Jesse Anderson

jesse.l.anderson@gmail.com

fROM sql to python

Leverage Your Knowledge of SQL to Learn Data-Relevant Python

Contents

[Section 1: Introduction to the Tutorial 3](#_Toc131962473)

[1.1 Choosing a Coding Platform 3](#_Toc131962474)

[1.2 Overview of the Tutorial’s Content 3](#_Toc131962475)

[Section 2: Differences Between SQL and Python 5](#_Toc131962476)

[2.1 Flexibility in Python 5](#_Toc131962477)

[2.2 Multi-Block Code in Python 5](#_Toc131962478)

[2.3 Python Libraries 5](#_Toc131962479)

[2.4 Placement of Column Names Relative to Functions 5](#_Toc131962480)

[2.5 Object-Oriented Programming 6](#_Toc131962481)

[2.6 Conclusion 6](#_Toc131962482)

[Section 3: The Data 7](#_Toc131962483)

[3.1 Code for Creating the top\_20\_countries Table in SQL 8](#_Toc131962484)

[3.2 Code for Creating the top\_20\_countries Table in Python 8](#_Toc131962485)

[3.3 Data Source 9](#_Toc131962486)

[Section 4: The SELECT and FROM Clauses in Python 10](#_Toc131962487)

[4.1 Retrieving Data in Python 10](#_Toc131962488)

[4.2 Retrieving Multiple Columns 11](#_Toc131962489)

[4.3 Conclusion 12](#_Toc131962490)

[Section 5: The WHERE Clause in Python 13](#_Toc131962491)

[5.1 Python’s Equivalent of the WHERE Clause 13](#_Toc131962492)

[5.2 Filtering Data Based on Multiple Criteria 14](#_Toc131962493)

[5.3 Adding Python’s Equivalent of OR to the WHERE Clause 14](#_Toc131962494)

[5.4 Conclusion 15](#_Toc131962495)

[Section 6: The GROUP BY Clause in Python 16](#_Toc131962496)

[6.1 Aggregate Functions and Python’s Equivalent of the GROUP BY Clause 16](#_Toc131962497)

[6.2 Aliasing Columns in Python 17](#_Toc131962498)

[6.3 Conclusion 18](#_Toc131962499)

[Section 7: The HAVING Clause in Python 19](#_Toc131962500)

[7.1 Python’s Equivalent of the HAVING Clause 19](#_Toc131962501)

[7.2 Conclusion 20](#_Toc131962502)

[Section 8: The ORDER BY Clause in Python 21](#_Toc131962503)

[8.1 Python’s Equivalent of the ORDER BY Clause 21](#_Toc131962504)

[8.2 Specifying Ascending or Descending Order 22](#_Toc131962505)

[8.3 Using sort\_values() with Multiple Columns 23](#_Toc131962506)

[8.4 Conclusion 24](#_Toc131962507)

[Section 9: Putting It All Together - Combining the Six Principal SQL Clauses in Python 25](#_Toc131962508)

[9.1 Writing a Python Equivalent of a SQL Statement with the Six Principal Clauses 25](#_Toc131962509)

[9.2 Conclusion 26](#_Toc131962510)

[Section 10: Going Forward - Three Additional SQL Concepts Coded in Python 27](#_Toc131962511)

[10.1 Returning a Specific Range of Rows from a Table 27](#_Toc131962512)

[10.2 Python’s Equivalent of the DISTINCT Clause 28](#_Toc131962513)

[10.3 Using Joins in Python 29](#_Toc131962514)

[10.4 Conclusion and Farewell 30](#_Toc131962515)

[Section 11: Sources 31](#_Toc131962516)

# Section 1: Introduction to the Tutorial

This tutorial is designed for anyone with SQL experience who’s hoping to expand their data skills by learning Python. While SQL is a requirement for a majority of jobs in data analytics, Python is a higher-level skill that many data analysts don’t learn until later in their careers. A strong command of Python can not only lead to a more senior position as a data analyst, it can also open up opportunities in other data-related fields, like data science and data engineering.

SQL and Python feature a great deal of cross-functionality, and no one with a foundation in SQL should approach Python as something entirely new. If you know how **WHERE**, **GROUP BY**, or any of the other common SQL clauses function, you’re already halfway to using the same concepts in Python—all you need to learn is how to code them.

This tutorial will show you how to code a variety of SQL statements in Python, allowing you to leverage your knowledge of SQL as a learning tool for Python. This will make your introduction to Python less intimidating. It will also help you focus on Python code that’s actually useful for a data-driven career.

By the end of this tutorial, you’ll have learned enough about Python syntax and functionality to begin a deep-dive into the language. It’s only a first step, but the first step is the most important one to take.

## 1.1 Choosing a Coding Platform

You’re free to use whatever SQL or Python platform you want, as long as the Python platform has access to the pandas library, which will be explained more in the next section. If you don’t have a preferred platform for SQL or Python, W3.com has platforms for both languages that can be easily accessed and require no installation.

## 1.2 Overview of the Tutorial’s Content

The following breaks down the tutorial section by section. If you’re trying to find out how to code a specific SQL clause in Python, feel free to jump ahead. If not, the information below will give you an idea of how the tutorial will unfold.

* Section 2 will explain some of the important differences between SQL and Python.
* Section 3 is an overview of the data table we’ll be using throughout the tutorial. If you want to try writing the code examples presented in the tutorial yourself—which is recommended—the code to create the table in both SQL and Python is provided.
* Section 4 looks at SQL’s **SELECT** and **FROM** clauses and show you how to code their equivalents in Python.
* Section 5 teaches you how to code Python’s equivalent of SQL’s **WHERE** clause.
* Section 6 explores SQL’s **GROUP BY** clause and aggregated data—two tightly connected topics.
* Section 7 builds off Section 6 by looking at how to code Python’s equivalent of SQL’s **HAVING** clause.
* Section 8 teaches you how to code Python’s equivalent of SQL’s **ORDER BY** clause.
* Section 9 will combine the clauses from the previous sections into a single SQL statement and show you how the same statement can be coded in Python.
* Section 10 will look at three additional SQL concepts and their Python equivalents. This section is meant to serve as a springboard for your Python journey after the tutorial.

Now, let’s look at some concepts that will be important to keep in mind as you begin learning Python. While SQL and Python have cross-functionality, there are some significant differences between the two languages. The next section will give a brief overview of these.

# Section 2: Differences Between SQL and Python

Before we start comparing SQL code to Python code, it’s important to understand some of the fundamental differences between the two languages. Below is a brief primer that will help smooth your transition from SQL into Python.

