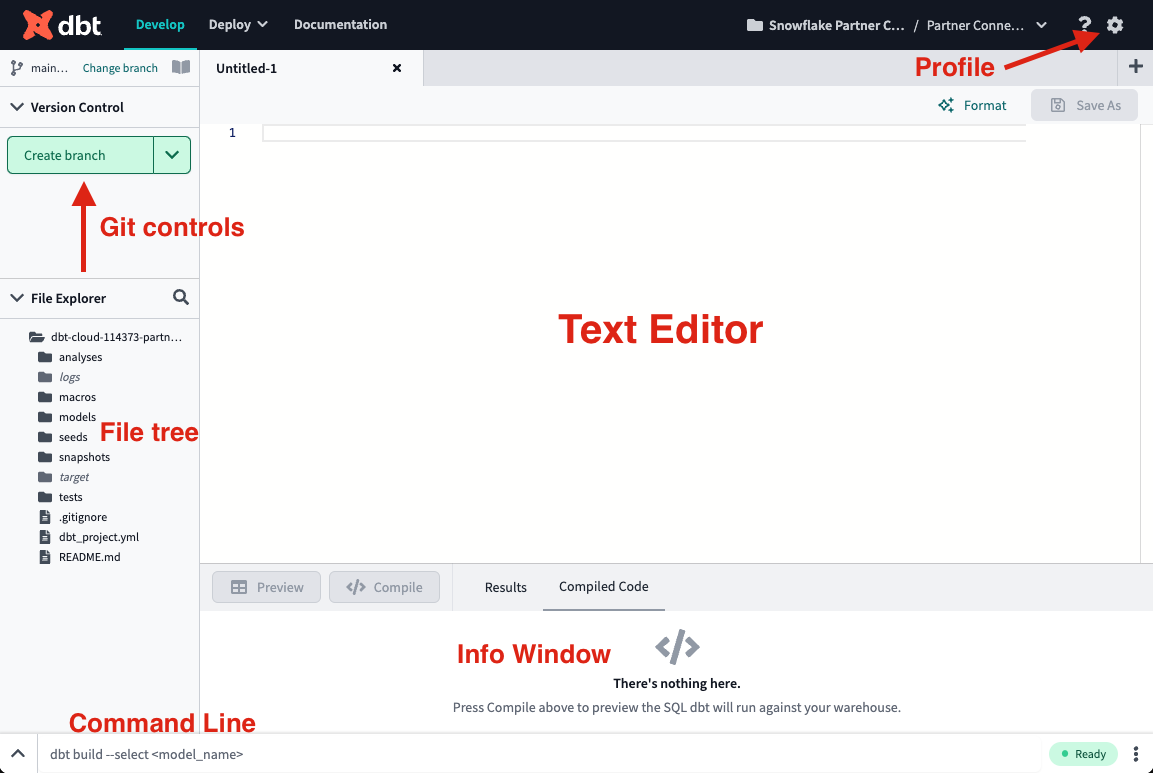
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Initialize dbt project

1. After the initialization is finished, you can view the files and folders in the file tree menu. As we move through the workshop we'll be sure to touch on a few key files and folders that we'll work with to build out our project.
2. Next click **Commit and push** to commit the new files and folders from the initialize step. We always want our commit messages to be relevant to the work we're committing, so be sure to provide a message like initialize project and select **Commit Changes**.

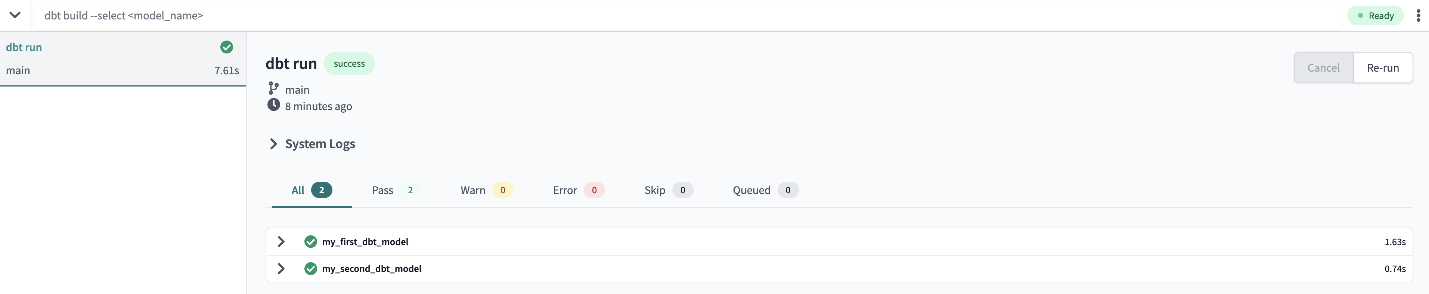
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)First commit and push

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Initialize project

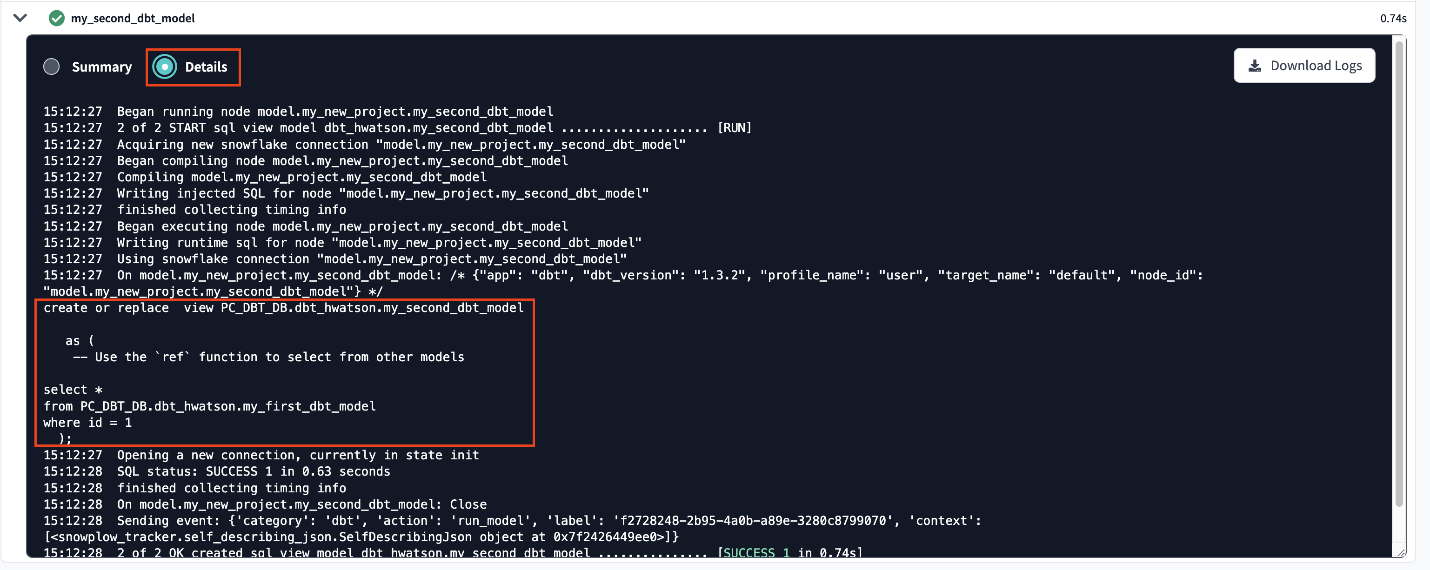
1. [Committing](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) your work here will save it to the managed git repository that was created during the Partner Connect signup. This initial commit is the only commit that will be made directly to our main branch and from *here on out we'll be doing all of our work on a development branch*. This allows us to keep our development work separate from our production code.
2. There are a couple of key features to point out about the IDE before we get to work. It is a text editor, an SQL and Python runner, and a CLI with Git version control all baked into one package! This allows you to focus on editing your SQL and Python files, previewing the results with the SQL runner (it even runs Jinja!), and building models at the command line without having to move between different applications. The Git workflow in dbt Cloud allows both Git beginners and experts alike to be able to easily version control all of their work with a couple clicks.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)IDE overview

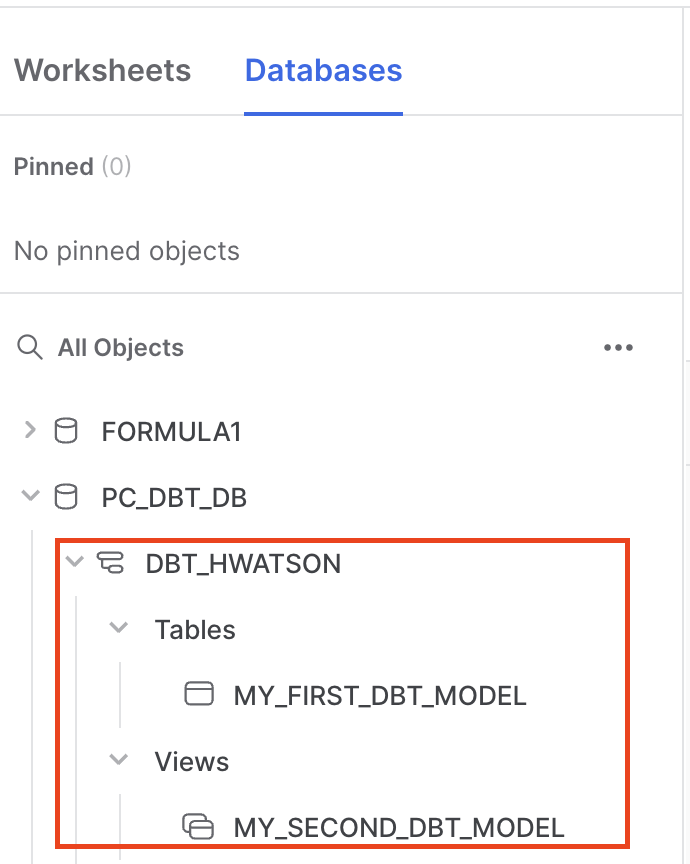
1. Let's run our first dbt models! Two example models are included in your dbt project in the models/examples folder that we can use to illustrate how to run dbt at the command line. Type dbt run into the command line and click **Enter** on your keyboard. When the run bar expands you'll be able to see the results of the run, where you should see the run complete successfully.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)dbt run example models

1. The run results allow you to see the code that dbt compiles and sends to Snowflake for execution. To view the logs for this run, select one of the model tabs using the  **>** icon and then **Details**. If you scroll down a bit you'll be able to see the compiled code and how dbt interacts with Snowflake. Given that this run took place in our development environment, the models were created in your development schema.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Details about the second model

1. Now let's switch over to Snowflake to confirm that the objects were actually created. Click on the three dots **…** above your database objects and then **Refresh**. Expand the **PC\_DBT\_DB** database and you should see your development schema. Select the schema, then **Tables**  and **Views**. Now you should be able to see MY\_FIRST\_DBT\_MODEL as a table and MY\_SECOND\_DBT\_MODEL as a view.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Confirm example models are built in Snowflake

**Create branch and set up project configs**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#create-branch-and-set-up-project-configs)

In this step, we’ll need to create a development branch and set up project level configurations.

1. To get started with development for our project, we'll need to create a new Git branch for our work. Select **create branch** and name your development branch. We'll call our branch snowpark\_python\_workshop then click **Submit**.
2. The first piece of development we'll do on the project is to update the dbt\_project.yml file. Every dbt project requires a dbt\_project.yml file — this is how dbt knows a directory is a dbt project. The [dbt\_project.yml](https://docs.getdbt.com/reference/dbt_project.yml) file also contains important information that tells dbt how to operate on your project.
3. Select the dbt\_project.yml file from the file tree to open it and replace all of the existing contents with the following code below. When you're done, save the file by clicking **save**. You can also use the Command-S or Control-S shortcut from here on out.

*# Name your project! Project names should contain only lowercase characters*  
*# and underscores. A good package name should reflect your organization's*  
*# name or the intended use of these models*  
name: 'snowflake\_dbt\_python\_formula1'  
version: '1.3.0'  
require-dbt-version: '>=1.3.0'  
config-version: 2  
  
*# This setting configures which "profile" dbt uses for this project.*  
profile: 'default'  
  
*# These configurations specify where dbt should look for different types of files.*  
*# The `model-paths` config, for example, states that models in this project can be*  
*# found in the "models/" directory. You probably won't need to change these!*  
model-paths: ["models"]  
analysis-paths: ["analyses"]  
test-paths: ["tests"]  
seed-paths: ["seeds"]  
macro-paths: ["macros"]  
snapshot-paths: ["snapshots"]  
  
target-path: "target" *# directory which will store compiled SQL files*  
clean-targets: *# directories to be removed by `dbt clean`*  
 - "target"  
 - "dbt\_packages"  
  
models:  
 snowflake\_dbt\_python\_formula1:  
 staging:  
  
 +docs:  
 node\_color: "CadetBlue"  
 marts:  
 +materialized: table  
 aggregates:  
 +docs:  
 node\_color: "Maroon"  
 +tags: "bi"  
  
 core:  
 +docs:  
 node\_color: "#800080"  
 intermediate:  
 +docs:  
 node\_color: "MediumSlateBlue"  
 ml:  
 prep:  
 +docs:  
 node\_color: "Indigo"  
 train\_predict:  
 +docs:  
 node\_color: "#36454f"

1. The key configurations to point out in the file with relation to the work that we're going to do are in the models section.
   * require-dbt-version — Tells dbt which version of dbt to use for your project. We are requiring 1.3.0 and any newer version to run python models and node colors.
   * materialized — Tells dbt how to materialize models when compiling the code before it pushes it down to Snowflake. All models in the marts folder will be built as tables.
   * tags — Applies tags at a directory level to all models. All models in the aggregates folder will be tagged as bi (abbreviation for business intelligence).
   * docs — Specifies the node\_color either by the plain color name or a hex value.
2. [Materializations](https://docs.getdbt.com/docs/build/materializations) are strategies for persisting dbt models in a warehouse, with tables and views being the most commonly utilized types. By default, all dbt models are materialized as views and other materialization types can be configured in the dbt\_project.yml file or in a model itself. It’s very important to note *Python models can only be materialized as tables or incremental models.* Since all our Python models exist under marts, the following portion of our dbt\_project.yml ensures no errors will occur when we run our Python models. Starting with [dbt version 1.4](https://docs.getdbt.com/docs/dbt-versions/core-upgrade/upgrading-to-v1.4" \l "updates-to-python-models), Python files will automatically get materialized as tables even if not explicitly specified.

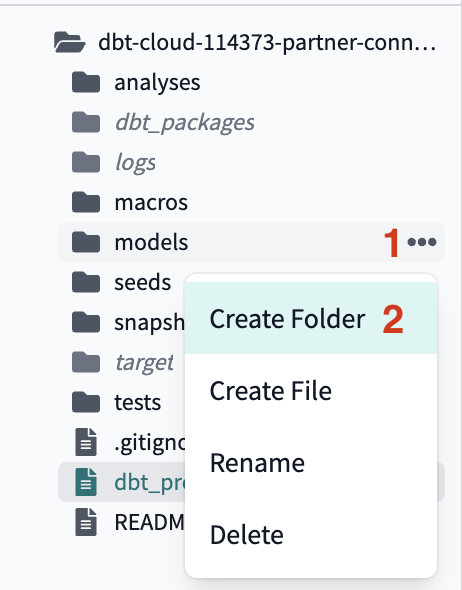
marts:       
 +materialized: table

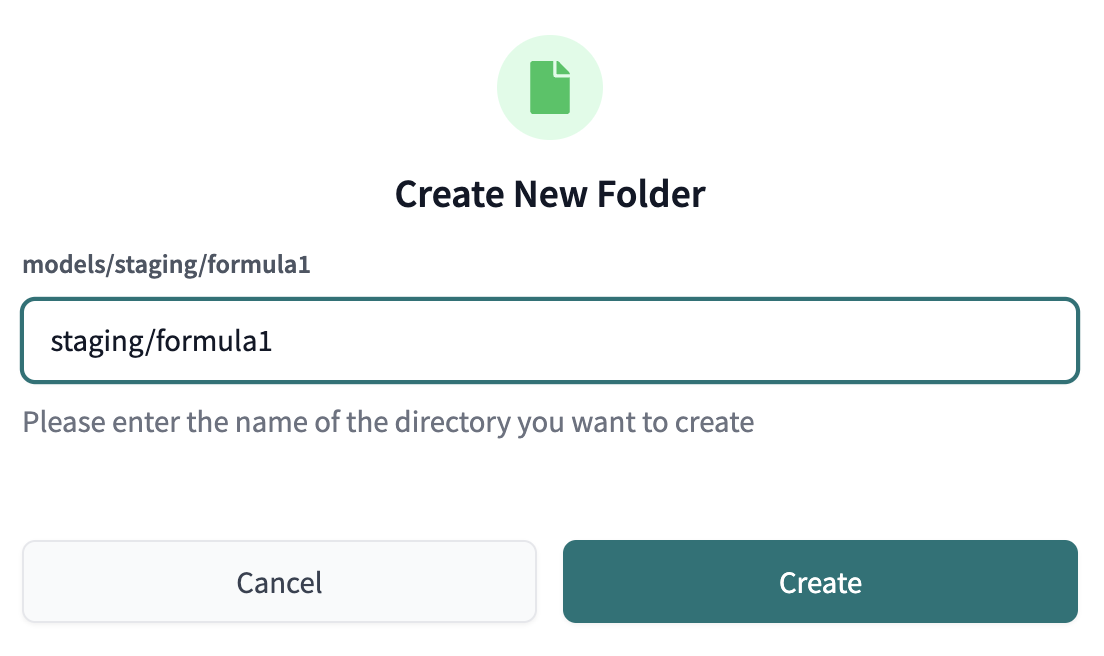
**Create folders and organize files**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#create-folders-and-organize-files)

dbt Labs has developed a [project structure guide](https://docs.getdbt.com/best-practices/how-we-structure/1-guide-overview) that contains a number of recommendations for how to build the folder structure for your project. Do check out that guide if you want to learn more. Right now we are going to create some folders to organize our files:

* Sources — This is our Formula 1 dataset and it will be defined in a source YAML file.
* Staging models — These models have a 1:1 with their source table.
* Intermediate — This is where we will be joining some Formula staging models.
* Marts models — Here is where we perform our major transformations. It contains these subfolders:
  + aggregates
  + core
  + ml

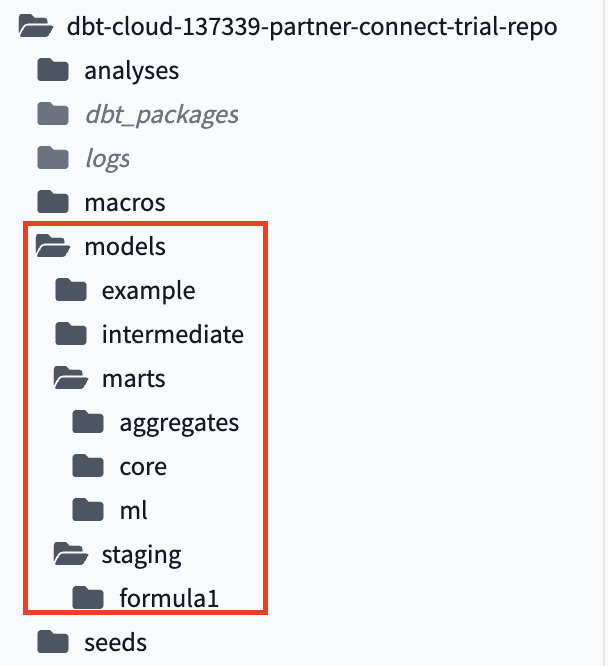
1. In your file tree, use your cursor and hover over the models subdirectory, click the three dots **…** that appear to the right of the folder name, then select **Create Folder**. We're going to add two new folders to the file path, staging and formula1 (in that order) by typing staging/formula1 into the file path.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Create folder

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Set file path

* + If you click into your models directory now, you should see the new staging folder nested within models and the formula1 folder nested within staging.

