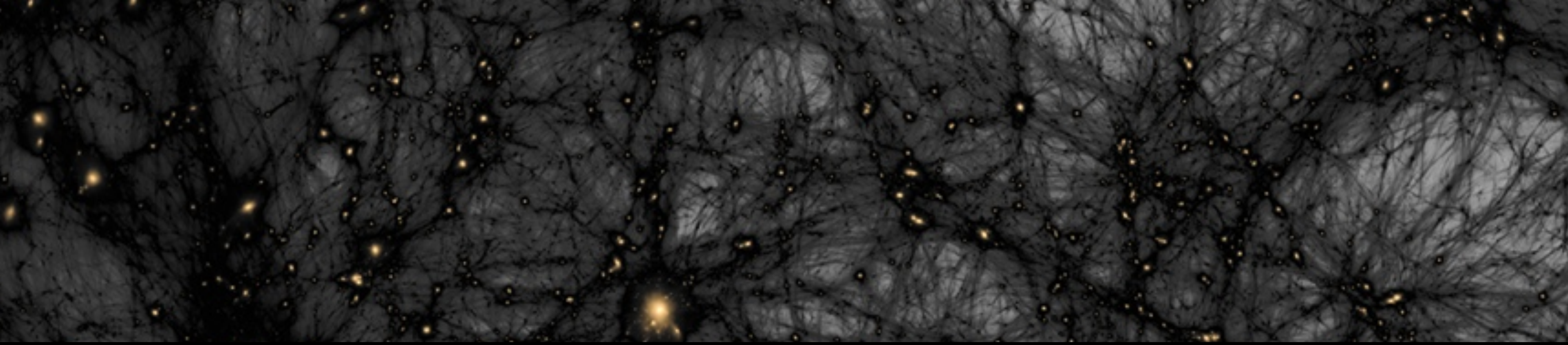


An introduction to SQL

Alkistis Pourtsidou
Queen Mary, University of London

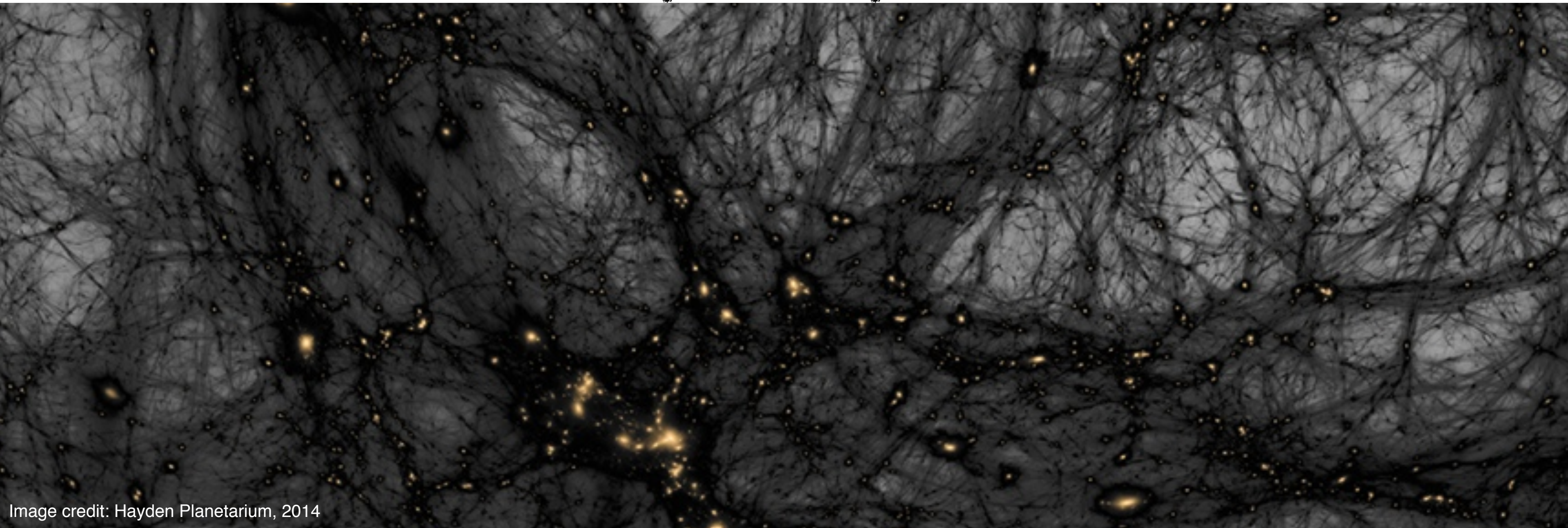


Image credit: Hayden Planetarium, 2014

WHAT IS SQL?