## 2.1 Flexibility in Python

In SQL, there’s a prescribed structure for writing a query, and it can only rarely be deviated from. You start with **SELECT** and **FROM**, and then add the other principal clauses as needed. In Python, however, there are usually many ways to write a data query. Keep this in mind as you go through the tutorial. The Python examples presented here work well for retrieving data, but they’re not the only ways to get the results you want.

## 2.2 Multi-Block Code in Python

In SQL, queries are almost always written as a single block of code. Common table expressions—which we’ll see in the tutorial’s final section—are one of the rare exceptions.

In Python, writing a data query in multiple blocks isn’t at all untypical. How a data query is divided into multiple blocks of Python code will become clear as you progress through the tutorial.

## 2.3 Python Libraries

To take full advantage of Python, it’s necessary to access libraries that contain functions not included in Python’s standard library. It’s very easy to do—you write **import** followed by the name of the library you want to access at the top of your code. It’s also common to alias libraries so they’re easier to reference while you code.

In this tutorial, we’ll be using pandas, one of Python’s most important data libraries. It’s often aliased as **pd**. The code to import pandas with the alias is as follows:

**import pandas as pd**

We’ll be using pandas in every example of Python in the tutorial, so always import it when you begin coding. Whenever you see a piece of code with **pd** before it, it means pandas is being referenced.

## 2.4 Placement of Column Names Relative to Functions

When a column is modified by a function in SQL, it’s referenced inside of the function’s parentheses. The following is an example:

**SELECT max(population)**

**FROM top\_20\_countries**

In the examples of Python we’ll be looking at, however, the functions will typically be appended to the column name. Here’s the Python equivalent of the SQL statement:

**result = top\_20\_countries[[‘population’]].max()**

Even though the SQL and Python functions share the same name, their placement relative to the column name—which here is **population**—differs. The Python is this tutorial will feature a lot of appending, so it’s important to become comfortable with it.

## 2.5 Object-Oriented Programming

Python is what’s known as an object-oriented programming language. When you see **=** after the first word of a line of code, it means you’re creating a new object. For example, the table we’ll be using is called **top\_20\_countries**. But because we’ll be retrieving specific data from the table, we have to create a new object to contain the retrieved data. For example, the following Python code retrieves a single column from the **top\_20\_countries** table, but in doing so a new object—named **result**—is created.

**result = top\_20\_countries[['country']]**

**result** is a both new object and a new table. To view it, you would write:

**print(result)**

Every object in Python has an object type. The pandas tables we’ll be accessing and creating are of the DataFrame object type. When you see DataFrame throughout the tutorial, it simply means a table created using pandas.

*Note***:** To view the results of a query in SQL, all you do is write the query and run its code. In Python, you write the query and then view it using the **print()** function. In all of the tutorial’s Python examples, you’ll need to **print()** the most recent object created in each query to view the results as a table.

## 2.6 Conclusion

With some of the major differences between SQL and Python now covered, we’re ready to learn about the data we’ll be retrieving. Go to the next section for an explanation of the **top\_20\_countries** table and the code to create it in SQL and Python.

# Section 3: The Data

To begin retrieving data in SQL and Python, we’re going to need a data source. For this tutorial, we’ll be using a table called **top\_20\_countries**, which draws on data from [Worldometer](https://www.worldometers.info/world-population/population-by-country/). The table consists of data from the 20 most populous countries in the world—meaning there are 20 rows—and includes five columns:

1. **country**: The country’s name.
2. **population**: The country’s total population.
3. **pop\_density**: The country’s population density per square kilometer.
4. **median\_age**: The median age of the country’s population.
5. **continent**: The continent where the country is located.

The code required to create the table in SQL and Python is written below. Simply copy and paste the code into your coding platform, run the code, and then query the table to make sure the data’s available.

To view in the entire table in SQL, write:

**SELECT \***

**FROM top\_20\_countries**

To view the entire in Python, write:

**print(top\_20\_countries)**

The table returned from both queries should look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **country** | **population** | **pop\_density** | **median\_age** | **continent** |
| **China** | **1439323776** | **153** | **38** | **Asia** |
| **India** | **1380004385** | **464** | **28** | **Asia** |
| **United States** | **331002651** | **36** | **38** | **North America** |
| **Indonesia** | **273523615** | **151** | **30** | **Asia** |
| **Pakistan** | **220892340** | **287** | **23** | **Asia** |
| **Brazil** | **212559417** | **25** | **33** | **South America** |
| **Nigeria** | **206139589** | **226** | **18** | **Africa** |
| **Bangladesh** | **164689383** | **1,265** | **28** | **Asia** |
| **Russia** | **145934462** | **9** | **40** | **Europe/Asia** |
| **Mexico** | **128932753** | **66** | **29** | **North America** |
| **Japan** | **126476461** | **347** | **48** | **Asia** |
| **Ethiopia** | **114963588** | **115** | **19** | **Africa** |
| **Philippines** | **109581078** | **368** | **26** | **Asia** |
| **Egypt** | **102334404** | **103** | **25** | **Africa** |
| **Vietnam** | **97338579** | **314** | **32** | **Asia** |
| **DR Congo** | **89561403** | **40** | **17** | **Africa** |
| **Turkey** | **84339067** | **110** | **32** | **Europe/Asia** |
| **Iran** | **83992949** | **52** | **32** | **Asia** |
| **Germany** | **83783942** | **240** | **46** | **Europe** |
| **Thailand** | **69799978** | **137** | **40** | **Asia** |

Once you’ve ensured that the table’s ready, go to the next section to begin learning the fundamentals of data retrieval in Python.