1. Create two additional folders the same as the last step. Within the models subdirectory, create new directories marts/core.
2. We will need to create a few more folders and subfolders using the UI. After you create all the necessary folders, your folder tree should look like this when it's all done:

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)File tree of new folders

Remember you can always reference the entire project in [GitHub](https://github.com/dbt-labs/python-snowpark-formula1/tree/python-formula1) to view the complete folder and file strucutre.

**Create source and staging models**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#create-source-and-staging-models)

In this section, we are going to create our source and staging models.

Sources allow us to create a dependency between our source database object and our staging models which will help us when we look at data lineage later. Also, if your source changes database or schema, you only have to update it in your f1\_sources.yml file rather than updating all of the models it might be used in.

Staging models are the base of our project, where we bring all the individual components we're going to use to build our more complex and useful models into the project.

Since we want to focus on dbt and Python in this workshop, check out our [sources](https://docs.getdbt.com/docs/build/sources) and [staging](https://docs.getdbt.com/best-practices/how-we-structure/2-staging) docs if you want to learn more (or take our [dbt Fundamentals](https://courses.getdbt.com/collections" \t "_blank) course which covers all of our core functionality).

**1. Create sources**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#1-create-sources)

We're going to be using each of our 8 Formula 1 tables from our formula1 database under the raw  schema for our transformations and we want to create those tables as sources in our project.

1. Create a new file called f1\_sources.yml with the following file path: models/staging/formula1/f1\_sources.yml.
2. Then, paste the following code into the file before saving it:

version: 2  
  
sources:  
 - name: formula1  
 description: formula 1 datasets with normalized tables   
 database: formula1   
 schema: raw  
 tables:  
 - name: circuits  
 description: One record per circuit, which is the specific race course.   
 columns:  
 - name: circuitid  
 tests:  
 - unique  
 - not\_null  
 - name: constructors   
 description: One record per constructor. Constructors are the teams that build their formula 1 cars.   
 columns:  
 - name: constructorid  
 tests:  
 - unique  
 - not\_null  
 - name: drivers  
 description: One record per driver. This table gives details about the driver.   
 columns:  
 - name: driverid  
 tests:  
 - unique  
 - not\_null  
 - name: lap\_times  
 description: One row per lap in each race. Lap times started being recorded in this dataset in 1984 and joined through driver\_id.  
 - name: pit\_stops   
 description: One row per pit stop. Pit stops do not have their own id column, the combination of the race\_id and driver\_id identify the pit stop.  
 columns:  
 - name: stop  
 tests:  
 - accepted\_values:  
 values: [1,2,3,4,5,6,7,8]  
 quote: false   
 - name: races   
 description: One race per row. Importantly this table contains the race year to understand trends.   
 columns:  
 - name: raceid  
 tests:  
 - unique  
 - not\_null   
 - name: results  
 columns:  
 - name: resultid  
 tests:  
 - unique  
 - not\_null   
 description: One row per result. The main table that we join out for grid and position variables.  
 - name: status  
 description: One status per row. The status contextualizes whether the race was finished or what issues arose e.g. collisions, engine, etc.   
 columns:  
 - name: statusid  
 tests:  
 - unique  
 - not\_null

**2. Create staging models**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#2-create-staging-models)

The next step is to set up the staging models for each of the 8 source tables. Given the one-to-one relationship between staging models and their corresponding source tables, we'll build 8 staging models here. We know it’s a lot and in the future, we will seek to update the workshop to make this step less repetitive and more efficient. This step is also a good representation of the real world of data, where you have multiple hierarchical tables that you will need to join together!

1. Let's go in alphabetical order to easily keep track of all our staging models! Create a new file called stg\_f1\_circuits.sql with this file path models/staging/formula1/stg\_f1\_circuits.sql. Then, paste the following code into the file before saving it:

with  
  
source as (  
  
 select \* from {{ source('formula1','circuits') }}  
  
),  
  
renamed as (  
 select  
 circuitid as circuit\_id,  
 circuitref as circuit\_ref,  
 name as circuit\_name,  
 location,  
 country,  
 lat as latitude,  
 lng as longitude,  
 alt as altitude  
 *-- omit the url*  
 from source  
)  
select \* from renamed

All we're doing here is pulling the source data into the model using the source function, renaming some columns, and omitting the column url with a commented note since we don’t need it for our analysis.

1. Create stg\_f1\_constructors.sql with this file path models/staging/formula1/stg\_f1\_constructors.sql. Paste the following code into it before saving the file:

with  
  
source as (  
  
 select \* from {{ source('formula1','constructors') }}  
  
),  
  
renamed as (  
 select  
 constructorid as constructor\_id,  
 constructorref as constructor\_ref,  
 name as constructor\_name,  
 nationality as constructor\_nationality  
 *-- omit the url*  
 from source  
)  
  
select \* from renamed

We have 6 other stages models to create. We can do this by creating new files, then copy and paste the code into our staging folder.

1. Create stg\_f1\_drivers.sql with this file path models/staging/formula1/stg\_f1\_drivers.sql:

with  
  
source as (  
  
 select \* from {{ source('formula1','drivers') }}  
  
),  
  
renamed as (  
 select  
 driverid as driver\_id,  
 driverref as driver\_ref,  
 number as driver\_number,  
 code as driver\_code,  
 forename,  
 surname,  
 dob as date\_of\_birth,  
 nationality as driver\_nationality  
 *-- omit the url*  
 from source  
)  
  
select \* from renamed

1. Create stg\_f1\_lap\_times.sql with this file path models/staging/formula1/stg\_f1\_lap\_times.sql:

with  
  
source as (  
  
 select \* from {{ source('formula1','lap\_times') }}  
  
),  
  
renamed as (  
 select  
 raceid as race\_id,  
 driverid as driver\_id,  
 lap,  
 position,  
 time as lap\_time\_formatted,  
 milliseconds as lap\_time\_milliseconds  
 from source  
)  
  
select \* from renamed

1. Create stg\_f1\_pit\_stops.sql with this file path models/staging/formula1/stg\_f1\_pit\_stops.sql:

with  
  
source as (  
  
 select \* from {{ source('formula1','pit\_stops') }}  
  
),  
  
renamed as (  
 select  
 raceid as race\_id,  
 driverid as driver\_id,  
 stop as stop\_number,  
 lap,  
 time as lap\_time\_formatted,  
 duration as pit\_stop\_duration\_seconds,  
 milliseconds as pit\_stop\_milliseconds  
 from source  
)  
  
select \* from renamed  
order by pit\_stop\_duration\_seconds desc

1. Create stg\_f1\_races.sql with this file path models/staging/formula1/stg\_f1\_races.sql:

with  
  
source as (  
  
 select \* from {{ source('formula1','races') }}  
  
),  
  
renamed as (  
 select  
 raceid as race\_id,  
 year as race\_year,  
 round as race\_round,  
 circuitid as circuit\_id,  
 name as circuit\_name,  
 date as race\_date,  
 to\_time(time) as race\_time,  
 *-- omit the url*  
 fp1\_date as free\_practice\_1\_date,  
 fp1\_time as free\_practice\_1\_time,  
 fp2\_date as free\_practice\_2\_date,  
 fp2\_time as free\_practice\_2\_time,  
 fp3\_date as free\_practice\_3\_date,  
 fp3\_time as free\_practice\_3\_time,  
 quali\_date as qualifying\_date,  
 quali\_time as qualifying\_time,  
 sprint\_date,  
 sprint\_time  
 from source  
)  
  
select \* from renamed

1. Create stg\_f1\_results.sql with this file path models/staging/formula1/stg\_f1\_results.sql:

with  
  
source as (  
  
 select \* from {{ source('formula1','results') }}  
  
),  
  
renamed as (  
 select  
 resultid as result\_id,  
 raceid as race\_id,  
 driverid as driver\_id,  
 constructorid as constructor\_id,  
 number as driver\_number,  
 grid,  
 position::int as position,  
 positiontext as position\_text,  
 positionorder as position\_order,  
 points,  
 laps,  
 time as results\_time\_formatted,  
 milliseconds as results\_milliseconds,  
 fastestlap as fastest\_lap,  
 rank as results\_rank,  
 fastestlaptime as fastest\_lap\_time\_formatted,  
 fastestlapspeed::decimal(6,3) as fastest\_lap\_speed,  
 statusid as status\_id  
 from source  
)  
  
select \* from renamed

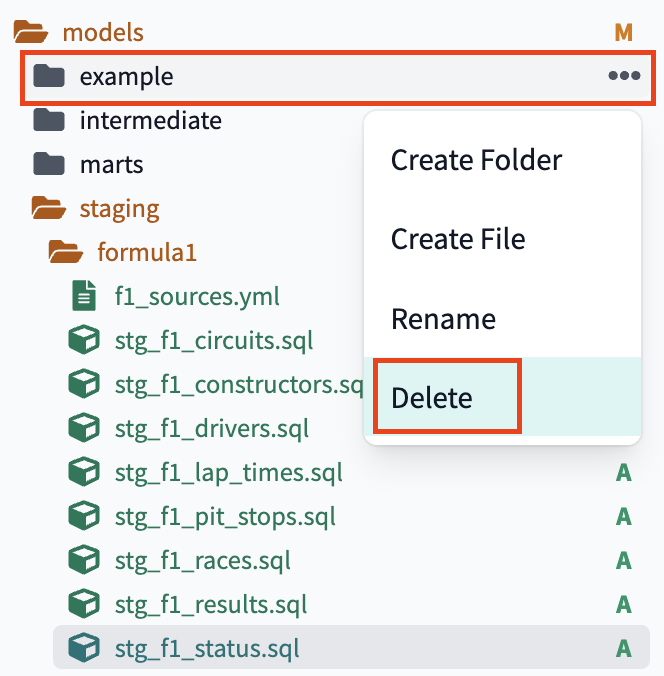
1. Last one! Create stg\_f1\_status.sql with this file path: models/staging/formula1/stg\_f1\_status.sql:

with  
  
source as (  
  
 select \* from {{ source('formula1','status') }}  
  
),  
  
renamed as (  
 select  
 statusid as status\_id,  
 status  
 from source  
)  
  
select \* from renamed

After the source and all the staging models are complete for each of the 8 tables, your staging folder should look like this:

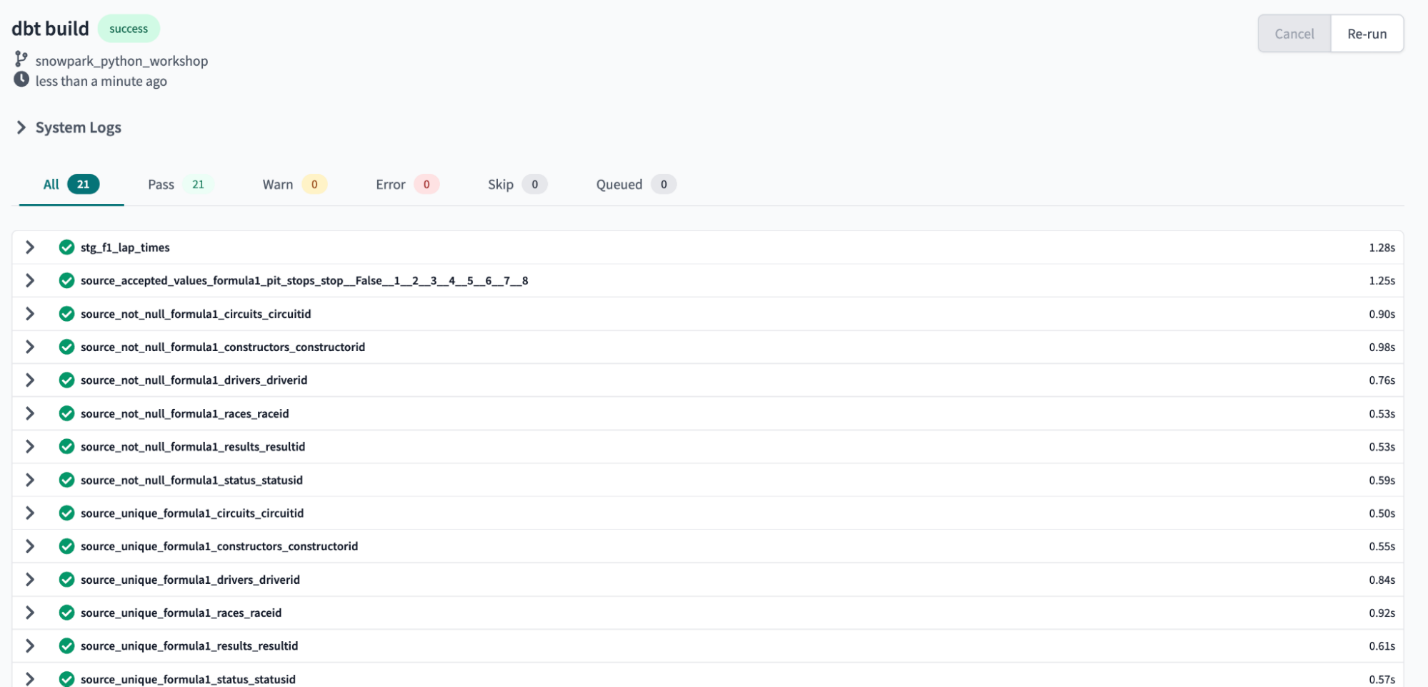
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Staging folder

1. It’s a good time to delete our example folder since these two models are extraneous to our formula1 pipeline and my\_first\_model fails a not\_null test that we won’t spend time investigating. dbt Cloud will warn us that this folder will be permanently deleted, and we are okay with that so select **Delete**.

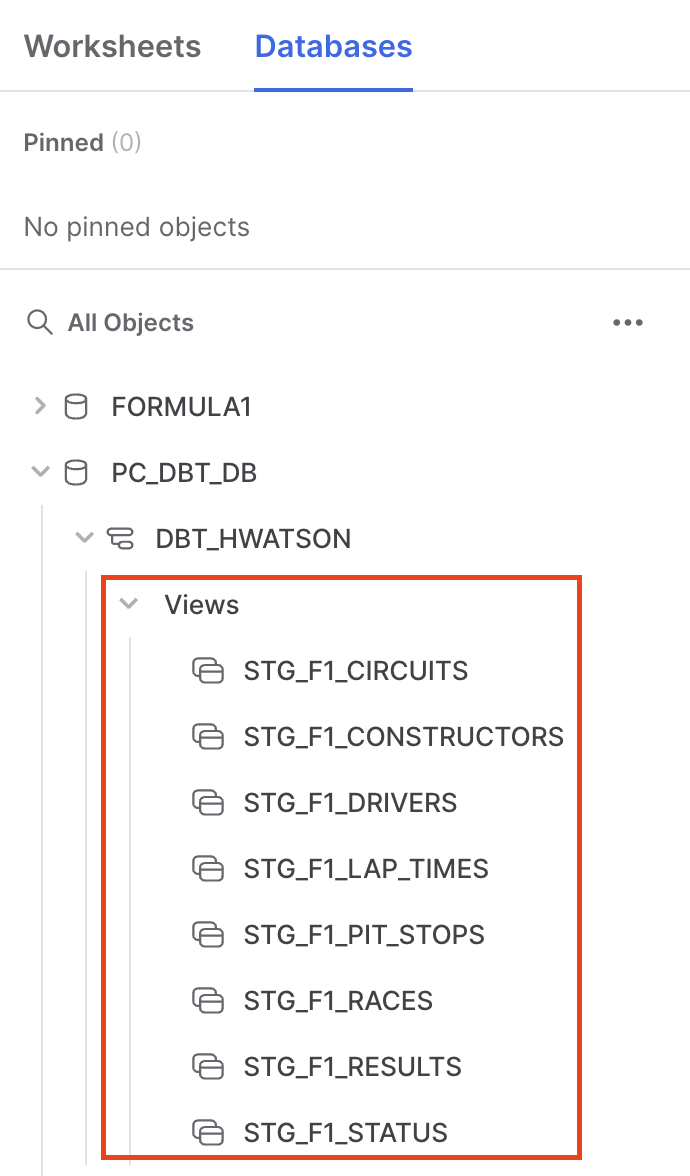
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Delete example folder

1. Now that the staging models are built and saved, it's time to create the models in our development schema in Snowflake. To do this we're going to enter into the command line dbt build to run all of the models in our project, which includes the 8 new staging models and the existing example models.

Your run should complete successfully and you should see green checkmarks next to all of your models in the run results. We built our 8 staging models as views and ran 13 source tests that we configured in the f1\_sources.yml file with not that much code, pretty cool!

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Successful dbt build in Snowflake

Let's take a quick look in Snowflake, refresh database objects, open our development schema, and confirm that the new models are there. If you can see them, then we're good to go!

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Confirm models

Before we move onto the next section, be sure to commit your new models to your Git branch. Click **Commit and push** and give your commit a message like profile, sources, and staging setup before moving on.

**Transform SQL**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#transform-sql)

Now that we have all our sources and staging models done, it's time to move into where dbt shines — transformation!

We need to:

* Create some intermediate tables to join tables that aren’t hierarchical
* Create core tables for business intelligence (BI) tool ingestion
* Answer the two questions about:
  + fastest pit stops
  + lap time trends about our Formula 1 data by creating aggregate models using python!

**Intermediate models**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#intermediate-models)

We need to join lots of reference tables to our results table to create a human readable dataframe. What does this mean? For example, we don’t only want to have the numeric status\_id in our table, we want to be able to read in a row of data that a driver could not finish a race due to engine failure (status\_id=5).

By now, we are pretty good at creating new files in the correct directories so we won’t cover this in detail. All intermediate models should be created in the path models/intermediate.