- SQL stands for Structured Query Language
- Used to query (“talk to”) a database server
- Data manipulation, database creation
- Almost all companies use databases to store their data
- Have a look at the Indeed Job Search Engine…tens of thousands of jobs mentioning SQL!
- **https://www.indeed.co.uk/**

WHY DO WE NEED DATABASES?

- **Concurrency:** multiple simultaneous changes to data
- Data changes regularly
- Large data sets but only need subsets
- Sharing large data sets
- Rapid queries
- Data web interfaces (dynamic data)

WAYS TO USE SQL

- Standard console command (e.g. *mysql -u user -p dbname*)
- GUI interfaces often available
- Interfaces to many programming languages (Python, R, …)
- **SQLite - use SQL without a database server:** this is what we are going to use in our tutorial
- **PostgreSQL - the world's most advanced open source database:** we'll see how that works later on

MORE ABOUT DATABASES

- A database server can contain many databases
- Basically, databases are collections of tables with rows (observations) and columns (variables)
- Limited mathematical operations available
- Very good at combining information from several related tables
- **We’ll explore the above (and more) in detail**

EXPLORING THE SERVER

- A given server can support multiple databases
- Each database contains many tables
- Each table contains many columns
- But keeping things under control is straightforward!
  - `SHOW DATABASES;`
  - `SHOW TABLES IN database;`
  - `SHOW COLUMNS IN table;`
  - `DESCRIBE table;` - shows the columns and their types

VARIABLE TYPES

- SQL supports a variety of different formats for storing information

Numeric

- `INTEGER`, `SMALLINT`, `BIGINT`
- `NUMERIC(w,d)`, `DECIMAL(w,d)` - numbers with width `w` and `d` decimal places
- `REAL`, `DOUBLE PRECISION` - machine and database dependent
- `FLOAT(p)` - floating point number with `p` binary digits of precision

VARIABLE TYPES

- SQL supports a variety of different formats for storing information

Character

- `CHARACTER(L)` - a fixed-length character of length `L`
- `CHARACTER VARYING(L)` or `VARCHAR(L)` - supports maximum length of `L`

Binary

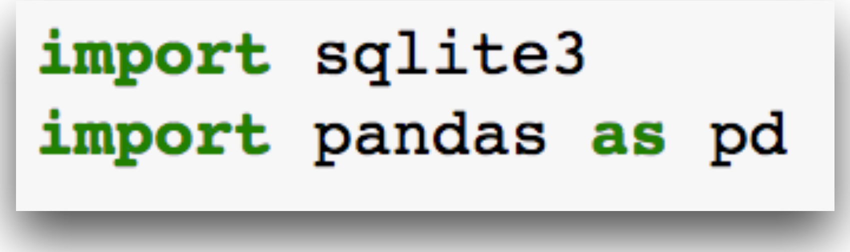
- `BIT(L)`, `BIT VARYING(L)` - like corresponding characters
- `BINARY LARGE OBJECT(L)` or `BLOB(L)`