## 3.1 Code for Creating the top\_20\_countries Table in SQL

**CREATE TABLE top\_20\_countries (**

**country varchar(30),**

**population int,**

**pop\_density int,**

**land\_area int,**

**median\_age int,**

**continent varchar(30))**

**INSERT INTO top\_20\_countries (country, population, pop\_density, median\_age, continent)**

**VALUES ('China', 1439323776, 153, 38, 'Asia'),**

**('India', 1380004385, 464, 28, 'Asia'),**

**('United States', 331002651, 36, 38, 'North America'),**

**('Indonesia', 273523615, 151, 30, 'Asia'),**

**('Pakistan', 220892340, 287, 23, 'Asia'),**

**('Brazil', 212559417, 25, 33, 'South America'),**

**('Nigeria', 206139589, 226, 18, 'Africa'),**

**('Bangladesh', 164689383, 1265, 28, 'Asia'),**

**('Russia', 145934462, 9, 40, 'Europe/Asia'),**

**('Mexico', 128932753, 66, 29, 'North America'),**

**('Japan', 126476461, 347, 48, 'Asia'),**

**('Ethiopia', 114963588, 115, 19, 'Africa'),**

**('Philippines', 109581078, 368, 26, 'Asia'),**

**('Egypt', 102334404, 103, 25, 'Africa'),**

**('Vietnam', 97338579, 314, 32, 'Asia'),**

**('DR Congo', 89561403, 40, 17, 'Africa'),**

**('Turkey', 84339067, 110, 32, 'Europe/Asia'),**

**('Iran', 83992949, 52, 32, 'Asia'),**

**('Germany', 83783942, 240, 46, 'Europe'),**

**('Thailand', 69799978, 137, 40, 'Asia')**

## 3.2 Code for Creating the top\_20\_countries Table in Python

**import pandas as pd**

**data = [['China', 1439323776, 153, 38, 'Asia'],**

**['India', 1380004385, 464, 28, 'Asia'],**

**['United States', 331002651, 36, 38, 'North America'],**

**['Indonesia', 273523615, 151, 30, 'Asia'],**

**['Pakistan', 220892340, 287, 23, 'Asia'],**

**['Brazil', 212559417, 25, 33, 'South America'],**

**['Nigeria', 206139589, 226, 18, 'Africa'],**

**['Bangladesh', 164689383, 1265, 28, 'Asia'],**

**['Russia', 145934462, 9, 40, 'Europe/Asia'],**

**['Mexico', 128932753, 66, 29, 'North America'],**

**['Japan', 126476461, 347, 48, 'Asia'],**

**['Ethiopia', 114963588, 115, 19, 'Africa'],**

**['Philippines', 109581078, 368, 26, 'Asia'],**

**['Egypt', 102334404, 103, 25, 'Africa'],**

**['Vietnam', 97338579, 314, 32, 'Asia'],**

**['DR Congo', 89561403, 40, 17, 'Africa'],**

**['Turkey', 84339067, 110, 32, 'Europe/Asia'],**

**['Iran', 83992949, 52, 32, 'Asia'],**

**['Germany', 83783942, 240, 46, 'Europe'],**

**['Thailand', 69799978, 137, 40, 'Asia']]**

**top\_20\_countries = pd.DataFrame(data, columns=['country', 'population', 'pop\_density', 'median\_age', 'continent'])**

## 3.3 Data Source

*Countries in the world by population (2023)*. Worldometer. (n.d.). Retrieved March 25, 2023, from https://www.worldometers.info/world-population/population-by-country/

# Section 4: The SELECT and FROM Clauses in Python

**SELECT** and **FROM** form the foundation of every SQL query. These clauses are necessary to retrieve data from a table, and they are almost always used together.

## 4.1 Retrieving Data in Python

As any SQL user quickly learns (and as we saw in the previous section), **SELECT** followed by **\*** will retrieve all the data stored in the table referenced in the **FROM** statement. We’ll begin with this formulation for our transition from SQL to Python.

Here’s a SQL query using the **top\_20\_countries** table:

**SELECT \***

**FROM top\_20\_countries**

The query is Python is as follows:

**print(top\_20\_countries)**

As can be seen, the Python query is shorter than the SQL query. Because the **top\_20\_countries** table is stored as a DataFrame object, all we have to do to retrieve the full table is use the **print()** function with the table’s name referenced inside the function’s parentheses.

But what if we only want to retrieve one column, as in the following SQL query?

**SELECT country**

**FROM top\_20\_countries**

In Python, the same query would be written as:

**result = top\_20\_countries[['country']]**

*Note*: The query uses double brackets because this lets Python know that you want to return the data as a DataFrame object. If you write your query with single brackets, the returned object will be a series and won’t be structured as a table.

Python first recognizes the reference to the **top\_20\_countries** DataFrame object. It then reads the quoted text inside the brackets attached to the table reference and retrieves the column that matches the text.

The resulting table should look like this:

|  |
| --- |
| **country** |
| **China** |
| **India** |
| **United States** |
| **Indonesia** |
| **Pakistan** |
| **Brazil** |
| **Nigeria** |
| **Bangladesh** |
| **Russia** |
| **Mexico** |
| **Japan** |
| **Ethiopia** |
| **Philippines** |
| **Egypt** |
| **Vietnam** |
| **DR Congo** |
| **Turkey** |
| **Iran** |
| **Germany** |
| **Thailand** |

## 4.2 Retrieving Multiple Columns

Next, let’s look at how you can retrieve multiple columns from a table using Python.

The following is a SQL query that retrieves two columns:

**SELECT country, continent**

**FROM top\_20\_countries**

In Python, the query would be written as:

**result = top\_20\_countries[['country', 'continent']]**

The result should look like the following:

|  |  |
| --- | --- |
| **country** | **continent** |
| **China** | **Asia** |
| **India** | **Asia** |
| **United States** | **North America** |
| **Indonesia** | **Asia** |
| **Pakistan** | **Asia** |
| **Brazil** | **South America** |
| **Nigeria** | **Africa** |
| **Bangladesh** | **Asia** |
| **Russia** | **Europe/Asia** |
| **Mexico** | **North America** |
| **Japan** | **Asia** |
| **Ethiopia** | **Africa** |
| **Philippines** | **Asia** |
| **Egypt** | **Africa** |
| **Vietnam** | **Asia** |
| **DR Congo** | **Africa** |
| **Turkey** | **Europe/Asia** |
| **Iran** | **Asia** |
| **Germany** | **Europe** |
| **Thailand** | **Asia** |

If you want to add more columns to the query, you simply put the column names inside the brackets and separate them from the other column names with a comma.

The following are all valid Python queries:

**result = top\_20\_countries[['country, 'continent', 'language']]**

**result = top\_20\_countries[['population', 'size', 'pop\_density']]**

**result = top\_20\_countries[['size', 'pop\_density', 'continent', 'language']]**

## 4.3 Conclusion

The information above gives you enough knowledge to begin writing basic Python data queries. **SELECT** and **FROM** may be the most elementary clauses in SQL, but they’re also the most fundamental.

You’re now ready to learn how to add the equivalent of the SQL statement **WHERE** to your Python data queries. Go to the next section to continue expanding your knowledge of Python.

# Section 5: The WHERE Clause in Python

SQL’s **WHERE** clause allows you to filter data on a column-by-column basis. If **WHERE** is used in a SQL statement, it comes after the **FROM** statement.