1. Create a new file called int\_lap\_times\_years.sql. In this model, we are joining our lap time and race information so we can look at lap times over years. In earlier Formula 1 eras, lap times were not recorded (only final results), so we filter out records where lap times are null.

with lap\_times as (  
  
 select \* from {{ ref('stg\_f1\_lap\_times') }}  
  
),  
  
races as (  
  
 select \* from {{ ref('stg\_f1\_races') }}  
  
),  
  
expanded\_lap\_times\_by\_year as (  
 select  
 lap\_times.race\_id,  
 driver\_id,  
 race\_year,  
 lap,  
 lap\_time\_milliseconds  
 from lap\_times  
 left join races  
 on lap\_times.race\_id = races.race\_id  
 where lap\_time\_milliseconds is not null  
)  
  
select \* from expanded\_lap\_times\_by\_year

1. Create a file called in\_pit\_stops.sql. Pit stops are a many-to-one (M:1) relationship with our races. We are creating a feature called total\_pit\_stops\_per\_race by partitioning over our race\_id and driver\_id, while preserving individual level pit stops for rolling average in our next section.

with stg\_f1\_\_pit\_stops as  
(  
 select \* from {{ ref('stg\_f1\_pit\_stops') }}  
),  
  
pit\_stops\_per\_race as (  
 select  
 race\_id,  
 driver\_id,  
 stop\_number,  
 lap,  
 lap\_time\_formatted,  
 pit\_stop\_duration\_seconds,  
 pit\_stop\_milliseconds,  
 max(stop\_number) over (partition by race\_id,driver\_id) as total\_pit\_stops\_per\_race  
 from stg\_f1\_\_pit\_stops  
)  
  
select \* from pit\_stops\_per\_race

1. Create a file called int\_results.sql. Here we are using 4 of our tables — races, drivers, constructors, and status — to give context to our results table. We are now able to calculate a new feature drivers\_age\_years by bringing the date\_of\_birth and race\_year into the same table. We are also creating a column to indicate if the driver did not finish (dnf) the race, based upon if their position was null called, dnf\_flag.

with results as (  
  
 select \* from {{ ref('stg\_f1\_results') }}  
  
),  
  
races as (  
  
 select \* from {{ ref('stg\_f1\_races') }}  
  
),  
  
drivers as (  
  
 select \* from {{ ref('stg\_f1\_drivers') }}  
  
),  
  
constructors as (  
  
 select \* from {{ ref('stg\_f1\_constructors') }}  
),  
  
status as (  
  
 select \* from {{ ref('stg\_f1\_status') }}  
),  
  
int\_results as (  
 select  
 result\_id,  
 results.race\_id,  
 race\_year,  
 race\_round,  
 circuit\_id,  
 circuit\_name,  
 race\_date,  
 race\_time,  
 results.driver\_id,  
 results.driver\_number,  
 forename ||' '|| surname as driver,  
 cast(datediff('year', date\_of\_birth, race\_date) as int) as drivers\_age\_years,  
 driver\_nationality,  
 results.constructor\_id,  
 constructor\_name,  
 constructor\_nationality,  
 grid,  
 position,  
 position\_text,  
 position\_order,  
 points,  
 laps,  
 results\_time\_formatted,  
 results\_milliseconds,  
 fastest\_lap,  
 results\_rank,  
 fastest\_lap\_time\_formatted,  
 fastest\_lap\_speed,  
 results.status\_id,  
 status,  
 case when position is null then 1 else 0 end as dnf\_flag  
 from results  
 left join races  
 on results.race\_id=races.race\_id  
 left join drivers  
 on results.driver\_id = drivers.driver\_id  
 left join constructors  
 on results.constructor\_id = constructors.constructor\_id  
 left join status  
 on results.status\_id = status.status\_id  
)  
  
select \* from int\_results

1. Create a *Markdown* file intermediate.md that we will go over in depth in the Test and Documentation sections of the [Leverage dbt Cloud to generate analytics and ML-ready pipelines with SQL and Python with Snowflake](https://docs.getdbt.com/guides/dbt-python-snowpark) guide.

# the intent of this .md is to allow for multi-line long form explanations for our intermediate transformations  
  
# below are descriptions   
{% docs int\_results %} In this query we want to join out other important information about the race results to have a human readable table about results, races, drivers, constructors, and status.   
We will have 4 left joins onto our results table. {% enddocs %}  
  
{% docs int\_pit\_stops %} There are many pit stops within one race, aka a M:1 relationship.   
We want to aggregate this so we can properly join pit stop information without creating a fanout. {% enddocs %}  
  
{% docs int\_lap\_times\_years %} Lap times are done per lap. We need to join them out to the race year to understand yearly lap time trends. {% enddocs %}

1. Create a *YAML* file intermediate.yml that we will go over in depth during the Test and Document sections of the [Leverage dbt Cloud to generate analytics and ML-ready pipelines with SQL and Python with Snowflake](https://docs.getdbt.com/guides/dbt-python-snowpark) guide.

version: 2  
  
models:  
 - name: int\_results  
 description: '{{ doc("int\_results") }}'  
 - name: int\_pit\_stops  
 description: '{{ doc("int\_pit\_stops") }}'  
 - name: int\_lap\_times\_years  
 description: '{{ doc("int\_lap\_times\_years") }}'

That wraps up the intermediate models we need to create our core models!

**Core models**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#core-models)

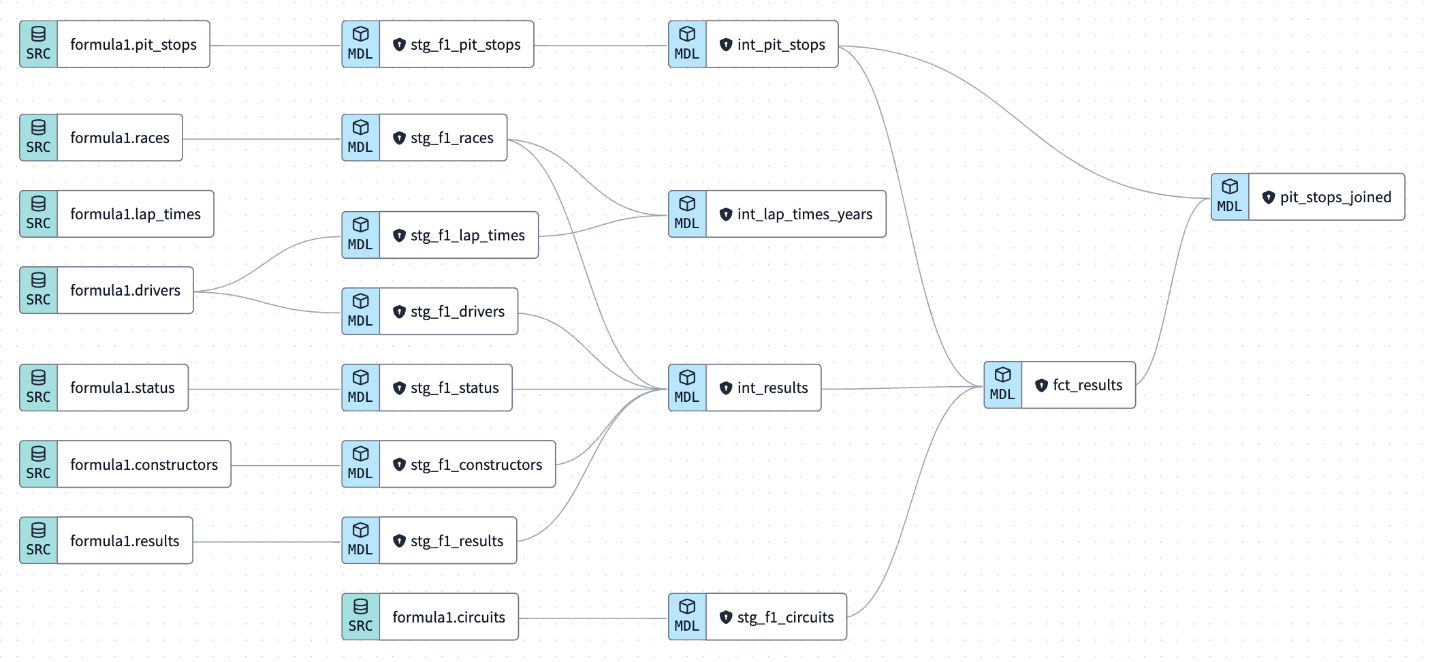
1. Create a file fct\_results.sql. This is what I like to refer to as the “mega table” — a really large denormalized table with all our context added in at row level for human readability. Importantly, we have a table circuits that is linked through the table races. When we joined races to results in int\_results.sql we allowed our tables to make the connection from circuits to results in fct\_results.sql. We are only taking information about pit stops at the result level so our join would not cause a [fanout](https://community.looker.com/technical-tips-tricks-1021/what-is-a-fanout-23327).

with int\_results as (  
  
 select \* from {{ ref('int\_results') }}  
  
),  
  
int\_pit\_stops as (  
 select  
 race\_id,  
 driver\_id,  
 max(total\_pit\_stops\_per\_race) as total\_pit\_stops\_per\_race  
 from {{ ref('int\_pit\_stops') }}  
 group by 1,2  
),  
  
circuits as (  
  
 select \* from {{ ref('stg\_f1\_circuits') }}  
),  
base\_results as (  
 select  
 result\_id,  
 int\_results.race\_id,  
 race\_year,  
 race\_round,  
 int\_results.circuit\_id,  
 int\_results.circuit\_name,  
 circuit\_ref,  
 location,  
 country,  
 latitude,  
 longitude,  
 altitude,  
 total\_pit\_stops\_per\_race,  
 race\_date,  
 race\_time,  
 int\_results.driver\_id,  
 driver,  
 driver\_number,  
 drivers\_age\_years,  
 driver\_nationality,  
 constructor\_id,  
 constructor\_name,  
 constructor\_nationality,  
 grid,  
 position,  
 position\_text,  
 position\_order,  
 points,  
 laps,  
 results\_time\_formatted,  
 results\_milliseconds,  
 fastest\_lap,  
 results\_rank,  
 fastest\_lap\_time\_formatted,  
 fastest\_lap\_speed,  
 status\_id,  
 status,  
 dnf\_flag  
 from int\_results  
 left join circuits  
 on int\_results.circuit\_id=circuits.circuit\_id  
 left join int\_pit\_stops  
 on int\_results.driver\_id=int\_pit\_stops.driver\_id and int\_results.race\_id=int\_pit\_stops.race\_id  
)  
  
select \* from base\_results

1. Create the file pit\_stops\_joined.sql. Our results and pit stops are at different levels of dimensionality (also called grain). Simply put, we have multiple pit stops per a result. Since we are interested in understanding information at the pit stop level with information about race year and constructor, we will create a new table pit\_stops\_joined.sql where each row is per pit stop. Our new table tees up our aggregation in Python.

with base\_results as (  
  
 select \* from {{ ref('fct\_results') }}  
  
),   
  
pit\_stops as (  
  
 select \* from {{ ref('int\_pit\_stops') }}  
  
),  
  
pit\_stops\_joined as (  
  
 select   
 base\_results.race\_id,  
 race\_year,  
 base\_results.driver\_id,  
 constructor\_id,  
 constructor\_name,  
 stop\_number,  
 lap,   
 lap\_time\_formatted,  
 pit\_stop\_duration\_seconds,   
 pit\_stop\_milliseconds  
 from base\_results  
 left join pit\_stops  
 on base\_results.race\_id=pit\_stops.race\_id and base\_results.driver\_id=pit\_stops.driver\_id  
)  
select \* from pit\_stops\_joined

1. Enter in the command line and execute dbt build to build out our entire pipeline to up to this point. Don’t worry about “overriding” your previous models – dbt workflows are designed to be idempotent so we can run them again and expect the same results.
2. Let’s talk about our lineage so far. It’s looking good 😎. We’ve shown how SQL can be used to make data type, column name changes, and handle hierarchical joins really well; all while building out our automated lineage!

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)The DAG

1. Time to **Commit and push** our changes and give your commit a message like intermediate and fact models before moving on.

**Running dbt Python models**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#running-dbt-python-models)

Up until now, SQL has been driving the project (car pun intended) for data cleaning and hierarchical joining. Now it’s time for Python to take the wheel (car pun still intended) for the rest of our lab! For more information about running Python models on dbt, check out our [docs](https://docs.getdbt.com/docs/build/python-models). To learn more about dbt python works under the hood, check out [Snowpark for Python](https://docs.snowflake.com/en/developer-guide/snowpark/python/index.html), which makes running dbt Python models possible.

There are quite a few differences between SQL and Python in terms of the dbt syntax and DDL, so we’ll be breaking our code and model runs down further for our python models.

**Pit stop analysis**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#pit-stop-analysis)

First, we want to find out: which constructor had the fastest pit stops in 2021? (constructor is a Formula 1 team that builds or “constructs” the car).

1. Create a new file called fastest\_pit\_stops\_by\_constructor.py in our aggregates (this is the first time we are using the .py extension!).
2. Copy the following code into the file:

import numpy as np  
import pandas as pd  
  
def model(dbt, session):  
 *# dbt configuration*  
 dbt.config(packages=["pandas","numpy"])  
  
 *# get upstream data*  
 pit\_stops\_joined = dbt.ref("pit\_stops\_joined").to\_pandas()  
  
 *# provide year so we do not hardcode dates*   
 year=2021  
  
 *# describe the data*  
 pit\_stops\_joined["PIT\_STOP\_SECONDS"] = pit\_stops\_joined["PIT\_STOP\_MILLISECONDS"]/1000  
 fastest\_pit\_stops = pit\_stops\_joined[(pit\_stops\_joined["RACE\_YEAR"]==year)].groupby(by="CONSTRUCTOR\_NAME")["PIT\_STOP\_SECONDS"].describe().sort\_values(by='mean')  
 fastest\_pit\_stops.reset\_index(inplace=True)  
 fastest\_pit\_stops.columns = fastest\_pit\_stops.columns.str.upper()  
   
 return fastest\_pit\_stops.round(2)

1. Let’s break down what this code is doing step by step:
   * First, we are importing the Python libraries that we are using. A *library* is a reusable chunk of code that someone else wrote that you may want to include in your programs/projects. We are using numpy and pandasin this Python model. This is similar to a dbt *package*, but our Python libraries do *not* persist across the entire project.
   * Defining a function called model with the parameter dbt and session. The parameter dbt is a class compiled by dbt, which enables you to run your Python code in the context of your dbt project and DAG. The parameter session is a class representing your Snowflake’s connection to the Python backend. The model function *must return a single DataFrame*. You can see that all the data transformation happening is within the body of the model function that the return statement is tied to.
   * Then, within the context of our dbt model library, we are passing in a configuration of which packages we need using dbt.config(packages=["pandas","numpy"]).
   * Use the .ref() function to retrieve the data frame pit\_stops\_joined that we created in our last step using SQL. We cast this to a pandas dataframe (by default it's a Snowpark Dataframe).
   * Create a variable named year so we aren’t passing a hardcoded value.
   * Generate a new column called PIT\_STOP\_SECONDS by dividing the value of PIT\_STOP\_MILLISECONDS by 1000.
   * Create our final data frame fastest\_pit\_stops that holds the records where year is equal to our year variable (2021 in this case), then group the data frame by CONSTRUCTOR\_NAME and use the describe() and sort\_values() and in descending order. This will make our first row in the new aggregated data frame the team with the fastest pit stops over an entire competition year.
   * Finally, it resets the index of the fastest\_pit\_stops data frame. The reset\_index() method allows you to reset the index back to the default 0, 1, 2, etc indexes. By default, this method will keep the "old" indexes in a column named "index"; to avoid this, use the drop parameter. Think of this as keeping your data “flat and square” as opposed to “tiered”. If you are new to Python, now might be a good time to [learn about indexes for 5 minutes](https://towardsdatascience.com/the-basics-of-indexing-and-slicing-python-lists-2d12c90a94cf) since it's the foundation of how Python retrieves, slices, and dices data. The inplace argument means we override the existing data frame permanently. Not to fear! This is what we want to do to avoid dealing with multi-indexed dataframes!
   * Convert our Python column names to all uppercase using .upper(), so Snowflake recognizes them.
   * Finally we are returning our dataframe with 2 decimal places for all the columns using the round() method.
2. Zooming out a bit, what are we doing differently here in Python from our typical SQL code:
   * Method chaining is a technique in which multiple methods are called on an object in a single statement, with each method call modifying the result of the previous one. The methods are called in a chain, with the output of one method being used as the input for the next one. The technique is used to simplify the code and make it more readable by eliminating the need for intermediate variables to store the intermediate results.
     + The way you see method chaining in Python is the syntax .().(). For example, .describe().sort\_values(by='mean') where the .describe() method is chained to .sort\_values().
   * The .describe() method is used to generate various summary statistics of the dataset. It's used on pandas dataframe. It gives a quick and easy way to get the summary statistics of your dataset without writing multiple lines of code.
   * The .sort\_values() method is used to sort a pandas dataframe or a series by one or multiple columns. The method sorts the data by the specified column(s) in ascending or descending order. It is the pandas equivalent to order by in SQL.

We won’t go as in depth for our subsequent scripts, but will continue to explain at a high level what new libraries, functions, and methods are doing.

1. Build the model using the UI which will **execute**:

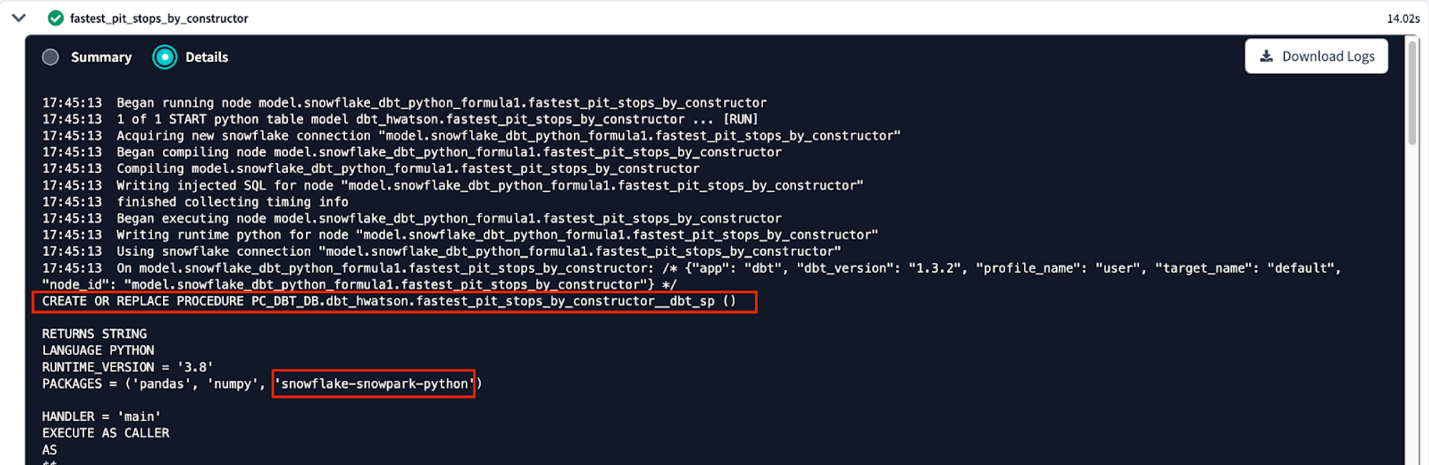
dbt run --select fastest\_pit\_stops\_by\_constructor

in the command bar.