Temporal

- `DATE`
- `TIME`
- `TIMESTAMP`

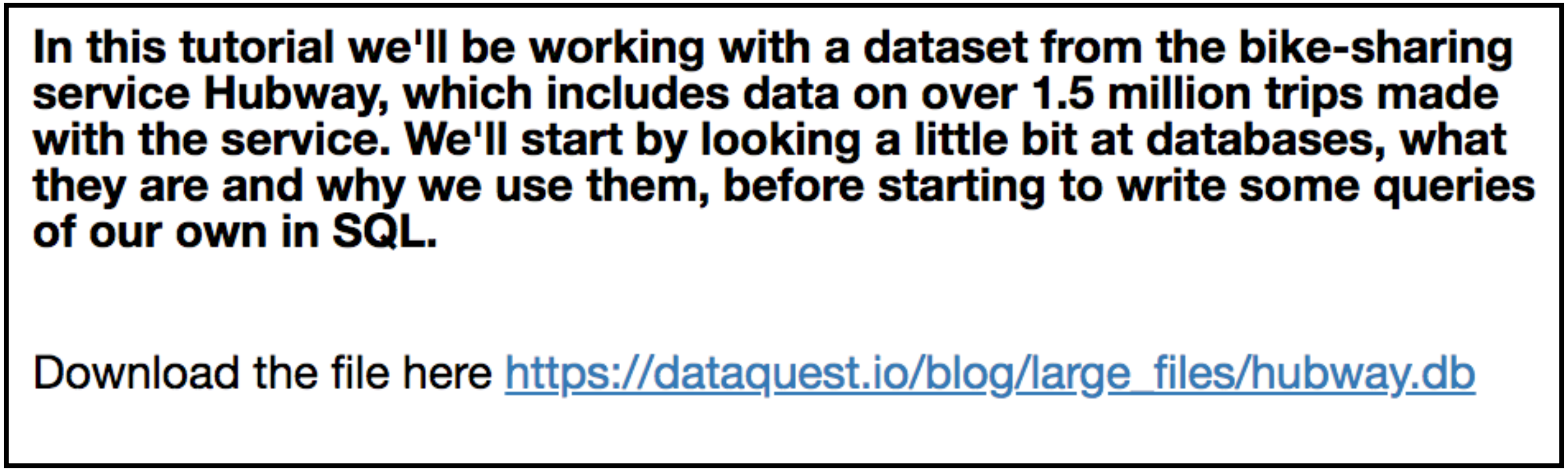
SQL: THE BASICS

- Enough intro! Let's dive into SQL with hands-on examples
- We will use **SQLite3**, which is part of Python, for details see https://www.pythoncentral.io/introduction-to-sqlite-in-python/
- SQLite is an embedded SQL database engine. It doesn't have a separate server process, which makes it really easy to use, immediately.
- Using Python (in the form of a Jupyter notebook) to run our SQL code allows us to use Pandas for importing our results and make everything look nice and clear!



```
import sqlite3
import pandas as pd
```

SQL: THE BASICS

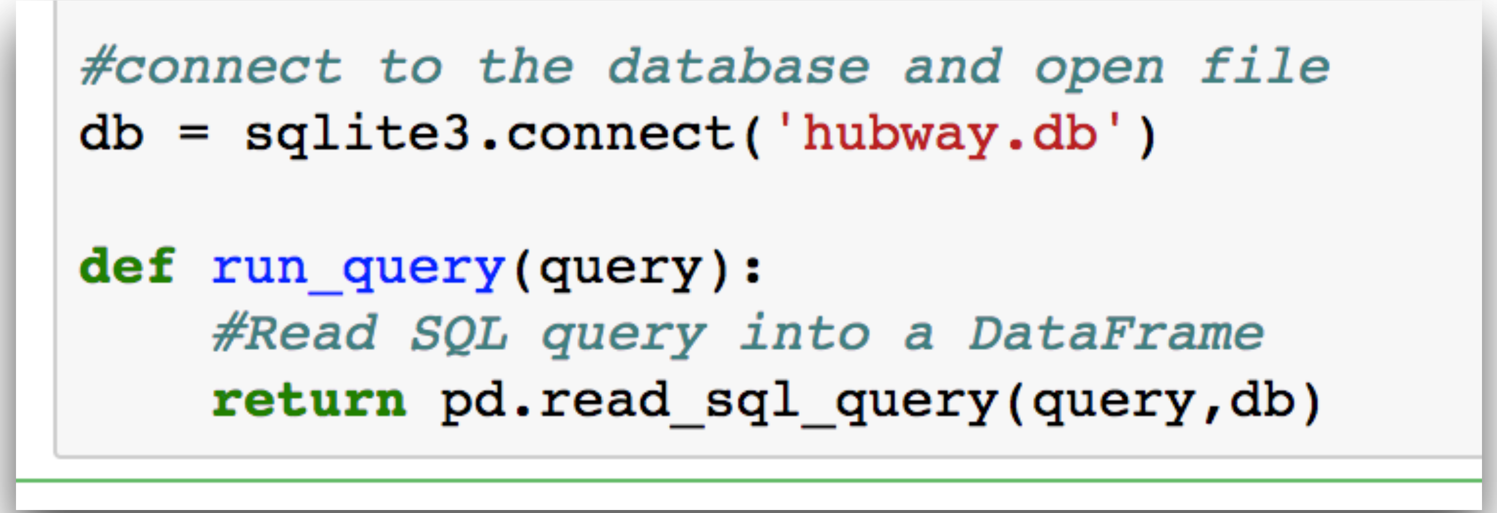


In this tutorial we'll be working with a dataset from the bike-sharing service Hubway, which includes data on over 1.5 million trips made with the service. We'll start by looking a little bit at databases, what they are and why we use them, before starting to write some queries of our own in SQL.