## 5.1 Python’s Equivalent of the WHERE Clause

Here’s a SQL query that features the **WHERE** clause:

**SELECT \***

**FROM top\_20\_countries**

**WHERE continent = 'Asia'**

The query will only retrieve data from countries located in Asia. In Python, we can get the same result with the following query:

**result = top\_20\_countries[top\_20\_countries['continent'] == 'Asia']**

This would be the resulting table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **country** | **population** | **pop\_density** | **median\_age** | **continent** |
| **China** | **1439323776** | **153** | **38** | **Asia** |
| **India** | **1380004385** | **464** | **28** | **Asia** |
| **Indonesia** | **273523615** | **151** | **30** | **Asia** |
| **Pakistan** | **220892340** | **287** | **23** | **Asia** |
| **Bangladesh** | **164689383** | **1265** | **28** | **Asia** |
| **Japan** | **126476461** | **347** | **48** | **Asia** |
| **Philippines** | **109581078** | **368** | **26** | **Asia** |
| **Vietnam** | **97338579** | **314** | **32** | **Asia** |
| **Iran** | **83992949** | **52** | **32** | **Asia** |
| **Thailand** | **69799978** | **137** | **40** | **Asia** |

There are two important things to point out about the Python query.

The first is that *the table is referenced twice*—once outside of the brackets, and once inside the brackets. The second reference inside the brackets is necessary when filtering data with this method. If you don’t add it, your query will fail.

The second thing to point out is the use of *two consecutive equal signs* to reference **'Asia'**. Two equal signs are necessary here because, in Python, a single equal sign is used to assign a name to an object or to specify a function’s parameter. If you used a single equal sign in the query above, Python would think you were trying to name an object **'Asia']**.

*Note*: In addition to the equal sign, the other comparison operator that doesn’t transfer directly from SQL to Python is SQL’s **<>**, which, in the case of a **WHERE** clause, returns rows that are different from the comparison value. The Python equivalent of this comparison operator is **!=**. SQL’s other comparison operators **(>**, **<**, **>=**, **<=**) are the same in Python.

## 5.2 Filtering Data Based on Multiple Criteria

Now, let’s say you want to filter the data further by only including countries with a population density per square kilometer above 200. In SQL, the query would be written as:

**SELECT \***

**FROM top\_20\_countries**

**WHERE continent = 'Asia' AND pop\_density > 200**

In Python, the query would look like this:

**result = top\_20\_countries[(top\_20\_countries['continent'] == 'Asia') & (top\_20\_countries['pop\_density'] > 200)]**

The query has some elements in common with the first Python query, but there are also new elements worth noting:

* While SQL uses **AND**, Python uses **&**.
* Each column and its filter criteria are surrounded by parentheses.
* Both columns require a separate reference to the **top\_20\_countries** table.

The table would look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **country** | **population** | **pop\_density** | **median\_age** | **continent** |
| **India** | **1380004385** | **464** | **28** | **Asia** |
| **Pakistan** | **220892340** | **287** | **23** | **Asia** |
| **Bangladesh** | **164689383** | **1265** | **28** | **Asia** |
| **Japan** | **126476461** | **347** | **48** | **Asia** |
| **Philippines** | **109581078** | **368** | **26** | **Asia** |
| **Vietnam** | **97338579** | **314** | **32** | **Asia** |

## 5.3 Adding Python’s Equivalent of OR to the WHERE Clause

What if you want to write a less restrictive query by retrieving rows that are either in Asia *or* have a population density above 200, as in the following SQL query?

**SELECT \***

**FROM top\_20\_countries**

**WHERE continent = 'Asia' OR pop\_density > 200**

In Python, the query would be written as:

**result = top\_20\_countries[(top\_20\_countries['continent'] == 'Asia') | (top\_20\_countries['pop\_density'] > 200)]**

Notice that the only difference between this query and the previous Python query is the use of **|** in place of **&**. In this context, **|** functions as the equivalent of **OR** in SQL. This can be confusing for SQL users coming to Python, because **|** is typically used in SQL to concatenate strings.

If you want to add a third filtered column, all you need to do is put an **&** or a **|** after the **pop\_density** portion of the query and put a new filter criteria in parentheses, as in the following query:

**result = top\_20\_countries[(top\_20\_countries['continent'] == 'Asia') & (top\_20\_countries['pop\_density'] > 200) & (top\_20\_countries['language'] == 'Japanese')]**

## 5.4 Conclusion

Now that you’re familiar with how to use Python’s equivalent of the **WHERE** clause, we’ll move on to the **GROUP BY** clause. In the **GROUP BY** section, we’ll also look at the Python equivalents of two SQL concepts tightly connected with the **GROUP BY** clause—aggregate functions and aliasing.

# Section 6: The GROUP BY Clause in Python

If you’re planning to include the **GROUP BY** clause in your SQL statement, it’s likely going to come after the **WHERE** clause and before the **HAVING** clause, which is covered in the next section.

As you’ll see below, **GROUP** **BY** is the only SQL clause covered in this tutorial whose Python equivalent has the same name. This section will also cover two SQL concepts that are tightly connected to the **GROUP** **BY** clause—aggregate functions and aliases. Because the aggregate functions are necessary to use the **GROUP** **BY** clause, we’ll start with them.

## 6.1 Aggregate Functions and Python’s Equivalent of the GROUP BY Clause

Here’s a SQL statement using the **sum()** aggregate function and the **GROUP BY** clause:

**SELECT sum(population), continent**

**FROM top\_20\_countries**

**GROUP BY continent**

The query groups the table’s countries into their respective continents, and then adds the grouped countries’ populations to give us a total population figure for each continent in the table.

In Python, the query would be written as follows:

**result = top\_20\_countries.groupby(['continent'])****['population'].sum().reset\_index()**

Let’s break the Python query down into steps:

1. We begin by specifying the table with **top\_20\_countries**
2. The table name is followed by Python’s **groupby()** function. The column we want to group by—in this case, the **continent** column—is referenced with both quotations and brackets inside the **groupby()** function’s parentheses. This is the **groupby(['continent'])** portion of the query.
3. We specify another column and attach the aggregate function we want to use on the column. This is the **['population'].sum()** portion of the query.
4. Finally, we use the **reset\_index()** function to reset the table’s indexing. The specifics of this function go beyond the bounds of this tutorial. Simply remember that when using the **groupby()** function in Python, it’s recommended that you end the query with the **reset\_index()** function.