Let’s look at some details of our first Python model to see what our model executed. There two major differences we can see while running a Python model compared to an SQL model:

* + Our Python model was executed as a stored procedure. Snowflake needs a way to know that it's meant to execute this code in a Python runtime, instead of interpreting in a SQL runtime. We do this by creating a Python stored proc, called by a SQL command.
  + The snowflake-snowpark-python library has been picked up to execute our Python code. Even though this wasn’t explicitly stated this is picked up by the dbt class object because we need our Snowpark package to run Python!

Python models take a bit longer to run than SQL models, however we could always speed this up by using [Snowpark-optimized Warehouses](https://docs.snowflake.com/en/user-guide/warehouses-snowpark-optimized.html) if we wanted to. Our data is sufficiently small, so we won’t worry about creating a separate warehouse for Python versus SQL files today.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)We can see our python model is run a stored procedure in our personal development schema

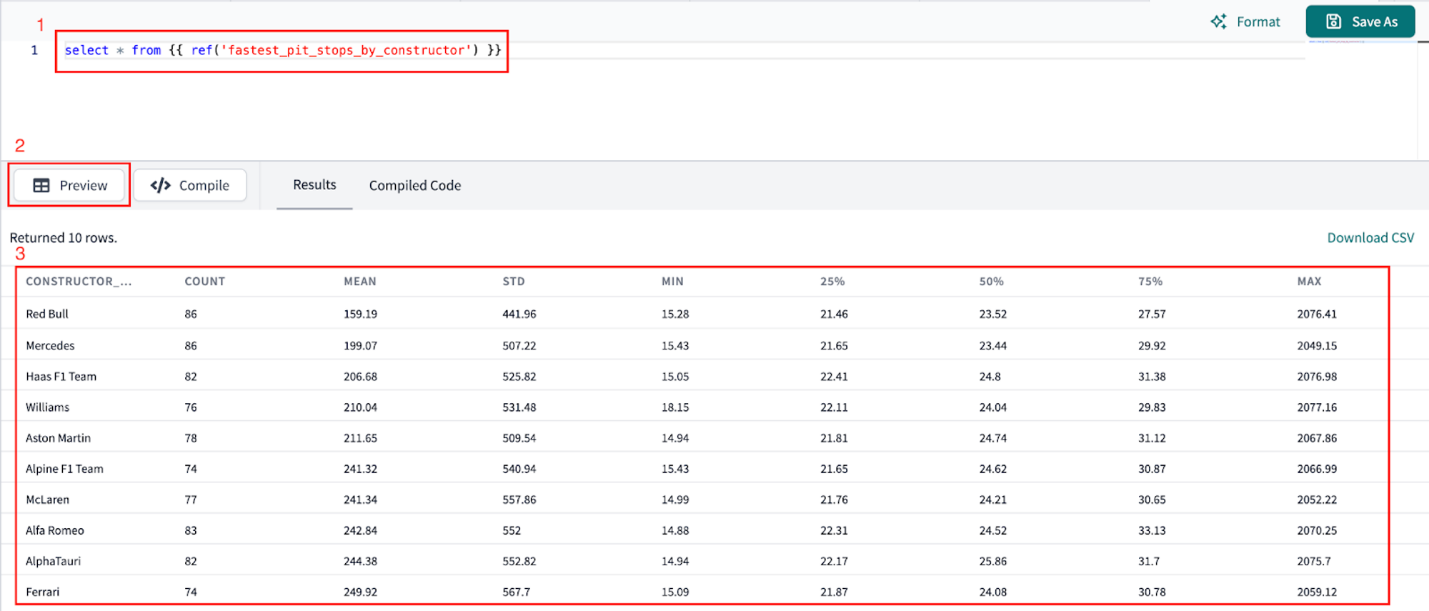
The rest of our **Details** output gives us information about how dbt and Snowpark for Python are working together to define class objects and apply a specific set of methods to run our models.

So which constructor had the fastest pit stops in 2021? Let’s look at our data to find out!

1. We can't preview Python models directly, so let’s create a new file using the **+** button or the Control-n shortcut to create a new scratchpad.
2. Reference our Python model:

select \* from {{ ref('fastest\_pit\_stops\_by\_constructor') }}

and preview the output:

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Looking at our new python data model we can see that Red Bull had the fastest pit stops!

Not only did Red Bull have the fastest average pit stops by nearly 40 seconds, they also had the smallest standard deviation, meaning they are both fastest and most consistent teams in pit stops. By using the .describe() method we were able to avoid verbose SQL requiring us to create a line of code per column and repetitively use the PERCENTILE\_COUNT() function.

Now we want to find the lap time average and rolling average through the years (is it generally trending up or down)?

1. Create a new file called lap\_times\_moving\_avg.py in our aggregates folder.
2. Copy the following code into the file:

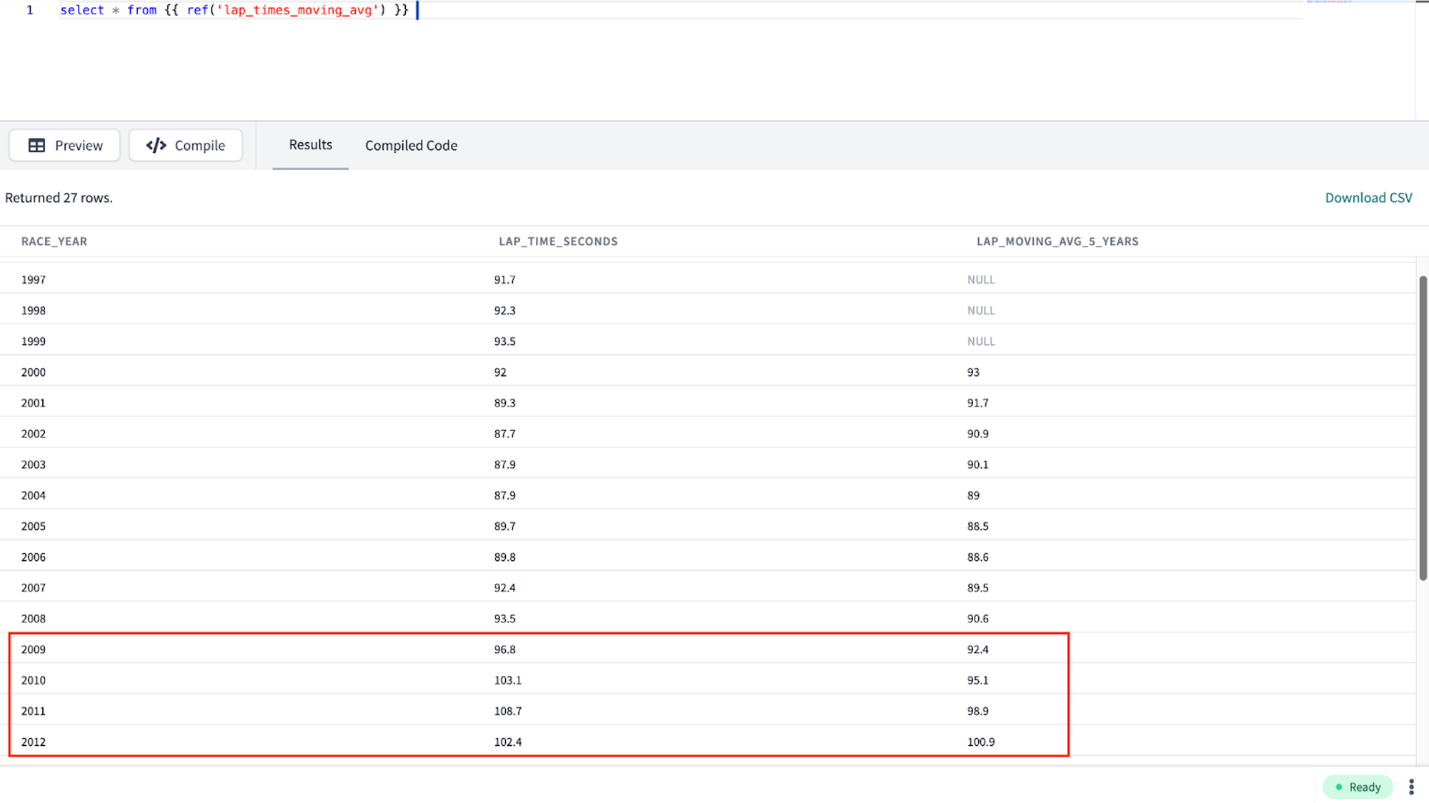
import pandas as pd  
  
def model(dbt, session):  
 *# dbt configuration*  
 dbt.config(packages=["pandas"])  
  
 *# get upstream data*  
 lap\_times = dbt.ref("int\_lap\_times\_years").to\_pandas()  
  
 *# describe the data*  
 lap\_times["LAP\_TIME\_SECONDS"] = lap\_times["LAP\_TIME\_MILLISECONDS"]/1000  
 lap\_time\_trends = lap\_times.groupby(by="RACE\_YEAR")["LAP\_TIME\_SECONDS"].mean().to\_frame()  
 lap\_time\_trends.reset\_index(inplace=True)  
 lap\_time\_trends["LAP\_MOVING\_AVG\_5\_YEARS"] = lap\_time\_trends["LAP\_TIME\_SECONDS"].rolling(5).mean()  
 lap\_time\_trends.columns = lap\_time\_trends.columns.str.upper()  
   
 return lap\_time\_trends.round(1)

1. Breaking down our code a bit:
   * We’re only using the pandas library for this model and casting it to a pandas data frame .to\_pandas().
   * Generate a new column called LAP\_TIMES\_SECONDS by dividing the value of LAP\_TIME\_MILLISECONDS by 1000.
   * Create the final dataframe. Get the lap time per year. Calculate the mean series and convert to a data frame.
   * Reset the index.
   * Calculate the rolling 5 year mean.
   * Round our numeric columns to one decimal place.
2. Now, run this model by using the UI **Run model** or

dbt run --select lap\_times\_moving\_avg

in the command bar.

1. Once again previewing the output of our data using the same steps for our fastest\_pit\_stops\_by\_constructor model.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Viewing our lap trends and 5 year rolling trends

We can see that it looks like lap times are getting consistently faster over time. Then in 2010 we see an increase occur! Using outside subject matter context, we know that significant rule changes were introduced to Formula 1 in 2010 and 2011 causing slower lap times.

1. Now is a good time to checkpoint and commit our work to Git. Click **Commit and push** and give your commit a message like aggregate python models before moving on.

**The dbt model, .source(), .ref() and .config() functions**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#the-dbt-model-source-ref-and-config-functions)

Let’s take a step back before starting machine learning to both review and go more in-depth at the methods that make running dbt python models possible. If you want to know more outside of this lab’s explanation read the documentation [here](https://docs.getdbt.com/docs/build/python-models?version=1.3).

* dbt model(dbt, session). For starters, 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.
* .source() and .ref() functions. Python models participate fully in dbt's directed acyclic graph (DAG) of transformations. If you want to read directly from a raw source table, use dbt.source(). We saw this in our earlier section using SQL with the source function. These functions have the same execution, but with different syntax. Use the dbt.ref() method within a Python model to read data from other models (SQL or Python). These methods return DataFrames pointing to the upstream source, model, seed, or snapshot.
* .config(). Just like SQL models, there are three ways to configure Python models:
  + In a dedicated .yml file, within the models/ directory
  + 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):  
  