Download the file here https://dataquest.io/blog/large_files/hubway.db

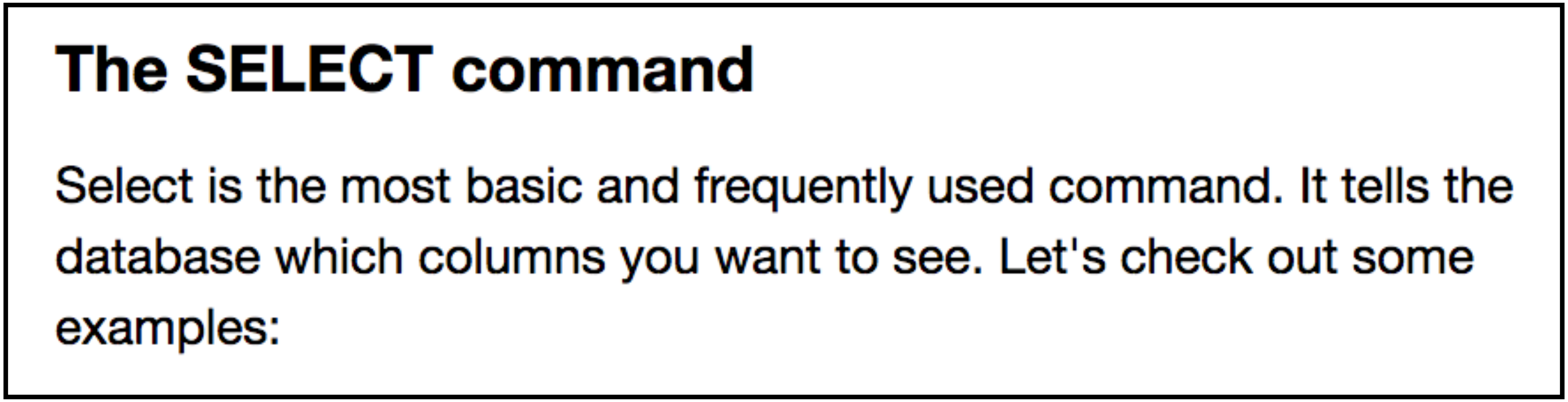
SQL: THE BASICS

- Let’s first define a function that takes our query, stored as a string, as an input.
- Then shows the result as a formatted data frame (we’ll see this in action in a bit…)



```
#connect to the database and open file
db = sqlite3.connect('hubway.db')

def run_query(query):
    #Read SQL query into a DataFrame
    return pd.read_sql_query(query,db)
```