The resulting table should look like this:

|  |  |
| --- | --- |
| **continent** | **sum(population)** |
| **Asia** | **3965622544** |
| **Africa** | **512998984** |
| **North America** | **459935404** |
| **Europe/Asia** | **230273529** |
| **South America** | **212559417** |

As any experienced SQL user knows, we can change the **sum()** function in the SQL query above to another aggregate function to return a different result. We can do the same thing in Python by replacing the **sum()** function with another aggregate function.

The following is a list of the most common aggregate functions in SQL:

* **sum()** – Returns the sum of the aggregated data.
* **avg()** – Returns the average of the aggregated data.
* **max()** – Returns the highest value from the aggregated data.
* **min()** – Returns the lowest value from the aggregated data.
* **count()** – Returns the number of entries in the aggregated data.

The **sum()**, **max()**, and **min()** functions are all written the same in Python. Python’s equivalent to SQL’s **avg()** function is **mean()**. Python’s equivalent to SQL’s **count()** function is **size()**.

If we want to modify the Python query above to calculate the average population of the countries in each continent, all we have to do is change **sum()** to **mean()**:

**result = top\_20\_countries.groupby(['continent'])['population'].mean().reset\_index()**

This would give us the following table:

|  |  |
| --- | --- |
| **continent** | **mean(population)** |
| **Asia** | **396562254.4** |
| **North America** | **229967702** |
| **South America** | **212559417** |
| **Africa** | **128249746** |
| **Europe/Asia** | **115136764.5** |

Similarly, you could replace the query’s aggregate function with **max()**, **min()**, or **size()**. It all depends on the result you’re trying to achieve.

## 6.2 Aliasing Columns in Python

Before moving on to the **HAVING** clause, let’s look at alias creation in Python. This will be useful in the **HAVING** clause section, but it’s also an important concept more generally.

Let’s return to the first SQL query in this section and add an alias to the aggregated column using the **AS** clause:

**SELECT sum(population) AS sum\_pop, continent**

**FROM top\_20\_countries**

**GROUP BY continent**

This changes the aggregated column name from **sum(population)** to the more convenient **sum\_pop**.

In Python, we can get the same result with the following code:

**result = top\_20\_countries.groupby(['continent'])['population'].sum().rename('sum\_pop').reset\_index()**

There’s a single modification to the original query: the addition of **rename('sum\_pop')** after the **sum()** function and before the **reset\_index()** function. The **rename()** function does exactly what its name suggests—it renames the column it’s applied to with the text written in its parentheses.

The resulting table would look like this:

|  |  |
| --- | --- |
| **continent** | **sum\_pop** |
| **Asia** | **3965622544** |
| **Africa** | **512998984** |
| **North America** | **459935404** |
| **Europe/Asia** | **230273529** |
| **South America** | **212559417** |

## 6.3 Conclusion

These are the basics to using Python’s **groupby()** function, Python’s equivalent to SQL’s similarly named **GROUP BY** clause. In the next section, we’ll look at how to recreate SQL’s **HAVING** clause in Python.

# Section 7: The HAVING Clause in Python

In SQL, the **HAVING** clause functions similarly to the **WHERE** clause, the difference being that **HAVING** is used to filter aggregated data. In fact, it can be helpful to think of **HAVING**’srelationship to **GROUP BY** (the clause used to aggregate data) as comparable to **WHERE**’s relationship to the **SELECT** clause.

While it’s possible to use **HAVING** without **GROUP BY**, it’s rarely done. This section will build off of the previous one to demonstrate how you can code **HAVING** and **GROUP BY** together in Python.

## 7.1 Python’s Equivalent of the HAVING Clause

Let’s return to the query from the end of the last section. In SQL, it was written as follows:

**SELECT sum(population) AS sum\_pop, continent**

**FROM top\_20\_countries**

**GROUP BY continent**

In Python, it was written as:

**result = top\_20\_countries.groupby(['continent'])['population'].sum().rename('sum\_pop').reset\_index()**

Now, let’s add the **HAVING** clause to the SQL query to filter the aggregated data:

**SELECT sum(population) AS sum\_pop, continent**

**FROM top\_20\_countries**

**GROUP BY continent**

**HAVING sum\_pop > 50000000**

*Note*: While you’re able to reference column aliases created in the **SELECT** clause in most versions of SQL, there are versions where it’s not possible. If the above query returns an error, change **sum\_pop** in the **HAVING** clause to **sum(population)**.

With the **HAVING** clause, we’ve filtered the result to continents whose total population is above 500 million. We’ve also used the **AS** clause to make the filtered column easier to reference in the **HAVING** clause.

To write an equivalent query in Python, we’re going to return to a concept introduced in the “Differences Between SQL and Python” section—multi-block queries in Python.

We’ll start our query with the same Python code above before adding more code to filter the aggregated **sum\_pop** column:

**result = top\_20\_countries.groupby(['continent'])['population'].sum().rename('sum\_pop').reset\_index()**

Remember that the object **result** is a new DataFrame. It’s based on data from the **top\_20\_countries** table, but it’s a separate table.

The next line of code we’ll add uses the same syntax as Python’s equivalent to the **WHERE** clause. This makes sense—**WHERE** and **HAVING** both filter data, and because we’ve already created the new **result** table with the aggregated data, the only thing left to do is filter it.

Let’s pretend we’re starting from scratch, so we can see the two blocks of code together.

**result = top\_20\_countries.groupby(['continent'])['population'].sum().rename('sum\_pop').reset\_index()**

**result2 = result[result['sum\_pop'] > 500000000]**

It’s as simple as that. We reference the aggregated column created in **result** and filter it using the syntax explained in the tutorial’s **WHERE** section. We now have a new table, called **result2**. It should look like this:

|  |  |
| --- | --- |
| **continent** | **sum\_pop** |
| **Asia** | **3965622544** |
| **Africa** | **512998984** |

## 7.2 Conclusion

Python’s **HAVING** equivalent is the same as its **WHERE** equivalent. The only difference is that in the first case, we’re filtering aggregated data.

This section was the first to demonstrate a multi-line query in Python. We’ll see this again in the “Putting it All Together” section. But before we get to that, we have one principal SQL clause left to look at—**ORDER BY**.

# Section 8: The ORDER BY Clause in Python

The **ORDER BY** clause always comes after the other principal clauses in a SQL statement. It’s Python equivalent is the **sort\_values** function and should be easy to understand for any experienced SQL user.

## 8.1 Python’s Equivalent of the ORDER BY Clause

Let’s start with a SQL query:

**SELECT \***

**FROM top\_20\_countries**

**ORDER BY median\_age**

This query will return every row in the table, but the rows will be ordered by median age. **ORDER BY** defaults to returning the data in an ascending order, which in this case means the median ages would be returned from lowest to highest.