 *# setting configuration*  
 dbt.config(materialized="table")  
 ```

* + There's a limit to how complex you can get with the dbt.config() method. It accepts only literal values (strings, booleans, and numeric types). Passing another function or a more complex data structure is not possible. The reason is that dbt statically analyzes the arguments to .config() while parsing your model without executing your Python code. If you need to set a more complex configuration, we recommend you define it using the config property in a [YAML file](https://docs.getdbt.com/reference/resource-properties/config). Learn more about configurations [here](https://docs.getdbt.com/reference/model-configs).

**Prepare for machine learning: cleaning, encoding, and splits**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#prepare-for-machine-learning-cleaning-encoding-and-splits)

Now that we’ve gained insights and business intelligence about Formula 1 at a descriptive level, we want to extend our capabilities into prediction. We’re going to take the scenario where we censor the data. This means that we will pretend that we will train a model using earlier data and apply it to future data. In practice, this means we’ll take data from 2010-2019 to train our model and then predict 2020 data.

In this section, we’ll be preparing our data to predict the final race position of a driver.

At a high level we’ll be:

* Creating new prediction features and filtering our dataset to active drivers
* Encoding our data (algorithms like numbers) and simplifying our target variable called position
* Splitting our dataset into training, testing, and validation

**ML data prep**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#ml-data-prep)

1. To keep our project organized, we’ll need to create two new subfolders in our ml directory. Under the ml folder, make the subfolders prep and train\_predict.
2. Create a new file under ml/prep called ml\_data\_prep.py. Copy the following code into the file and **Save**.

import pandas as pd  
  
def model(dbt, session):  
 *# dbt configuration*  
 dbt.config(packages=["pandas"])  
  
 *# get upstream data*  
 fct\_results = dbt.ref("fct\_results").to\_pandas()  
  
 *# provide years so we do not hardcode dates in filter command*  
 start\_year=2010  
 end\_year=2020  
  
 *# describe the data for a full decade*  
 data = fct\_results.loc[fct\_results['RACE\_YEAR'].between(start\_year, end\_year)]  
  
 *# convert string to an integer*  
 data['POSITION'] = data['POSITION'].astype(float)  
  
 *# we cannot have nulls if we want to use total pit stops*   
 data['TOTAL\_PIT\_STOPS\_PER\_RACE'] = data['TOTAL\_PIT\_STOPS\_PER\_RACE'].fillna(0)  
  
 *# some of the constructors changed their name over the year so replacing old names with current name*  
 mapping = {'Force India': 'Racing Point', 'Sauber': 'Alfa Romeo', 'Lotus F1': 'Renault', 'Toro Rosso': 'AlphaTauri'}  
 data['CONSTRUCTOR\_NAME'].replace(mapping, inplace=True)  
  
 *# create confidence metrics for drivers and constructors*  
 dnf\_by\_driver = data.groupby('DRIVER').sum(numeric\_only=True)['DNF\_FLAG']  
 driver\_race\_entered = data.groupby('DRIVER').count()['DNF\_FLAG']  
 driver\_dnf\_ratio = (dnf\_by\_driver/driver\_race\_entered)  
 driver\_confidence = 1-driver\_dnf\_ratio  
 driver\_confidence\_dict = dict(zip(driver\_confidence.index,driver\_confidence))  
  
 dnf\_by\_constructor = data.groupby('CONSTRUCTOR\_NAME').sum(numeric\_only=True)['DNF\_FLAG']  
 constructor\_race\_entered = data.groupby('CONSTRUCTOR\_NAME').count()['DNF\_FLAG']  
 constructor\_dnf\_ratio = (dnf\_by\_constructor/constructor\_race\_entered)  
 constructor\_relaiblity = 1-constructor\_dnf\_ratio  
 constructor\_relaiblity\_dict = dict(zip(constructor\_relaiblity.index,constructor\_relaiblity))  
  
 data['DRIVER\_CONFIDENCE'] = data['DRIVER'].apply(lambda x:driver\_confidence\_dict[x])  
 data['CONSTRUCTOR\_RELAIBLITY'] = data['CONSTRUCTOR\_NAME'].apply(lambda x:constructor\_relaiblity\_dict[x])  
  
 *#removing retired drivers and constructors*  
 active\_constructors = ['Renault', 'Williams', 'McLaren', 'Ferrari', 'Mercedes',  
 'AlphaTauri', 'Racing Point', 'Alfa Romeo', 'Red Bull',  
 'Haas F1 Team']  
 active\_drivers = ['Daniel Ricciardo', 'Kevin Magnussen', 'Carlos Sainz',  
 'Valtteri Bottas', 'Lance Stroll', 'George Russell',  
 'Lando Norris', 'Sebastian Vettel', 'Kimi Räikkönen',  
 'Charles Leclerc', 'Lewis Hamilton', 'Daniil Kvyat',  
 'Max Verstappen', 'Pierre Gasly', 'Alexander Albon',  
 'Sergio Pérez', 'Esteban Ocon', 'Antonio Giovinazzi',  
 'Romain Grosjean','Nicholas Latifi']  
  
 *# create flags for active drivers and constructors so we can filter downstream*   
 data['ACTIVE\_DRIVER'] = data['DRIVER'].apply(lambda x: int(x in active\_drivers))  
 data['ACTIVE\_CONSTRUCTOR'] = data['CONSTRUCTOR\_NAME'].apply(lambda x: int(x in active\_constructors))  
   
 return data

1. As usual, let’s break down what we are doing in this Python model:
   * We’re first referencing our upstream fct\_results table and casting it to a pandas dataframe.
   * Filtering on years 2010-2020 since we’ll need to clean all our data we are using for prediction (both training and testing).
   * Filling in empty data for total\_pit\_stops and making a mapping active constructors and drivers to avoid erroneous predictions
     + ⚠️ You might be wondering why we didn’t do this upstream in our fct\_results table! The reason for this is that we want our machine learning cleanup to reflect the year 2020 for our predictions and give us an up-to-date team name. However, for business intelligence purposes we can keep the historical data at that point in time. Instead of thinking of one table as “one source of truth” we are creating different datasets fit for purpose: one for historical descriptions and reporting and another for relevant predictions.
   * Create new confidence features for drivers and constructors
   * Generate flags for the constructors and drivers that were active in 2020
2. Execute the following in the command bar:

dbt run --select ml\_data\_prep

1. There are more aspects we could consider for this project, such as normalizing the driver confidence by the number of races entered. Including this would help account for a driver’s history and consider whether they are a new or long-time driver. We’re going to keep it simple for now, but these are some of the ways we can expand and improve our machine learning dbt projects. Breaking down our machine learning prep model:
   * Lambda functions — We use some lambda functions to transform our data without having to create a fully-fledged function using the def notation. So what exactly are lambda functions?
     + In Python, a lambda function is a small, anonymous function defined using the keyword "lambda". Lambda functions are used to perform a quick operation, such as a mathematical calculation or a transformation on a list of elements. They are often used in conjunction with higher-order functions, such as apply, map, filter, and reduce.
   * .apply() method — We used .apply() to pass our functions into our lambda expressions to the columns and perform this multiple times in our code. Let’s explain apply a little more:
     + The .apply() function in the pandas library is used to apply a function to a specified axis of a DataFrame or a Series. In our case the function we used was our lambda function!
     + The .apply() function takes two arguments: the first is the function to be applied, and the second is the axis along which the function should be applied. The axis can be specified as 0 for rows or 1 for columns. We are using the default value of 0 so we aren’t explicitly writing it in the code. This means that the function will be applied to each *row* of the DataFrame or Series.
2. Let’s look at the preview of our clean dataframe after running our ml\_data\_prep model:

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)What our clean dataframe fit for machine learning looks like

**Covariate encoding**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#covariate-encoding)

In this next part, we’ll be performing covariate encoding. Breaking down this phrase a bit, a *covariate* is a variable that is relevant to the outcome of a study or experiment, and *encoding* refers to the process of converting data (such as text or categorical variables) into a numerical format that can be used as input for a model. This is necessary because most machine learning algorithms can only work with numerical data. Algorithms don’t speak languages, have eyes to see images, etc. so we encode our data into numbers so algorithms can perform tasks by using calculations they otherwise couldn’t.

🧠 We’ll think about this as : “algorithms like numbers”.

1. Create a new file under ml/prep called covariate\_encoding copy the code below and save.

import pandas as pd  
import numpy as np  
from sklearn.preprocessing import StandardScaler,LabelEncoder,OneHotEncoder  
from sklearn.linear\_model import LogisticRegression  
  
def model(dbt, session):  
 *# dbt configuration*  
 dbt.config(packages=["pandas","numpy","scikit-learn"])  
  
 *# get upstream data*  
 data = dbt.ref("ml\_data\_prep").to\_pandas()  
  
 *# list out covariates we want to use in addition to outcome variable we are modeling - position*  
 covariates = data[['RACE\_YEAR','CIRCUIT\_NAME','GRID','CONSTRUCTOR\_NAME','DRIVER','DRIVERS\_AGE\_YEARS','DRIVER\_CONFIDENCE','CONSTRUCTOR\_RELAIBLITY','TOTAL\_PIT\_STOPS\_PER\_RACE','ACTIVE\_DRIVER','ACTIVE\_CONSTRUCTOR', 'POSITION']]  
  
 *# filter covariates on active drivers and constructors*  
 *# use fil\_cov as short for "filtered\_covariates"*  
 fil\_cov = covariates[(covariates['ACTIVE\_DRIVER']==1)&(covariates['ACTIVE\_CONSTRUCTOR']==1)]  
  
 *# Encode categorical variables using LabelEncoder*  
 *# TODO: we'll update this to both ohe in the future for non-ordinal variables!*   
 le = LabelEncoder()  
 fil\_cov['CIRCUIT\_NAME'] = le.fit\_transform(fil\_cov['CIRCUIT\_NAME'])  
 fil\_cov['CONSTRUCTOR\_NAME'] = le.fit\_transform(fil\_cov['CONSTRUCTOR\_NAME'])  
 fil\_cov['DRIVER'] = le.fit\_transform(fil\_cov['DRIVER'])  
 fil\_cov['TOTAL\_PIT\_STOPS\_PER\_RACE'] = le.fit\_transform(fil\_cov['TOTAL\_PIT\_STOPS\_PER\_RACE'])  
  
 *# Simply target variable "position" to represent 3 meaningful categories in Formula1*  
 *# 1. Podium position 2. Points for team 3. Nothing - no podium or points!*  
 def position\_index(x):  
 if x<4:  
 return 1  
 if x>10:  
 return 3  
 else :  
 return 2  
  
 *# we are dropping the columns that we filtered on in addition to our training variable*  
 encoded\_data = fil\_cov.drop(['ACTIVE\_DRIVER','ACTIVE\_CONSTRUCTOR'],axis=1))  
 encoded\_data['POSITION\_LABEL']= encoded\_data['POSITION'].apply(lambda x: position\_index(x))  
 encoded\_data\_grouped\_target = encoded\_data.drop(['POSITION'],axis=1))  
  
 return encoded\_data\_grouped\_target

1. Execute the following in the command bar:

dbt run --select covariate\_encoding

1. In this code, we are using a ton of functions from libraries! This is really cool, because we can utilize code other people have developed and bring it into our project simply by using the import function. [Scikit-learn](https://scikit-learn.org/stable/), “sklearn” for short, is an extremely popular data science library. Sklearn contains a wide range of machine learning techniques, including supervised and unsupervised learning algorithms, feature scaling and imputation, as well as tools model evaluation and selection. We’ll be using Sklearn for both preparing our covariates and creating models (our next section).
2. Our dataset is pretty small data so we are good to use pandas and sklearn. If you have larger data for your own project in mind, consider dask or category\_encoders.
3. Breaking it down a bit more:
   * We’re selecting a subset of variables that will be used as predictors for a driver’s position.
   * Filter the dataset to only include rows using the active driver and constructor flags we created in the last step.
   * The next step is to use the LabelEncoder from scikit-learn to convert the categorical variables CIRCUIT\_NAME, CONSTRUCTOR\_NAME, DRIVER, and TOTAL\_PIT\_STOPS\_PER\_RACE into numerical values.
   * Create a new variable called POSITION\_LABEL, which is a derived from our position variable.
     + 💭 Why are we changing our position variable? There are 20 total positions in Formula 1 and we are grouping them together to simplify the classification and improve performance. We also want to demonstrate you can create a new function within your dbt model!
     + Our new position\_label variable has meaning:
       - In Formula1 if you are in:
         * Top 3 you get a “podium” position
         * Top 10 you gain points that add to your overall season total
         * Below top 10 you get no points!
     + We are mapping our original variable position to position\_label to the corresponding places above to 1,2, and 3 respectively.
   * Drop the active driver and constructor flags since they were filter criteria and additionally drop our original position variable.

**Splitting into training and testing datasets**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#splitting-into-training-and-testing-datasets)

Now that we’ve cleaned and encoded our data, we are going to further split in by time. In this step, we will create dataframes to use for training and prediction. We’ll be creating two dataframes 1) using data from 2010-2019 for training, and 2) data from 2020 for new prediction inferences. We’ll create variables called start\_year and end\_year so we aren’t filtering on hardcasted values (and can more easily swap them out in the future if we want to retrain our model on different timeframes).

1. Create a file called train\_test\_dataset.py copy and save the following code:

import pandas as pd  
  
def model(dbt, session):  
  
 *# dbt configuration*  
 dbt.config(packages=["pandas"], tags="train")  
  
 *# get upstream data*  
 encoding = dbt.ref("covariate\_encoding").to\_pandas()  
  
 *# provide years so we do not hardcode dates in filter command*  
 start\_year=2010  
 end\_year=2019  
  
 *# describe the data for a full decade*  
 train\_test\_dataset = encoding.loc[encoding['RACE\_YEAR'].between(start\_year, end\_year)]  
  
 return train\_test\_dataset

1. Create a file called hold\_out\_dataset\_for\_prediction.py copy and save the following code below. Now we’ll have a dataset with only the year 2020 that we’ll keep as a hold out set that we are going to use similar to a deployment use case.

import pandas as pd  
  
def model(dbt, session):  
 *# dbt configuration*  
 dbt.config(packages=["pandas"], tags="predict")  
  
 *# get upstream data*  
 encoding = dbt.ref("covariate\_encoding").to\_pandas()  
   
 *# variable for year instead of hardcoding it*   
 year=2020  
  
 *# filter the data based on the specified year*  
 hold\_out\_dataset = encoding.loc[encoding['RACE\_YEAR'] == year]  
   
 return hold\_out\_dataset

1. Execute the following in the command bar:

dbt run --select train\_test\_dataset hold\_out\_dataset\_for\_prediction

To run our temporal data split models, we can use this syntax in the command line to run them both at once. Make sure you use a *space* [syntax](https://docs.getdbt.com/reference/node-selection/syntax) between the model names to indicate you want to run both!

1. **Commit and push** our changes to keep saving our work as we go using ml data prep and splits before moving on.

👏 Now that we’ve finished our machine learning prep work we can move onto the fun part — training and prediction!

**Training a model to predict in machine learning**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#training-a-model-to-predict-in-machine-learning)

We’re ready to start training a model to predict the driver’s position. Now is a good time to pause and take a step back and say, usually in ML projects you’ll try multiple algorithms during development and use an evaluation method such as cross validation to determine which algorithm to use. You can definitely do this in your dbt project, but for the content of this lab we’ll have decided on using a logistic regression to predict position (we actually tried some other algorithms using cross validation outside of this lab such as k-nearest neighbors and a support vector classifier but that didn’t perform as well as the logistic regression and a decision tree that overfit).

There are 3 areas to break down as we go since we are working at the intersection all within one model file:

1. Machine Learning
2. Snowflake and Snowpark
3. dbt Python models

If you haven’t seen code like this before or use joblib files to save machine learning models, we’ll be going over them at a high level and you can explore the links for more technical in-depth along the way! Because Snowflake and dbt have abstracted away a lot of the nitty gritty about serialization and storing our model object to be called again, we won’t go into too much detail here. There’s *a lot* going on here so take it at your pace!

**Training and saving a machine learning model**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#training-and-saving-a-machine-learning-model)

1. Project organization remains key, so let’s make a new subfolder called train\_predict under the ml folder.
2. Now create a new file called train\_test\_position.py and copy and save the following code:

import snowflake.snowpark.functions as F  
from sklearn.model\_selection import train\_test\_split  
import pandas as pd  
from sklearn.metrics import confusion\_matrix, balanced\_accuracy\_score  
import io  
from sklearn.linear\_model import LogisticRegression  
from joblib import dump, load  
import joblib  
import logging  
import sys  
from joblib import dump, load  
  
logger = logging.getLogger("mylog")  
  
def save\_file(session, model, path, dest\_filename):  
 input\_stream = io.BytesIO()  
 joblib.dump(model, input\_stream)  
 session.\_conn.upload\_stream(input\_stream, path, dest\_filename)  
 return "successfully created file: " + path  
  
def model(dbt, session):  
 dbt.config(  
 packages = ['numpy','scikit-learn','pandas','numpy','joblib','cachetools'],  
 materialized = "table",  
 tags = "train"  
 )  
 *# Create a stage in Snowflake to save our model file*  
 session.sql('create or replace stage MODELSTAGE').collect()  
  
 *#session.\_use\_scoped\_temp\_objects = False*  
 version = "1.0"  
 logger.info('Model training version: ' + version)  
  
 *# read in our training and testing upstream dataset*  
 test\_train\_df = dbt.ref("train\_test\_dataset")  
  
 *# cast snowpark df to pandas df*  
 test\_train\_pd\_df = test\_train\_df.to\_pandas()  
 target\_col = "POSITION\_LABEL"  
  
 *# split out covariate predictors, x, from our target column position\_label, y.*  
 split\_X = test\_train\_pd\_df.drop([target\_col], axis=1)  
 split\_y = test\_train\_pd\_df[target\_col]  
  
 *# Split out our training and test data into proportions*  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(split\_X, split\_y, train\_size=0.7, random\_state=42)  
 train = [X\_train, y\_train]  
 test = [X\_test, y\_test]  
 *# now we are only training our one model to deploy*  
 *# we are keeping the focus on the workflows and not algorithms for this lab!*  
 model = LogisticRegression()  
  
 *# fit the preprocessing pipeline and the model together*   
 model.fit(X\_train, y\_train)   
 y\_pred = model.predict\_proba(X\_test)[:,1]  
 predictions = [round(value) for value in y\_pred]  
 balanced\_accuracy = balanced\_accuracy\_score(y\_test, predictions)  
  
 *# Save the model to a stage*  
 save\_file(session, model, "@MODELSTAGE/driver\_position\_"+version, "driver\_position\_"+version+".joblib" )  
 logger.info('Model artifact:' + "@MODELSTAGE/driver\_position\_"+version+".joblib")  
  
 *# Take our pandas training and testing dataframes and put them back into snowpark dataframes*  
 snowpark\_train\_df = session.write\_pandas(pd.concat(train, axis=1, join='inner'), "train\_table", auto\_create\_table=True, create\_temp\_table=True)  
 snowpark\_test\_df = session.write\_pandas(pd.concat(test, axis=1, join='inner'), "test\_table", auto\_create\_table=True, create\_temp\_table=True)  
  
 *# Union our training and testing data together and add a column indicating train vs test rows*  
 return snowpark\_train\_df.with\_column("DATASET\_TYPE", F.lit("train")).union(snowpark\_test\_df.with\_column("DATASET\_TYPE", F.lit("test")))

1. Execute the following in the command bar:

dbt run --select train\_test\_position

1. Breaking down our Python script here:
   * We’re importing some helpful libraries.
     + Defining a function called save\_file() that takes four parameters: session, model, path and dest\_filename that will save our logistic regression model file.
       - session — an object representing a connection to Snowflake.
       - model — an object that needs to be saved. In this case, it's a Python object that is a scikit-learn that can be serialized with joblib.
       - path — a string representing the directory or bucket location where the file should be saved.
       - dest\_filename — a string representing the desired name of the file.
     + Creating our dbt model
       - Within this model we are creating a stage called MODELSTAGE to place our logistic regression joblib model file. This is really important since we need a place to keep our model to reuse and want to ensure it's there. When using Snowpark commands, it's common to see the .collect() method to ensure the action is performed. Think of the session as our “start” and collect as our “end” when [working with Snowpark](https://docs.snowflake.com/en/developer-guide/snowpark/python/working-with-dataframes.html) (you can use other ending methods other than collect).
       - Using .ref() to connect into our train\_test\_dataset model.
       - Now we see the machine learning part of our analysis:
         * Create new dataframes for our prediction features from our target variable position\_label.
         * Split our dataset into 70% training (and 30% testing), train\_size=0.7 with a random\_state specified to have repeatable results.
         * Specify our model is a logistic regression.
         * Fit our model. In a logistic regression this means finding the coefficients that will give the least classification error.
         * Round our predictions to the nearest integer since logistic regression creates a probability between for each class and calculate a balanced accuracy to account for imbalances in the target variable.
     + Right now our model is only in memory, so we need to use our nifty function save\_file to save our model file to our Snowflake stage. We save our model as a joblib file so Snowpark can easily call this model object back to create predictions. We really don’t need to know much else as a data practitioner unless we want to. It’s worth noting that joblib files aren’t able to be queried directly by SQL. To do this, we would need to transform the joblib file to an SQL querable format such as JSON or CSV (out of scope for this workshop).
     + Finally we want to return our dataframe, but create a new column indicating what rows were used for training and those for training.
2. Viewing our output of this model:

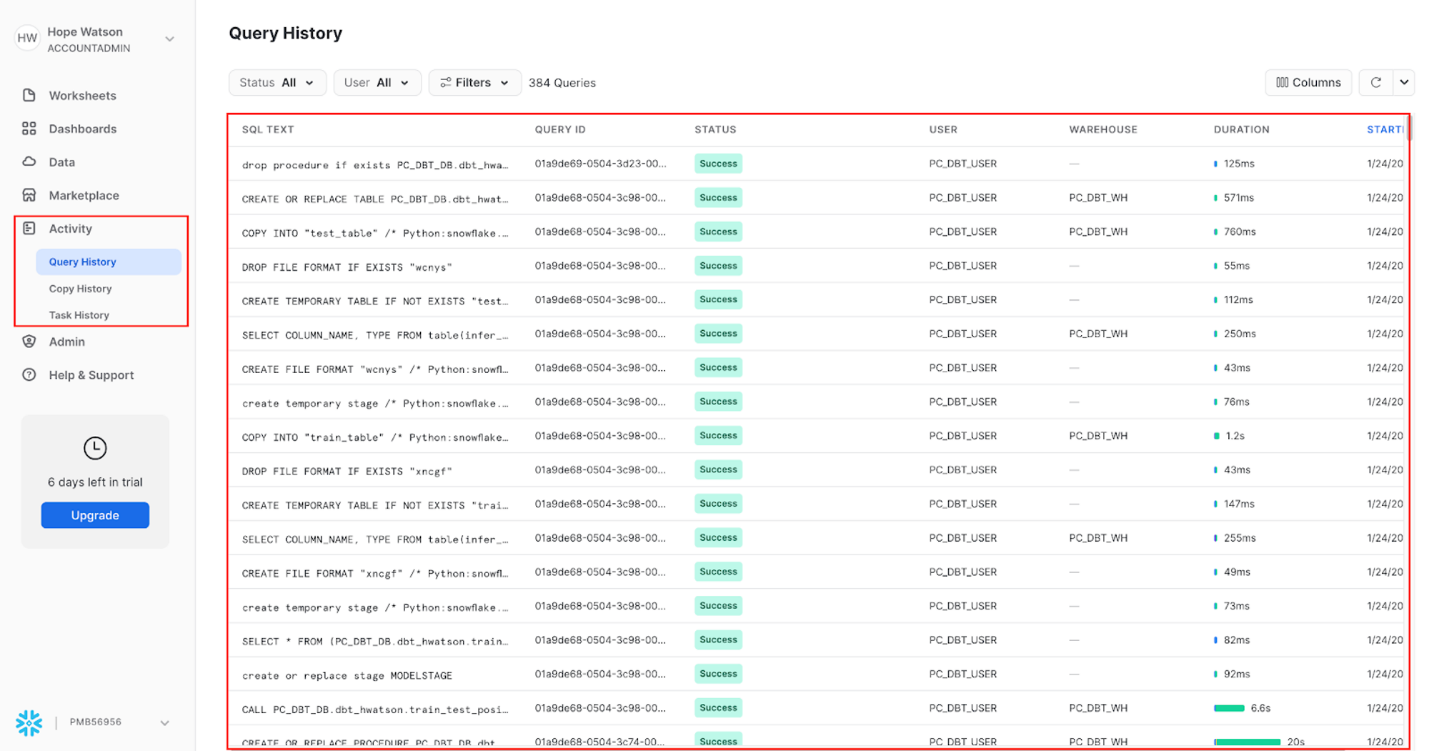
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Preview which rows of our model were used for training and testing

1. Let’s pop back over to Snowflake and check that our logistic regression model has been stored in our MODELSTAGE using the command:

list @modelstage

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)List the objects in our Snowflake stage to check for our logistic regression to predict driver position

1. To investigate the commands run as part of train\_test\_position script, navigate to Snowflake query history to view it **Activity > Query History**. We can view the portions of query that we wrote such as create or replace stage MODELSTAGE, but we also see additional queries that Snowflake uses to interpret python code.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)View Snowflake query history to see how python models are run under the hood

**Predicting on new data**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#predicting-on-new-data)

1. Create a new file called predict\_position.py and copy and save the following code:

import logging  
import joblib  
import pandas as pd  
import os  
from snowflake.snowpark import types as T  
  
DB\_STAGE = 'MODELSTAGE'  
version = '1.0'  
*# The name of the model file*  
model\_file\_path = 'driver\_position\_'+version  