Python’s **sort\_values** functions also defaults to an ascending order. We can write an equivalent of the SQL query as follows:

**result = top\_20\_languages.sort\_values('median\_age')**

But before we see the resulting table, let’s make our queries more specific by explicitly ordering the ages in an ascending order:

In SQL, the query would be written:

**SELECT \***

**FROM top\_20\_countries**

**ORDER BY median age ASC**

In Python, it would be written:

**result = top\_20\_languages.sort\_values('median\_age', ascending = True)**

Python’s equivalent of **ASC** is what’s called a parameter in the **sort\_values()** function. You add the parameter by putting a comma after the column you’re ordering by and then writing **ascending = True**.

This is the resulting table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **country** | **population** | **pop\_density** | **median\_age** | **continent** |
| **DR Congo** | **89561403** | **40** | **17** | **Africa** |
| **Nigeria** | **206139589** | **226** | **18** | **Africa** |
| **Ethiopia** | **114963588** | **115** | **19** | **Africa** |
| **Pakistan** | **220892340** | **287** | **23** | **Asia** |
| **Egypt** | **102334404** | **103** | **25** | **Africa** |
| **Philippines** | **109581078** | **368** | **26** | **Asia** |
| **India** | **1380004385** | **464** | **28** | **Asia** |
| **Bangladesh** | **164689383** | **1265** | **28** | **Asia** |
| **Mexico** | **128932753** | **66** | **29** | **North America** |
| **Indonesia** | **273523615** | **151** | **30** | **Asia** |
| **Vietnam** | **97338579** | **314** | **32** | **Asia** |
| **Turkey** | **84339067** | **110** | **32** | **Europe/Asia** |
| **Iran** | **83992949** | **52** | **32** | **Asia** |
| **Brazil** | **212559417** | **25** | **33** | **South America** |
| **China** | **1439323776** | **153** | **38** | **Asia** |
| **United States** | **331002651** | **36** | **38** | **North America** |
| **Russia** | **145934462** | **9** | **40** | **Europe/Asia** |
| **Thailand** | **69799978** | **137** | **40** | **Asia** |
| **Germany** | **83783942** | **240** | **46** | **Europe** |
| **Japan** | **126476461** | **347** | **48** | **Asia** |

## 8.2 Specifying Ascending or Descending Order

What if we want to order the **median\_age** column in descending order, meaning the ages would be returned from highest to lowest?

In SQL, the query would be:

**SELECT \***

**FROM top\_20\_countries**

**ORDER BY median\_age DESC**

In Python, all you do is change the **True** in the **sort\_values()** function to **False**.

**result = top\_20\_languages.sort\_values('median\_age', ascending = False)**

The following table would be the result:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **country** | **population** | **pop\_density** | **median\_age** | **continent** |
| **Japan** | **126476461** | **347** | **48** | **Asia** |
| **Germany** | **83783942** | **240** | **46** | **Europe** |
| **Russia** | **145934462** | **9** | **40** | **Europe/Asia** |
| **Thailand** | **69799978** | **137** | **40** | **Asia** |
| **China** | **1439323776** | **153** | **38** | **Asia** |
| **United States** | **331002651** | **36** | **38** | **North America** |
| **Brazil** | **212559417** | **25** | **33** | **South America** |
| **Vietnam** | **97338579** | **314** | **32** | **Asia** |
| **Turkey** | **84339067** | **110** | **32** | **Europe/Asia** |
| **Iran** | **83992949** | **52** | **32** | **Asia** |
| **Indonesia** | **273523615** | **151** | **30** | **Asia** |
| **Mexico** | **128932753** | **66** | **29** | **North America** |
| **India** | **1380004385** | **464** | **28** | **Asia** |
| **Bangladesh** | **164689383** | **1265** | **28** | **Asia** |
| **Philippines** | **109581078** | **368** | **26** | **Asia** |
| **Egypt** | **102334404** | **103** | **25** | **Africa** |
| **Pakistan** | **220892340** | **287** | **23** | **Asia** |
| **Ethiopia** | **114963588** | **115** | **19** | **Africa** |
| **Nigeria** | **206139589** | **226** | **18** | **Africa** |
| **DR Congo** | **89561403** | **40** | **17** | **Africa** |

## 8.3 Using sort\_values() with Multiple Columns

Finally, let’s look at how you can order multiple columns in a single query. We’ll retrieve the **continent** column and retrieve in it an ascending order. We’ll also retrieve the **population** column in descending order. Because the **continent** column’s data type is a string (i.e., text), descending means it will be ordered in reverse alphabetical order.

**SELECT \***

**FROM top\_20\_countries**

**ORDER BY continent ASC, population DESC**

In Python, the equivalent query would be written as:

**result = top\_20\_countries.sort\_values(['continent', 'population'], ascending = [True, False])**

When referencing two columns instead of one, the **sort\_values()** function changes slightly. First, the column names are placed in brackets. Without brackets, Python would expect the text after the comma separating **'continent'** and **'population'** to be a part of the ascending parameter. It would find **'population'** where it expects to find **ascending**, and the query would fail.

Second, the **ascending** condition’s **True** and **False** portion is also in brackets. As would be expected, the first condition is applied to the first column and the second condition is applied to the second column—the **ascending** condition for **'continent'** is **True**, and the condition for **'population'** is **False**.

This would be the resulting table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **country** | **population** | **pop\_density** | **median\_age** | **continent** |
| **Nigeria** | **206139589** | **226** | **18** | **Africa** |
| **Ethiopia** | **114963588** | **115** | **19** | **Africa** |
| **Egypt** | **102334404** | **103** | **25** | **Africa** |
| **DR Congo** | **89561403** | **40** | **17** | **Africa** |
| **China** | **1439323776** | **153** | **38** | **Asia** |
| **India** | **1380004385** | **464** | **28** | **Asia** |
| **Indonesia** | **273523615** | **151** | **30** | **Asia** |
| **Pakistan** | **220892340** | **287** | **23** | **Asia** |
| **Bangladesh** | **164689383** | **1265** | **28** | **Asia** |
| **Japan** | **126476461** | **347** | **48** | **Asia** |
| **Philippines** | **109581078** | **368** | **26** | **Asia** |
| **Vietnam** | **97338579** | **314** | **32** | **Asia** |
| **Iran** | **83992949** | **52** | **32** | **Asia** |
| **Thailand** | **69799978** | **137** | **40** | **Asia** |
| **Germany** | **83783942** | **240** | **46** | **Europe** |
| **Russia** | **145934462** | **9** | **40** | **Europe/Asia** |
| **Turkey** | **84339067** | **110** | **32** | **Europe/Asia** |
| **United States** | **331002651** | **36** | **38** | **North America** |
| **Mexico** | **128932753** | **66** | **29** | **North America** |
| **Brazil** | **212559417** | **25** | **33** | **South America** |

If you wanted to add a third column to the **sort\_values()** function, all you would do is add the column in quotations to the first part of the function, separated from the other column names by a comma. You would then specify its **ascending** condition in the function’s second part.