model\_file\_packaged = 'driver\_position\_'+version+'.joblib'  
  
*# This is a local directory, used for storing the various artifacts locally*  
LOCAL\_TEMP\_DIR = f'/tmp/driver\_position'  
DOWNLOAD\_DIR = os.path.join(LOCAL\_TEMP\_DIR, 'download')  
TARGET\_MODEL\_DIR\_PATH = os.path.join(LOCAL\_TEMP\_DIR, 'ml\_model')  
TARGET\_LIB\_PATH = os.path.join(LOCAL\_TEMP\_DIR, 'lib')  
  
*# The feature columns that were used during model training*  
*# and that will be used during prediction*  
FEATURE\_COLS = [  
 "RACE\_YEAR"  
 ,"CIRCUIT\_NAME"  
 ,"GRID"  
 ,"CONSTRUCTOR\_NAME"  
 ,"DRIVER"  
 ,"DRIVERS\_AGE\_YEARS"  
 ,"DRIVER\_CONFIDENCE"  
 ,"CONSTRUCTOR\_RELAIBLITY"  
 ,"TOTAL\_PIT\_STOPS\_PER\_RACE"]  
  
def register\_udf\_for\_prediction(p\_predictor ,p\_session ,p\_dbt):  
  
 *# The prediction udf*  
  
 def predict\_position(p\_df: T.PandasDataFrame[int, int, int, int,  
 int, int, int, int, int]) -> T.PandasSeries[int]:  
 *# Snowpark currently does not set the column name in the input dataframe*  
 *# The default col names are like 0,1,2,... Hence we need to reset the column*  
 *# names to the features that we initially used for training.*  
 p\_df.columns = [\*FEATURE\_COLS]  
   
 *# Perform prediction. this returns an array object*  
 pred\_array = p\_predictor.predict(p\_df)  
 *# Convert to series*  
 df\_predicted = pd.Series(pred\_array)  
 return df\_predicted  
  
 *# The list of packages that will be used by UDF*  
 udf\_packages = p\_dbt.config.get('packages')  
  
 predict\_position\_udf = p\_session.udf.register(  
 predict\_position  
 ,name=f'predict\_position'  
 ,packages = udf\_packages  
 )  
 return predict\_position\_udf  
  
def download\_models\_and\_libs\_from\_stage(p\_session):  
 p\_session.file.get(f'@{DB\_STAGE}/{model\_file\_path}/{model\_file\_packaged}', DOWNLOAD\_DIR)  
  
def load\_model(p\_session):  
 *# Load the model and initialize the predictor*  
 model\_fl\_path = os.path.join(DOWNLOAD\_DIR, model\_file\_packaged)  
 predictor = joblib.load(model\_fl\_path)  
 return predictor  
  
*# -------------------------------*  
def model(dbt, session):  
 dbt.config(  
 packages = ['snowflake-snowpark-python' ,'scipy','scikit-learn' ,'pandas' ,'numpy'],  
 materialized = "table",  
 tags = "predict"  
 )  
 session.\_use\_scoped\_temp\_objects = False  
 download\_models\_and\_libs\_from\_stage(session)  
 predictor = load\_model(session)  
 predict\_position\_udf = register\_udf\_for\_prediction(predictor, session ,dbt)  
  
 *# Retrieve the data, and perform the prediction*  
 hold\_out\_df = (dbt.ref("hold\_out\_dataset\_for\_prediction")  
 .select(\*FEATURE\_COLS)  
 )  
  
 *# Perform prediction.*  
 new\_predictions\_df = hold\_out\_df.withColumn("position\_predicted"  
 ,predict\_position\_udf(\*FEATURE\_COLS)  
 )  
  
 return new\_predictions\_df

1. Execute the following in the command bar:

dbt run --select predict\_position

1. **Commit and push** our changes to keep saving our work as we go using the commit message logistic regression model training and application before moving on.
2. At a high level in this script, we are:
   * Retrieving our staged logistic regression model
   * Loading the model in
   * Placing the model within a user defined function (UDF) to call in line predictions on our driver’s position
3. At a more detailed level:
   * Import our libraries.
   * Create variables to reference back to the MODELSTAGE we just created and stored our model to.
   * The temporary file paths we created might look intimidating, but all we’re doing here is programmatically using an initial file path and adding to it to create the following directories:
     + LOCAL\_TEMP\_DIR ➡️ /tmp/driver\_position
     + DOWNLOAD\_DIR ➡️ /tmp/driver\_position/download
     + TARGET\_MODEL\_DIR\_PATH ➡️ /tmp/driver\_position/ml\_model
     + TARGET\_LIB\_PATH ➡️ /tmp/driver\_position/lib
   * Provide a list of our feature columns that we used for model training and will now be used on new data for prediction.
   * Next, we are creating our main function register\_udf\_for\_prediction(p\_predictor ,p\_session ,p\_dbt):. This function is used to register a user-defined function (UDF) that performs the machine learning prediction. It takes three parameters: p\_predictor is an instance of the machine learning model, p\_session is an instance of the Snowflake session, and p\_dbt is an instance of the dbt library. The function creates a UDF named predict\_churn which takes a pandas dataframe with the input features and returns a pandas series with the predictions.
   * ⚠️ Pay close attention to the whitespace here. We are using a function within a function for this script.
   * We have 2 simple functions that are programmatically retrieving our file paths to first get our stored model out of our MODELSTAGE and downloaded into the session download\_models\_and\_libs\_from\_stage and then to load the contents of our model in (parameters) in load\_model to use for prediction.
   * Take the model we loaded in and call it predictor and wrap it in a UDF.
   * Return our dataframe with both the features used to predict and the new label.

🧠 Another way to read this script is from the bottom up. This can help us progressively see what is going into our final dbt model and work backwards to see how the other functions are being referenced.

1. Let’s take a look at our predicted position alongside our feature variables. Open a new scratchpad and use the following query. I chose to order by the prediction of who would obtain a podium position:

select \* from {{ ref('predict\_position') }} order by position\_predicted

1. We can see that we created predictions in our final dataset, we are ready to move on to testing!

**Test your data models**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#test-your-data-models)

We have now completed building all the models for today’s lab, but how do we know if they meet our assertions? Put another way, how do we know the quality of our data models are any good? This brings us to testing!

We test data models for mainly two reasons:

* Ensure that our source data is clean on ingestion before we start data modeling/transformation (aka avoid garbage in, garbage out problem).
* Make sure we don’t introduce bugs in the transformation code we wrote (stop ourselves from creating bad joins/fanouts).

Testing in dbt comes in two flavors: [generic](https://docs.getdbt.com/docs/build/data-tests#generic-data-tests) and [singular](https://docs.getdbt.com/docs/build/data-tests#singular-data-tests).

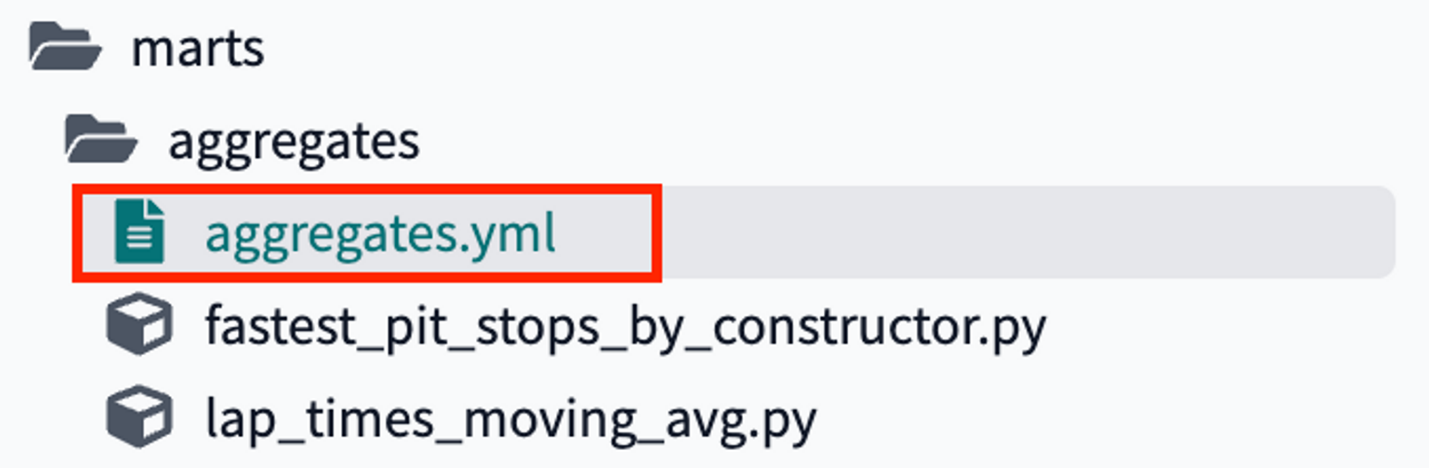
You define them in a test block (similar to a macro) and once defined, you can reference them by name in your .yml files (applying them to models, columns, sources, snapshots, and seeds).

You might be wondering: *what about testing Python models?*

Since the output of our Python models are tables, we can test SQL and Python models the same way! We don’t have to worry about any syntax differences when testing SQL versus Python data models. This means we use .yml and .sql files to test our entities (tables, views, etc.). Under the hood, dbt is running an SQL query on our tables to see if they meet assertions. If no rows are returned, dbt will surface a passed test. Conversely, if a test results in returned rows, it will fail or warn depending on the configuration (more on that later).

**Generic tests**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#generic-tests)

1. To implement generic out-of-the-box tests dbt comes with, we can use YAML files to specify information about our models. To add generic tests to our aggregates model, create a file called aggregates.yml, copy the code block below into the file, and save.

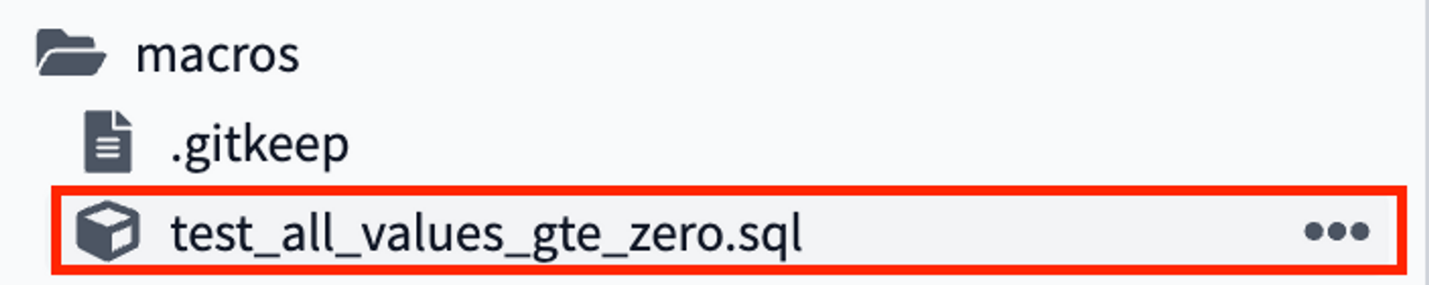
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)The aggregates.yml file in our file tree

version: 2  
  
models:  
 - name: fastest\_pit\_stops\_by\_constructor  
 description: Use the python .describe() method to retrieve summary statistics table about pit stops by constructor. Sort by average stop time ascending so the first row returns the fastest constructor.  
 columns:  
 - name: constructor\_name  
 description: team that makes the car  
 tests:  
 - unique  
  
 - name: lap\_times\_moving\_avg  
 description: Use the python .rolling() method to calculate the 5 year rolling average of pit stop times alongside the average for each year.   
 columns:  
 - name: race\_year  
 description: year of the race  
 tests:  
 - relationships:  
 to: ref('int\_lap\_times\_years')  
 field: race\_year

1. Let’s unpack the code we have here. We have both our aggregates models with the model name to know the object we are referencing and the description of the model that we’ll populate in our documentation. At the column level (a level below our model), we are providing the column name followed by our tests. We want to ensure our constructor\_name is unique since we used a pandas groupby on constructor\_name in the model fastest\_pit\_stops\_by\_constructor. Next, we want to ensure our race\_year has referential integrity from the model we selected from int\_lap\_times\_years into our subsequent lap\_times\_moving\_avg model.
2. Finally, if we want to see how tests were deployed on sources and SQL models, we can look at other files in our project such as the f1\_sources.yml we created in our Sources and staging section.

**Using macros for testing**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#using-macros-for-testing)

1. Under your macros folder, create a new file and name it test\_all\_values\_gte\_zero.sql. Copy the code block below and save the file. For clarity, “gte” is an abbreviation for greater than or equal to.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)macro file for reusable testing code

{% macro test\_all\_values\_gte\_zero(table, column) %}  
  
select \* from {{ ref(table) }} where {{ column }} < 0  
  
{% endmacro %}

1. Macros in Jinja are pieces of code that can be reused multiple times in our SQL models — they are analogous to "functions" in other programming languages, and are extremely useful if you find yourself repeating code across multiple models.
2. We use the {% macro %} to indicate the start of the macro and {% endmacro %} for the end. The text after the beginning of the macro block is the name we are giving the macro to later call it. In this case, our macro is called test\_all\_values\_gte\_zero. Macros take in *arguments* to pass through, in this case the table and the column. In the body of the macro, we see an SQL statement that is using the ref function to dynamically select the table and then the column. You can always view macros without having to run them by using dbt run-operation. You can learn more [here](https://docs.getdbt.com/reference/commands/run-operation).
3. Great, now we want to reference this macro as a test! Let’s create a new test file called macro\_pit\_stops\_mean\_is\_positive.sql in our tests folder.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)creating a test on our pit stops model referencing the macro

1. Copy the following code into the file and save:

{{  
 config(  
 enabled=true,  
 severity='warn',  
 tags = ['bi']  
 )  
}}  
  
{{ test\_all\_values\_gte\_zero('fastest\_pit\_stops\_by\_constructor', 'mean') }}

1. In our testing file, we are applying some configurations to the test including enabled, which is an optional configuration for disabling models, seeds, snapshots, and tests. Our severity is set to warn instead of error, which means our pipeline will still continue to run. We have tagged our test with bi since we are applying this test to one of our bi models.

Then, in our final line, we are calling the test\_all\_values\_gte\_zero macro that takes in our table and column arguments and inputting our table 'fastest\_pit\_stops\_by\_constructor' and the column 'mean'.

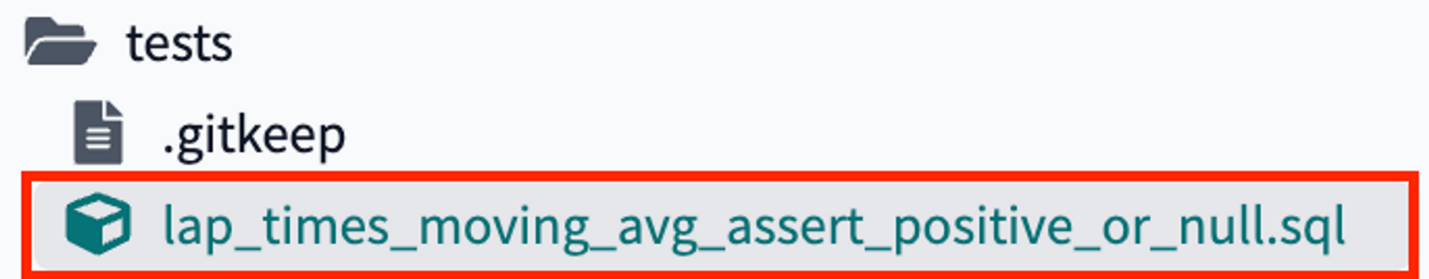
**Custom singular tests to validate Python models**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#custom-singular-tests-to-validate-python-models)

The simplest way to define a test is by writing the exact SQL that will return failing records. We call these "singular" tests, because they're one-off assertions usable for a single purpose.

These tests are defined in .sql files, typically in your tests directory (as defined by your test-paths config). You can use Jinja in SQL models (including ref and source) in the test definition, just like you can when creating models. Each .sql file contains one select statement, and it defines one test.

Let’s add a custom test that asserts that the moving average of the lap time over the last 5 years is greater than zero (it’s impossible to have time less than 0!). It is easy to assume if this is not the case the data has been corrupted.

1. Create a file lap\_times\_moving\_avg\_assert\_positive\_or\_null.sql under the tests folder.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)custom singular test for testing lap times are positive values

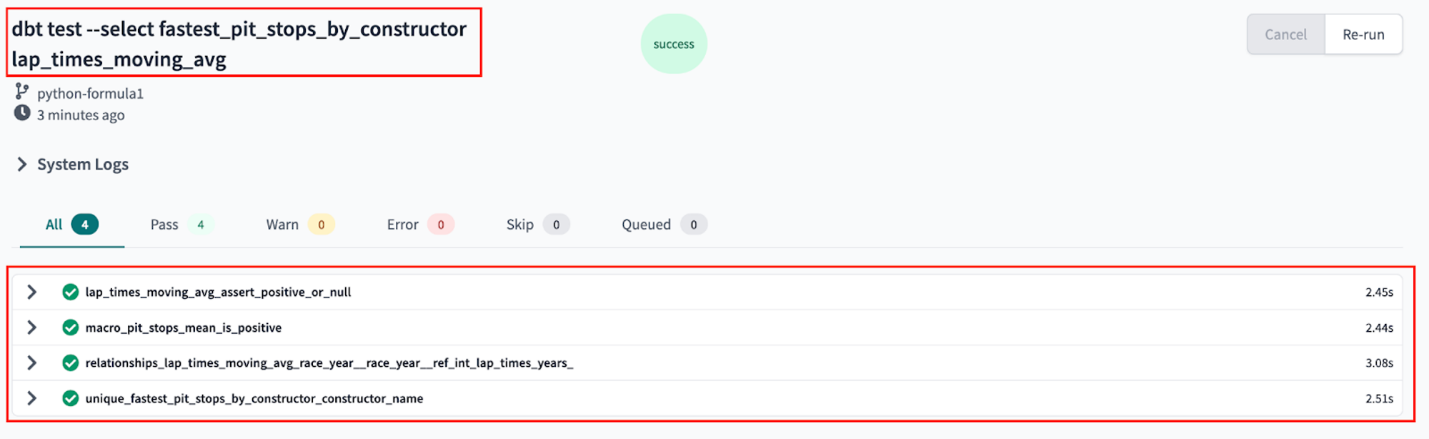
1. Copy the following code and save the file:

{{  
 config(  
 enabled=true,  
 severity='error',  
 tags = ['bi']  
 )  
}}  
  
with lap\_times\_moving\_avg as ( select \* from {{ ref('lap\_times\_moving\_avg') }} )  
  
select \*  
from lap\_times\_moving\_avg   
where lap\_moving\_avg\_5\_years < 0 and lap\_moving\_avg\_5\_years is not null

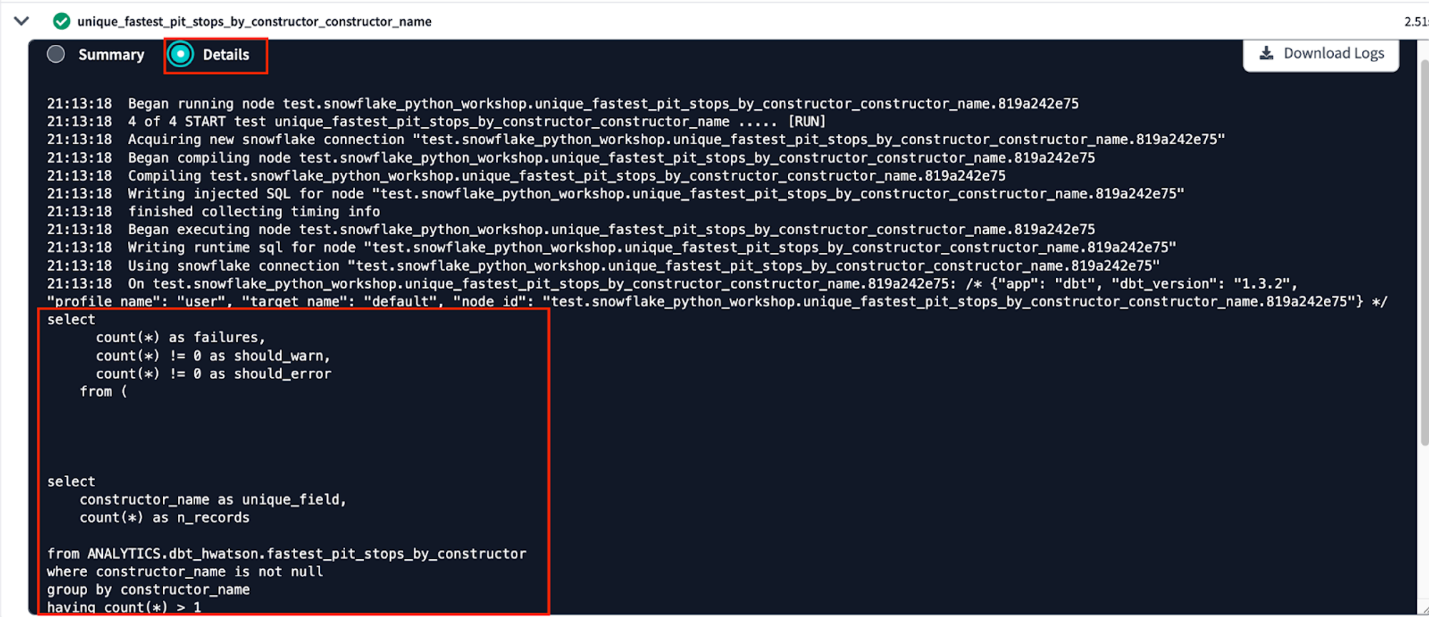
**Putting all our tests together**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#putting-all-our-tests-together)

1. Time to run our tests! Altogether, we have created 4 tests for our 2 Python models:
   * fastest\_pit\_stops\_by\_constructor
     + Unique constructor\_name
     + Lap times are greater than 0 or null (to allow for the first leading values in a rolling calculation)
   * lap\_times\_moving\_avg
     + Referential test on race\_year
     + Mean pit stop times are greater than or equal to 0 (no negative time values)
2. To run the tests on both our models, we can use this syntax in the command line to run them both at once, similar to how we did our data splits earlier. Execute the following in the command bar:

dbt test --select fastest\_pit\_stops\_by\_constructor lap\_times\_moving\_avg

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)running tests on our python models

1. All 4 of our tests passed (yay for clean data)! To understand the SQL being run against each of our tables, we can click into the details of the test.
2. Navigating into the **Details** of the unique\_fastest\_pit\_stops\_by\_constructor\_name, we can see that each line constructor\_name should only have one row.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)view details of testing our python model that used SQL to test data assertions

**Document your dbt project**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#document-your-dbt-project)

When it comes to documentation, dbt brings together both column and model level descriptions that you can provide as well as details from your Snowflake information schema in a static site for consumption by other data team members and stakeholders.

We are going to revisit 2 areas of our project to understand our documentation:

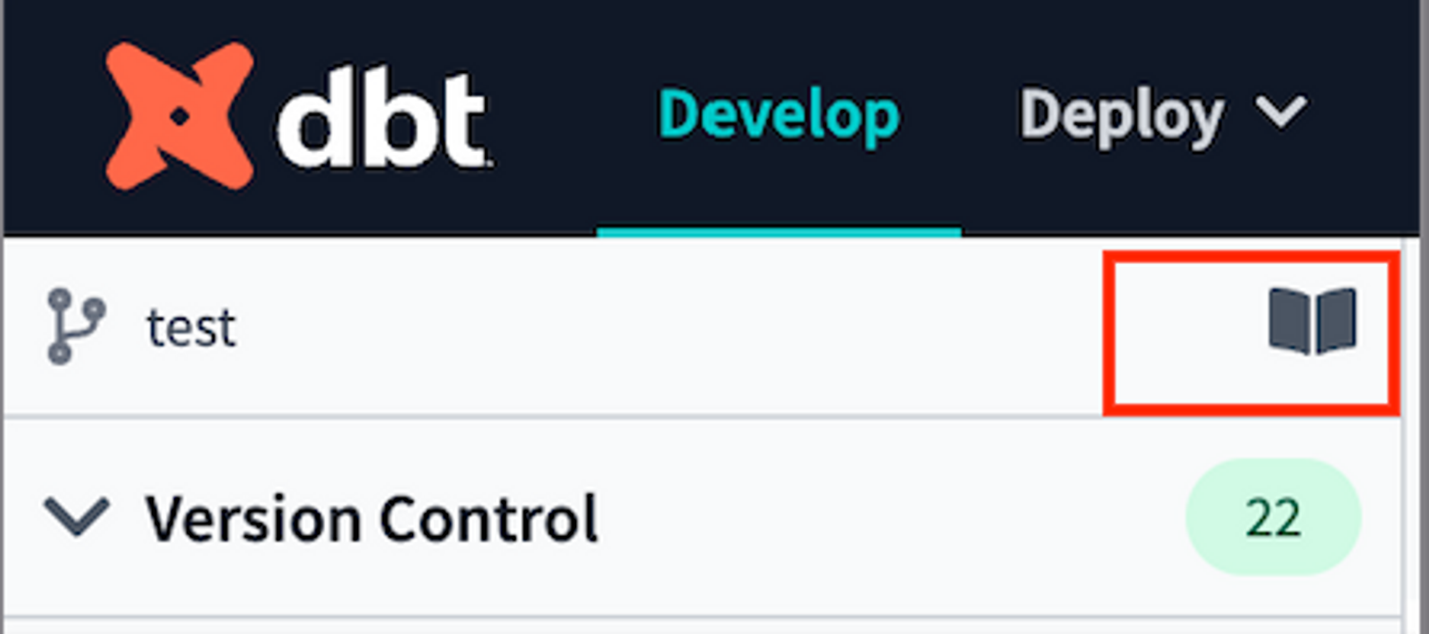
* intermediate.md file
* dbt\_project.yml file

To start, let’s look back at our intermediate.md file. We can see that we provided multi-line descriptions for the models in our intermediate models using [docs blocks](https://docs.getdbt.com/docs/collaborate/documentation#using-docs-blocks). Then we reference these docs blocks in our .yml file. Building descriptions with doc blocks in Markdown files gives you the ability to format your descriptions with Markdown and are particularly helpful when building long descriptions, either at the column or model level. In our dbt\_project.yml, we added node\_colors at folder levels.

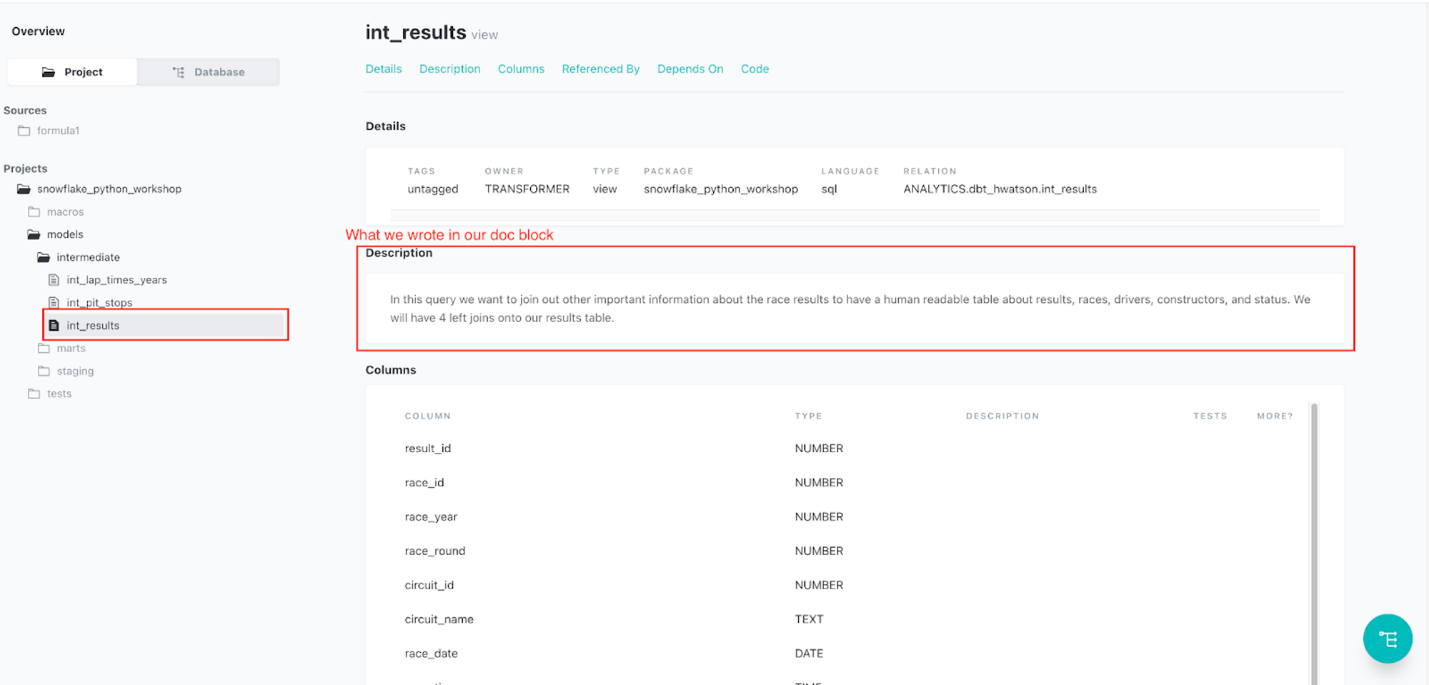
1. To see all these pieces come together, execute this in the command bar:

dbt docs generate

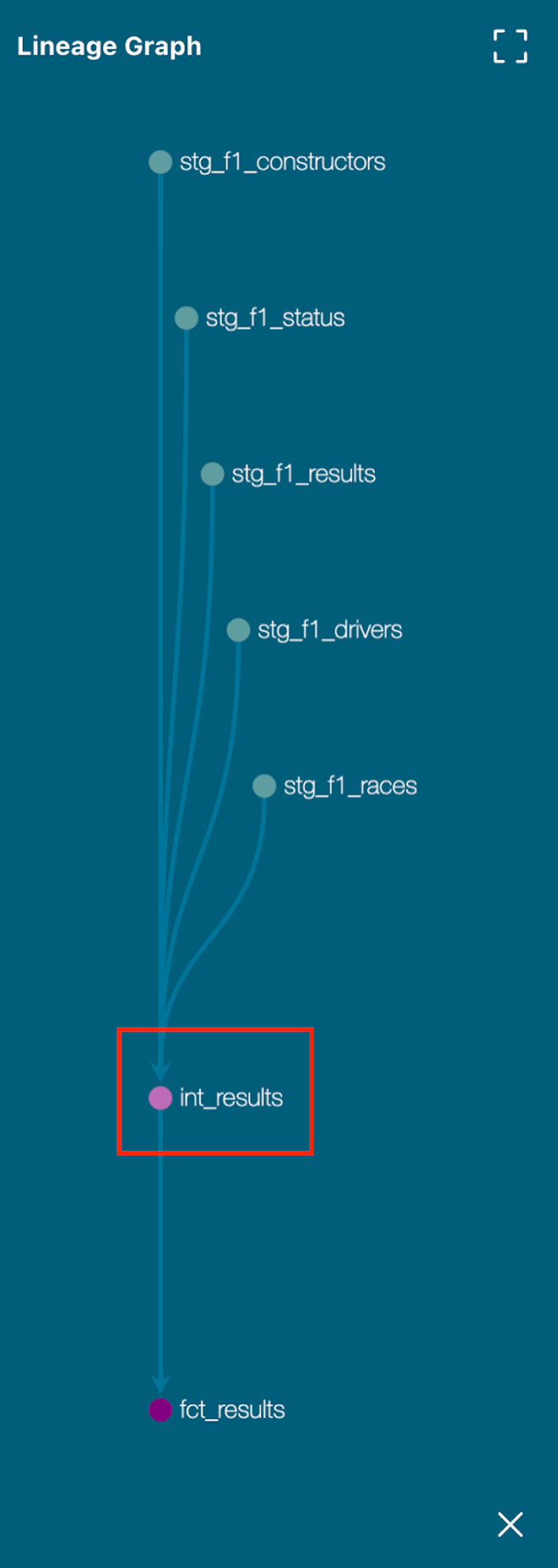
This will generate the documentation for your project. Click the book button, as shown in the screenshot below to access the docs.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)dbt docs book icon

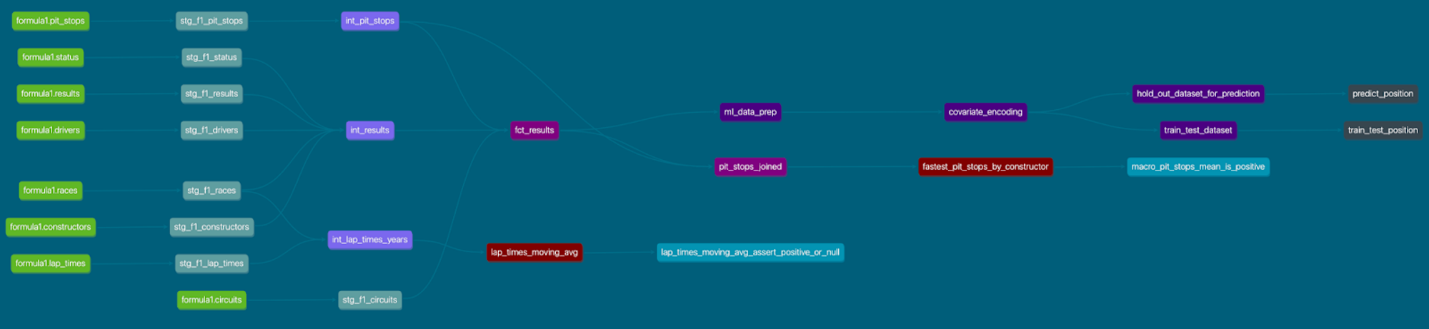
1. Go to our project area and view int\_results. View the description that we created in our doc block.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Docblock description within docs site

1. View the mini-lineage that looks at the model we are currently selected on (int\_results in this case).

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Mini lineage view on docs site

1. In our dbt\_project.yml, we configured node\_colors depending on the file directory. Starting in dbt v1.3, we can see how our lineage in our docs looks. By color coding your project, it can help you cluster together similar models or steps and more easily troubleshoot.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Full project DAG on docs site

**Deploy your code**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#deploy-your-code)

Before we jump into deploying our code, let's have a quick primer on environments. Up to this point, all of the work we've done in the dbt Cloud IDE has been in our development environment, with code committed to a feature branch and the models we've built created in our development schema in Snowflake as defined in our Development environment connection. Doing this work on a feature branch, allows us to separate our code from what other coworkers are building and code that is already deemed production ready. Building models in a development schema in Snowflake allows us to separate the database objects we might still be modifying and testing from the database objects running production dashboards or other downstream dependencies. Together, the combination of a Git branch and Snowflake database objects form our environment.

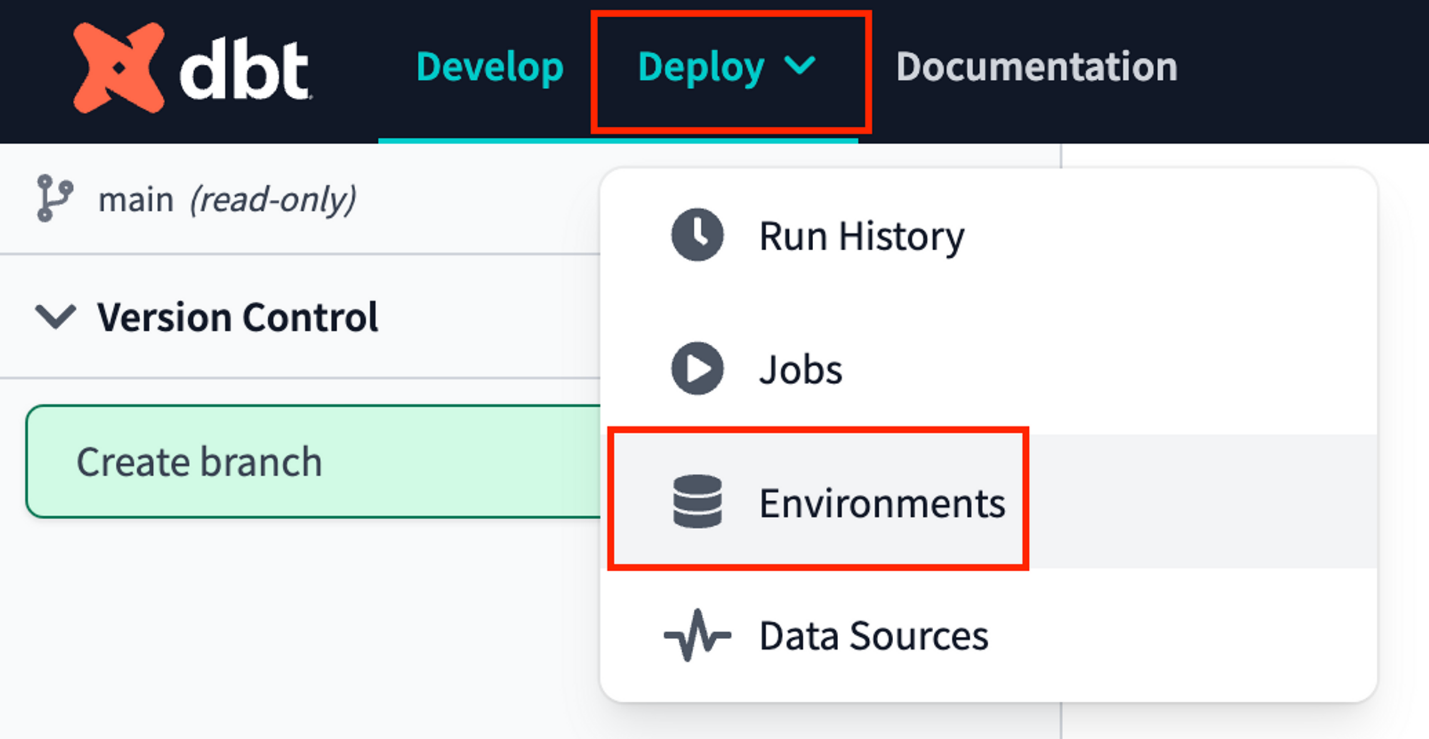
Now that we've completed testing and documenting our work, we're ready to deploy our code from our development environment to our production environment and this involves two steps:

* Promoting code from our feature branch to the production branch in our repository.
  + Generally, the production branch is going to be named your main branch and there's a review process to go through before merging code to the main branch of a repository. Here we are going to merge without review for ease of this workshop.
* Deploying code to our production environment.
  + Once our code is merged to the main branch, we'll need to run dbt in our production environment to build all of our models and run all of our tests. This will allow us to build production-ready objects into our production environment in Snowflake. Luckily for us, the Partner Connect flow has already created our deployment environment and job to facilitate this step.

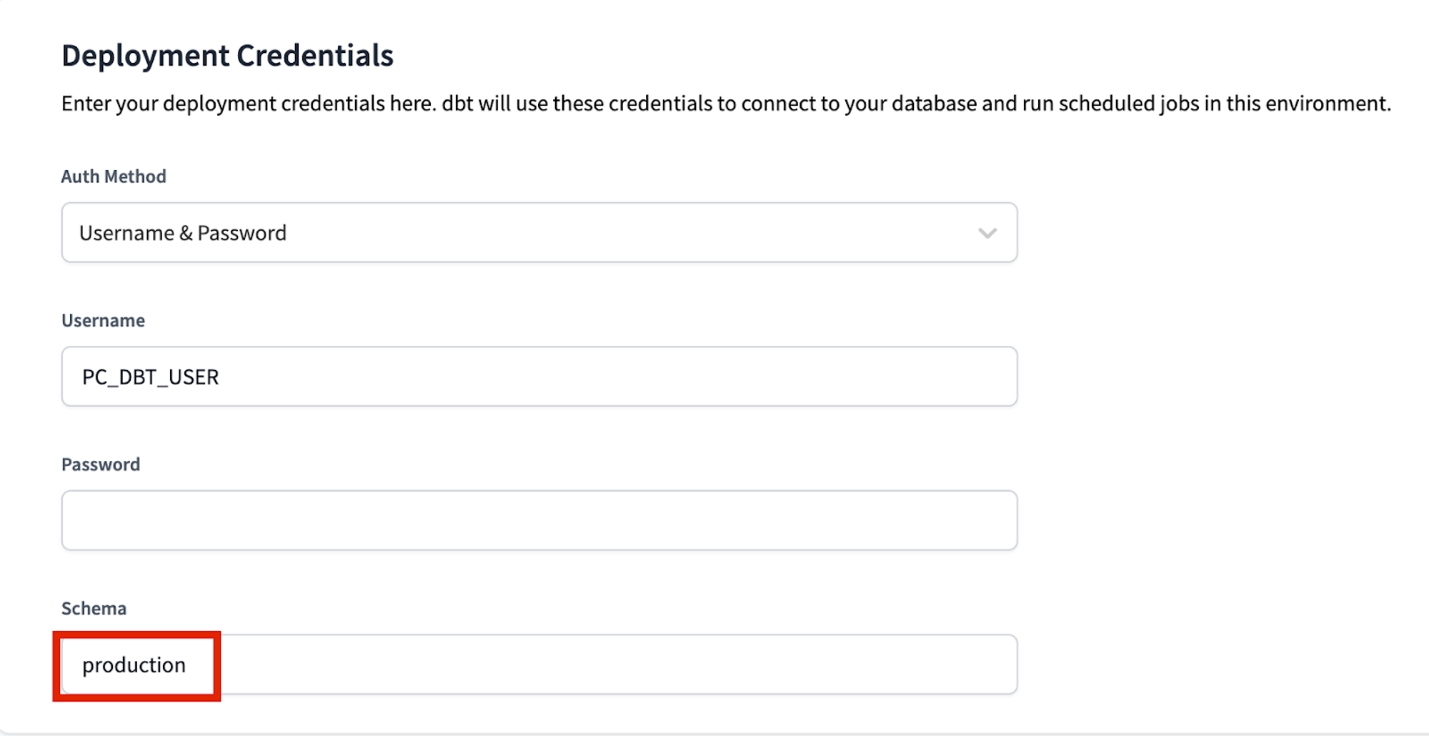
1. Before getting started, let's make sure that we've committed all of our work to our feature branch. If you still have work to commit, you'll be able to select the **Commit and push**, provide a message, and then select **Commit** again.
2. Once all of your work is committed, the git workflow button will now appear as **Merge to main**. Select **Merge to main** and the merge process will automatically run in the background.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Merge into main

1. When it's completed, you should see the git button read **Create branch** and the branch you're currently looking at will become **main**.
2. Now that all of our development work has been merged to the main branch, we can build our deployment job. Given that our production environment and production job were created automatically for us through Partner Connect, all we need to do here is update some default configurations to meet our needs.
3. In the menu, select **Deploy** **> Environments**

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Navigate to environments within the UI

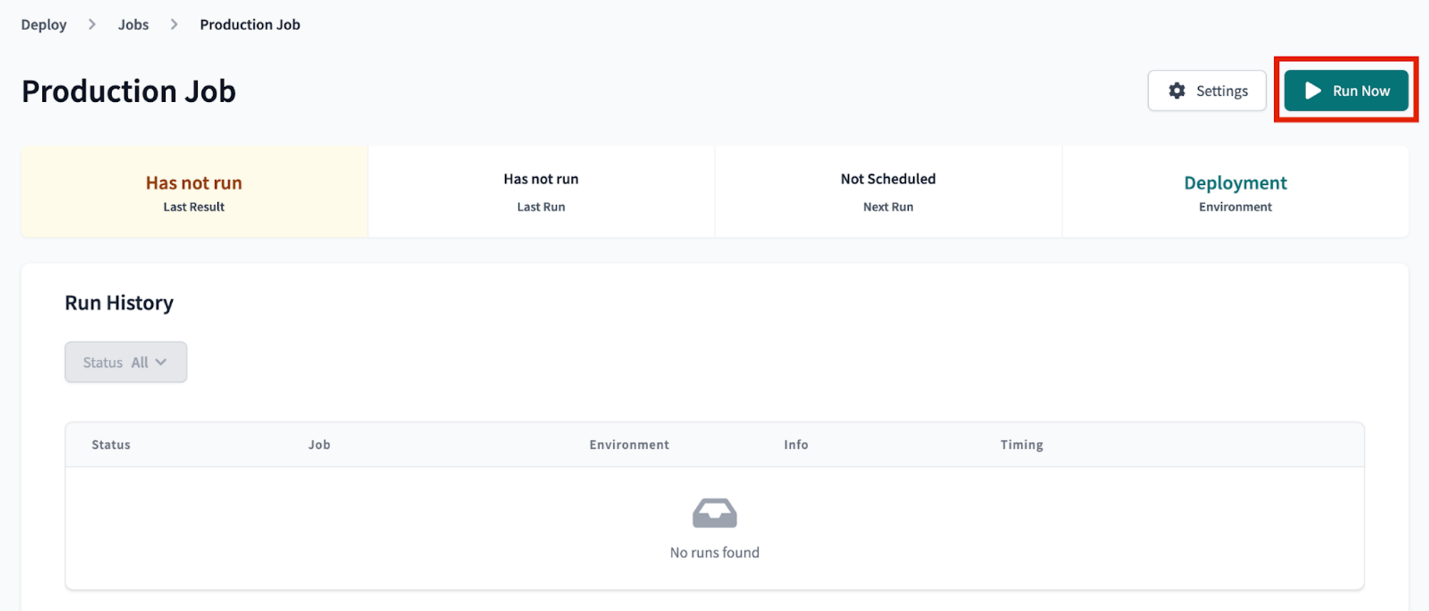
1. You should see two environments listed and you'll want to select the **Deployment** environment then **Settings** to modify it.
2. Before making any changes, let's touch on what is defined within this environment. The Snowflake connection shows the credentials that dbt Cloud is using for this environment and in our case they are the same as what was created for us through Partner Connect. Our deployment job will build in our PC\_DBT\_DB database and use the default Partner Connect role and warehouse to do so. The deployment credentials section also uses the info that was created in our Partner Connect job to create the credential connection. However, it is using the same default schema that we've been using as the schema for our development environment.
3. Let's update the schema to create a new schema specifically for our production environment. Click **Edit** to allow you to modify the existing field values. Navigate to **Deployment Credentials >** **schema.**
4. Update the schema name to **production**. Remember to select **Save** after you've made the change.

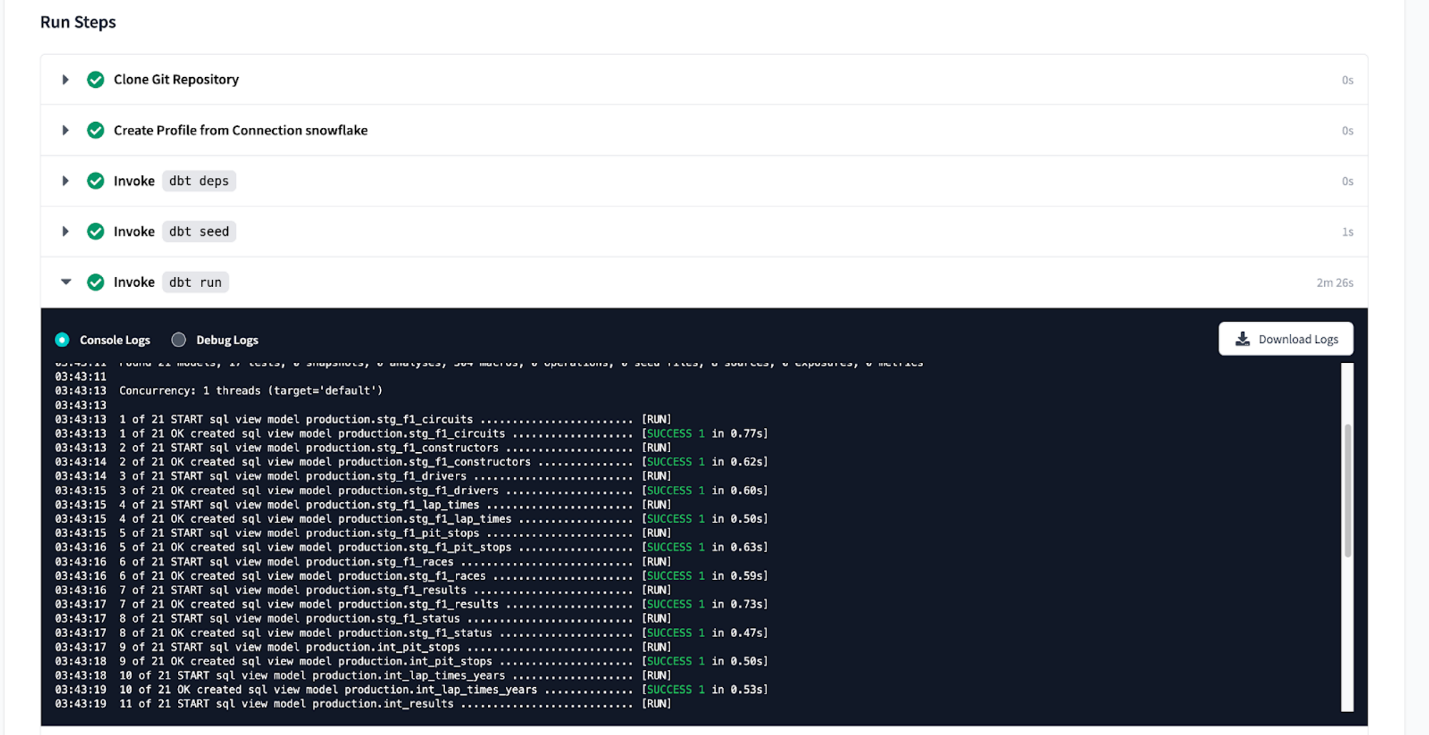
[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Update the deployment credentials schema to production

1. By updating the schema for our production environment to **production**, it ensures that our deployment job for this environment will build our dbt models in the **production** schema within the PC\_DBT\_DB database as defined in the Snowflake Connection section.
2. Now let's switch over to our production job. Click on the deploy tab again and then select **Jobs**. You should see an existing and preconfigured **Partner Connect Trial Job**. Similar to the environment, click on the job, then select **Settings** to modify it. Let's take a look at the job to understand it before making changes.
   * The Environment section is what connects this job with the environment we want it to run in. This job is already defaulted to use the Deployment environment that we just updated and the rest of the settings we can keep as is.
   * The Execution settings section gives us the option to generate docs, run source freshness, and defer to a previous run state. For the purposes of our lab, we're going to keep these settings as is as well and stick with just generating docs.
   * The Commands section is where we specify exactly which commands we want to run during this job, and we also want to keep this as is. We want our seed to be uploaded first, then run our models, and finally test them. The order of this is important as well, considering that we need our seed to be created before we can run our incremental model, and we need our models to be created before we can test them.
   * Finally, we have the Triggers section, where we have a number of different options for scheduling our job. Given that our data isn't updating regularly here and we're running this job manually for now, we're also going to leave this section alone.

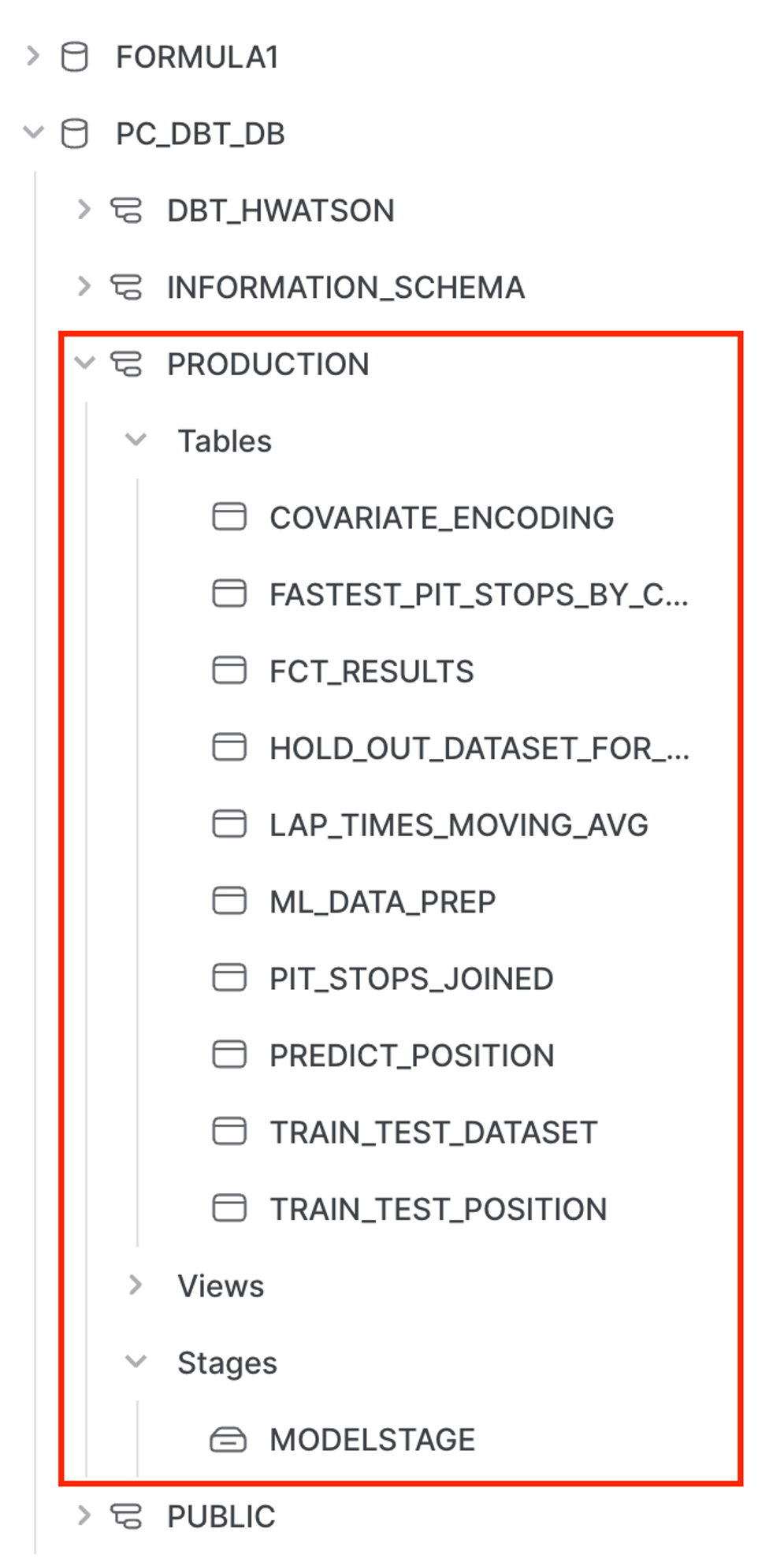
So, what are we changing then? Just the name! Click **Edit** to allow you to make changes. Then update the name of the job to **Production Job** to denote this as our production deployment job. After that's done, click **Save**.

1. Now let's go to run our job. Clicking on the job name in the path at the top of the screen will take you back to the job run history page where you'll be able to click **Run run** to kick off the job. If you encounter any job failures, try running the job again before further troubleshooting.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Run production job

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)View production job details

1. Let's go over to Snowflake to confirm that everything built as expected in our production schema. Refresh the database objects in your Snowflake account and you should see the production schema now within our default Partner Connect database. If you click into the schema and everything ran successfully, you should be able to see all of the models we developed.

[](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15)Check all our models in our pipeline are in Snowflake

**Conclusion**[**​**](https://docs.getdbt.com/guides/dbt-python-snowpark?step=15#conclusion)

Fantastic! You’ve finished the workshop! We hope you feel empowered in using both SQL and Python in your dbt Cloud workflows with Snowflake. Having a reliable pipeline to surface both analytics and machine learning is crucial to creating tangible business value from your data.