## 8.4 Conclusion

With **sort\_values()**—Python’s equivalent of SQL’s **ORDER** **BY** clause—we’ve now covered SQL’s six principal clauses. Next, we’ll see how you can code a Python query that combines them.

# Section 9: Putting It All Together - Combining the Six Principal SQL Clauses in Python

Now that we’ve seen how the six principal SQL clauses can be coded in Python, let’s look at how we can convert a single SQL statement that uses all six clauses into Python. As we did in the **HAVING** section, here we’ll take advantage of Python’s functionality by writing our equivalent using multiple blocks of code.

## 9.1 Writing a Python Equivalent of a SQL Statement with the Six Principal Clauses

Here’s our SQL statement:

**SELECT AVG(population\_density) AS apd, continent**

**FROM top\_20\_countries**

**WHERE population > 150000000**

**GROUP BY continent**

**HAVING apd > 100**

**ORDER BY continent ASC**

Before looking at the Python equivalent, let’s break down the SQL statement so we know exactly what it’s doing:

* In the **SELECT** line, we’re finding the average population density and aliasing the column. We’re also selecting the **continent** column and leaving it unaggregated.
* The **FROM** clause is specifying the table we’re drawing our data from.
* The **WHERE** clause is restricting the rows retrieved to countries whose population is above 150 million. This is the first time we’re filtering data in the statement. We’ll filter it further with the **HAVING** clause.
* In the **GROUP BY** line, we’re grouping the aggregated **apd** column from the **SELECT** line by continent.
* In the **HAVING** line, we’re filtering our aggregated data to only include rows with an average population density above 100.
* Finally, in the **ORDER BY** clause, we’re ordering the table alphabetically by continent.

Now, we’ll see the whole Python equivalent and then break it down step by step.

**result = top\_20\_countries[top\_20\_countries['population'] > 150000000]**

**result2 = result.groupby(['continent'])['population\_density'].mean().rename(‘apd’).reset\_index()**

**result3 = result2[result2['apd'] > 100]**

**final\_result = result3[['apd', 'continent']].sort\_values('continent', ascending = True)**

Here’s what each block of code is doing:

* In **result**, we’re filtering the data to countries with a population above 150 million as we did with the **WHERE** clause in SQL.
* In **result2**, we’re grouping the data by continent and then finding the average population density for the countries in each continent—this is what we did with **GROUP BY** and **HAVING** in the SQL statement. With the **rename()** function, we’re aliasing the aggregated population density column. Finally, with **reset\_index()**, we’re resetting the index numbers for each column (though, as mentioned in the **GROUP BY** section, Python beginners don’t need to focus on the specifics of the **reset\_index()** function).
* In **result3**, we’re filtering the aggregated apd column to only include values above 100.
* In **final\_result**, we’re selecting the columns we want returned in our final table and then ordering them alphabetically by continent using the **sort\_values()** function. In this block of code, we’ve combined the **SELECT** and **ORDER** BY equivalents.

*Note*: It may seem strange that we’ve saved the **SELECT** equivalent for the last block, given that it’s the first clause in the SQL statement. But the choice makes more sense when we consider the fact that SQL’s order of execution—meaning the way a SQL statement is actually processed—reads the **SELECT** clause after every other clause except **ORDER BY**. There’s a reason for this, but the theory behind it goes beyond the goals of this tutorial.

The resulting table should look like this:

|  |  |
| --- | --- |
| **continent** | **apd** |
| **Africa** | **226** |
| **Asia** | **464** |

## 9.2 Conclusion

You’ve now seen how to code the Python equivalent of a SQL statement that includes all six principal clauses. It’s important to keep in mind, however, that Python offers multiple potential SQL equivalents for retrieving data—the techniques demonstrated in this tutorial are just a few of many available options.

The next section of this tutorial briefly explores three additional aspects of SQL—the **DISTINCT** clause, how to select a specified range of rows from a table, and joins. By seeing how these concepts can be applied in Python, you’ll be better prepared to continue learning data-relevant Python beyond this tutorial.

# Section 10: Going Forward - Three Additional SQL Concepts Coded in Python

To end the tutorial, we’ll explore three additional common SQL functions and their Python equivalents. By the end of this section, you’ll have a more expansive idea of how SQL can transfer over to Python code and be better prepared to continue studying Python.

We’ll begin with Python’s equivalent of SQL’s **DISTINCT** clause and then move onto selecting a specific range of rows from a table. The clause for this function varies depending on the version of SQL, but you may be familiar with it if you’ve seen the **TOP**, **LIMIT**, or **FETCH** clauses. Finally, we’ll look at joins, a fundamental SQL concept whose equivalent Python function can be easily grasped by any experienced SQL user.

## 10.1 Returning a Specific Range of Rows from a Table

Let’s say you want to return the **top\_20\_countries** table, but you only want the first five rows. In SQL, the relevant clause could be **TOP**, **LIMIT**, **FETCH**, or **ROWNUM**—it all depends on the version of SQL you’re using. In MySQL , for instance, you would use **LIMIT**, as demonstrated in the following statement:

**SELECT \***

**FROM top\_20\_countries**

**LIMIT 5**

SQL Server, on the other hand, uses **TOP**:

**SELECT TOP 5 \***

**FROM top\_20\_countries**

The resulting table in each case would be the same.

To return the same table in Python, we use a concept called slicing. Slicing can be somewhat confusing at first because of how Python indexes rows. The first row in a table has an index value of 0, the second row has an index value of 1, and so on.

With this in mind, here’s the Python equivalent of the SQL statements above:

**result = top\_20\_countries[0:5]**

The slicing syntax consists of two brackets, a colon, and the beginning and end index values of the range you want to return. You might be wondering why the value on the right side of the colon is 5 if, as mentioned above, the fifth row’s index value would be 4. This is another quirk of Python slicing—the rows returned *don’t include the value to the right of the colon*. Instead, the row returned is the row that comes immediately before the value. This can be confusing until you’ve seen or coded enough examples to master the concept.

Here's the resulting table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **country** | **population** | **pop\_density** | **median\_age** | **continent** |
| **China** | **1439323776** | **153** | **38** | **Asia** |
| **India** | **1380004385** | **464** | **28** | **Asia** |
| **United States** | **331002651** | **36** | **38** | **North America** |
| **Indonesia** | **273523615** | **151** | **30** | **Asia** |
| **Pakistan** | **220892340** | **287** | **23** | **Asia** |

Let’s look at a more slightly more complicated example of slicing by adding the **WHERE** clause to the first SQL statement above:

**SELECT \***

**FROM top\_20\_countries**

**WHERE continent = 'Asia'**

**LIMIT 5**

In Python, we use the **WHERE** equivalent code we saw in the **WHERE** section and then append slicing to it. Slicing typically comes last when you’re coding to retrieve data in Python.

**result = top\_20\_countries[top\_20\_countries['continent'] == 'Asia'][0:5]**

Here’s the result:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **country** | **population** | **pop\_density** | **median\_age** | **continent** |
| **China** | **1439323776** | **153** | **38** | **Asia** |
| **India** | **1380004385** | **464** | **28** | **Asia** |
| **Indonesia** | **273523615** | **151** | **30** | **Asia** |
| **Pakistan** | **220892340** | **287** | **23** | **Asia** |
| **Bangladesh** | **164689383** | **1265** | **28** | **Asia** |

## 10.2 Python’s Equivalent of the DISTINCT Clause

The **DISTINCT** clause is SQL allows you filter out any duplicates from a table. For instance, if you wanted to see the continents included in the **top\_20\_countries** table without seeing the same continent listed twice, you could write the following statement:

**SELECT DISTINCT continent**

**FROM top\_20\_countries**

Python offers a straightforward way to carry out the same procedure—the **drop\_duplicates()** function. As with the **sort\_values()** function, the **drop\_duplicates()** function does exactly what its name implies.

Here’s the Python equivalent of the SQL statement:

**result = top\_20\_countries[['continent']].drop\_duplicates()**

The resulting table would look like this:

|  |
| --- |
| **continent** |
| **Africa** |
| **Asia** |
| **Europe** |
| **Europe/Asia** |
| **North America** |
| **South America** |

## 10.3 Using Joins in Python

Finally, to end the section and the tutorial, we’ll look at the **merge()** function—Python’s equivalent of SQL’s **JOIN**. We’ll also be using SQL’s common table expression functionality to create a second table. Common table expressions are an intermediate SQL concept. If you’re unfamiliar with them, you should still be able to grasp the **JOIN** portion of the SQL code.

Common table expressions allow us to create new tables from existing tables. This is important, because it’s easiest to demonstrate the **JOIN** clause with two tables.

Here’s the common table expression code in SQL:

**WITH a AS**

**(SELECT country, median\_age**

**FROM top\_20\_countries),**

**WITH b AS**

**(SELECT country, continent**

**FROM top\_20\_countries)**

**SELECT a.country, a.language, b.continent**

**FROM a**

**INNER JOIN b ON a.country = b.country**

We’ve created two new tables based on the **top\_20\_countries** table’s data. The first is aliased **a** and includes only the **country** and **median\_age** columns. The second is aliased b and includes only the **country** and **continent** columns from the original table. In the final block, we’ve joined the two new tables using their shared **country** columns.

The Python equivalent of the SQL code is as follows:

**a = top\_20\_countries[['country', 'median\_age']]**

**b = top\_20\_countries[['country', 'continent']]**

**result = a.merge('b', how='inner')**

Here’s a breakdown of the Python code:

* In the first line, we’ve created a new table **a**, which only includes the **country** and **language** columns.
* In the second line, we’ve created another new table, **b**, which only includes the **country** and **continent** columns.
* With the third and final line, we’ve used the **merge()** function from Python’s pandas library to join the two new tables. The function is first appended to **a**. Then, inside the function’s parentheses, we choose the table we want to join (in this case, table **b**) and choose what kind of join we want to use (in this case, an inner join). You could replace **'inner'** with any of the other join types you’ve encountered in SQL.

There are optional parameters that can be specified in the **merge()** function, but they’re not necessary for the most common types of joins.

The result of the code looks like this:

|  |  |  |
| --- | --- | --- |
| **country** | **median\_age** | **continent** |
| **China** | **38** | **Asia** |
| **India** | **28** | **Asia** |
| **United States** | **38** | **North America** |
| **Indonesia** | **30** | **Asia** |
| **Pakistan** | **23** | **Asia** |
| **Brazil** | **33** | **South America** |
| **Nigeria** | **18** | **Africa** |
| **Bangladesh** | **28** | **Asia** |
| **Russia** | **40** | **Europe/Asia** |
| **Mexico** | **29** | **North America** |
| **Japan** | **48** | **Asia** |
| **Ethiopia** | **19** | **Africa** |
| **Philippines** | **26** | **Asia** |
| **Egypt** | **25** | **Africa** |
| **Vietnam** | **32** | **Asia** |
| **DR Congo** | **17** | **Africa** |
| **Turkey** | **32** | **Europe/Asia** |
| **Iran** | **32** | **Asia** |
| **Germany** | **46** | **Europe** |
| **Thailand** | **40** | **Asia** |

## 10.4 Conclusion and Farewell

You’ve now added three new SQL equivalents to your knowledge of Python. With these and the six principal SQL clauses, your foundation for continuing to study data-relevant Python is solid.

The ideal next step is to find a platform where you can practice coding Python to reinforce what you’ve learned here. Stratascratch, LeetCode, and CodingWars are just a few of many options to explore.

# Section 11: Sources

*Countries in the world by population (2023)*. Worldometer. (n.d.). Retrieved March 25, 2023, from https://www.worldometers.info/world-population/population-by-country/

McIntire, G. M. I., Martin, B., & Washington, L. (n.d.). *Python pandas tutorial: A complete introduction for beginners*. Learn Data Science. Retrieved March 22, 2023, from https://www.learndatasci.com/tutorials/python-pandas-tutorial-complete-introduction-for-beginners/

*Python Pandas - Merging/Joining*. Tutorials Point. (n.d.). Retrieved March 22, 2023, from https://www.tutorialspoint.com/python\_pandas/python\_pandas\_merging\_joining.htm

Rosidi, N. (n.d.). *Python cheat sheet – technical concepts for the job interview*. StrataScratch. Retrieved March 22, 2023, from https://www.stratascratch.com/blog/python-cheat-sheet-technical-concepts-for-the-job-interview/?utm\_source=newsletter&utm\_medium=click&utm\_campaign=011723%2Bpython%2Bcheatsheet