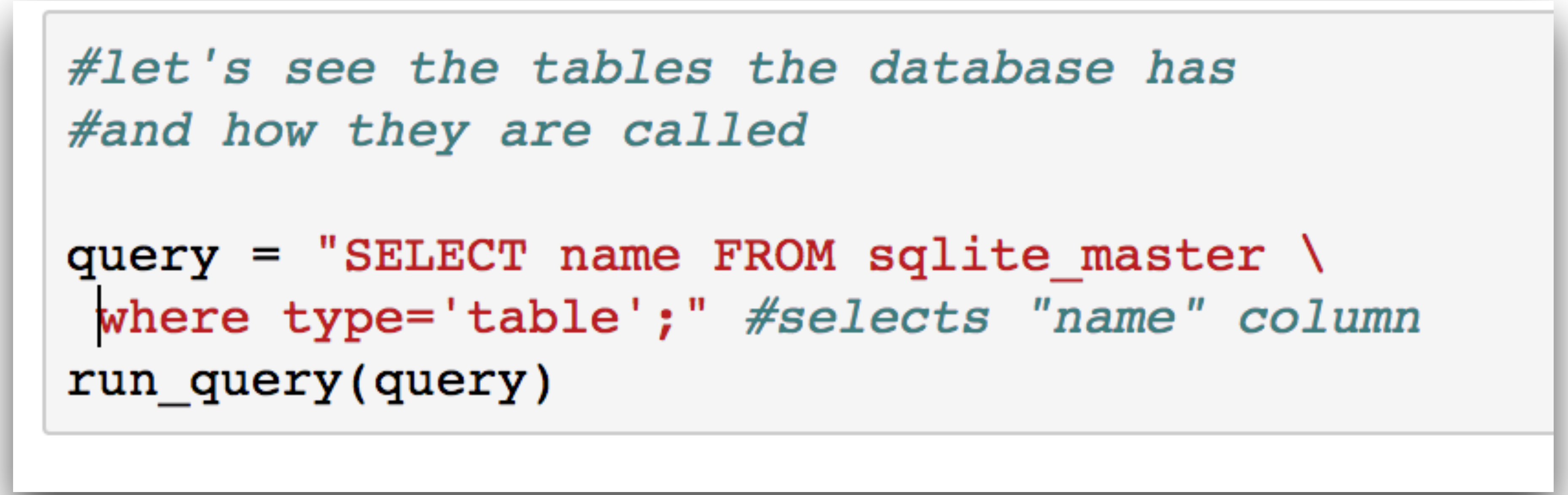
SQL: THE BASICS



The SELECT command

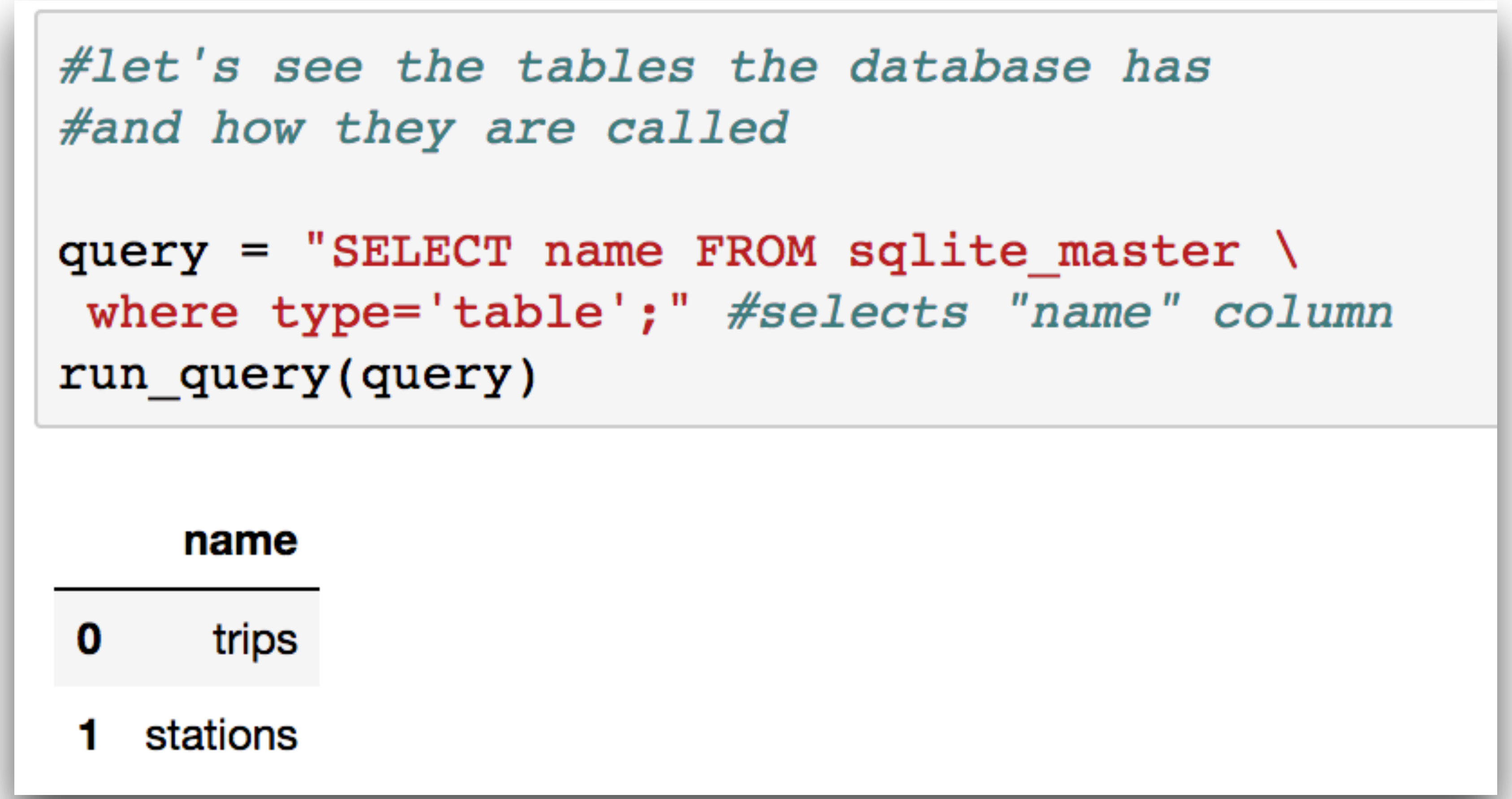
Select is the most basic and frequently used command. It tells the database which columns you want to see. Let's check out some examples:

SQL: THE BASICS



```
#let's see the tables the database has
#and how they are called

query = "SELECT name FROM sqlite_master \
 where type='table';" #selects "name" column
run_query(query)
```

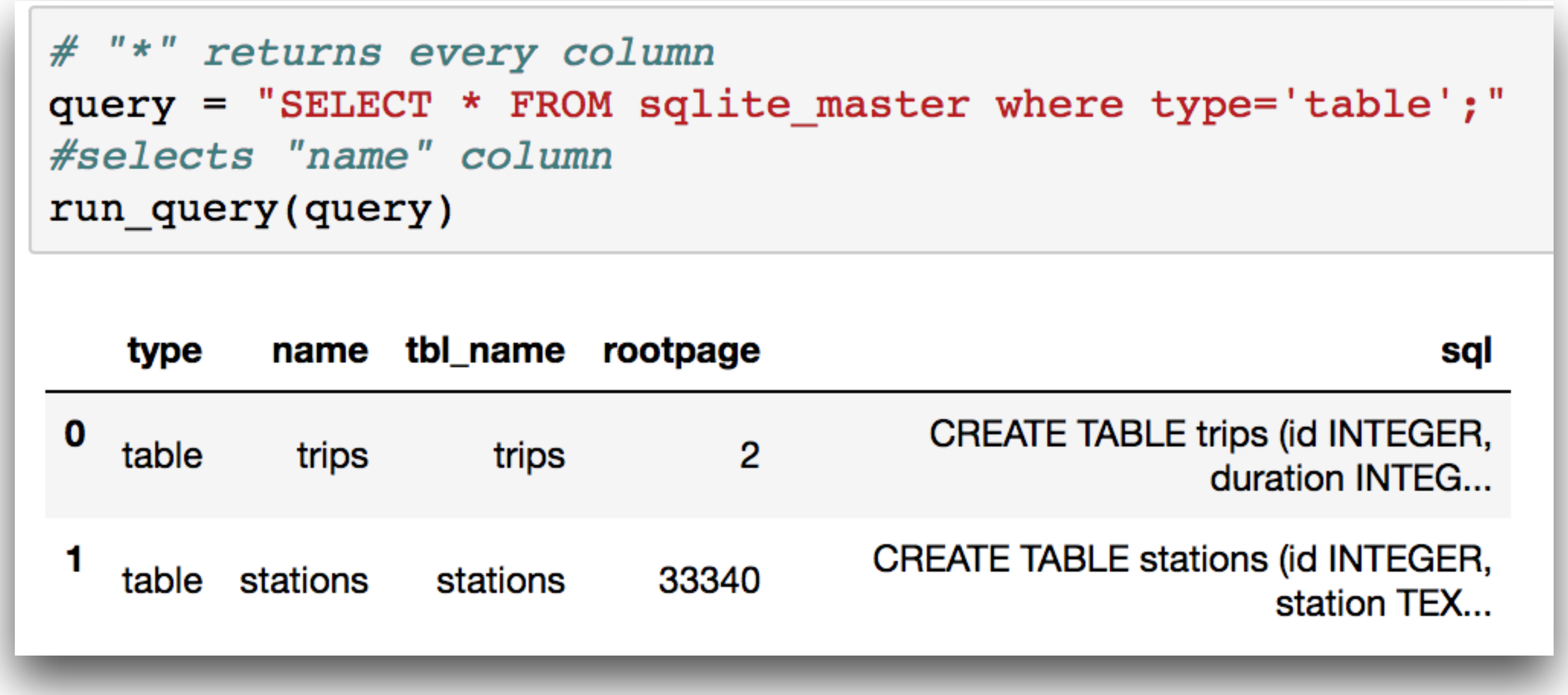
SQL: THE BASICS



```
#let's see the tables the database has
#and how they are called

query = "SELECT name FROM sqlite_master \
 where type='table';" #selects "name" column
run_query(query)
```

	name
0	trips
1	stations

SQL: THE BASICS

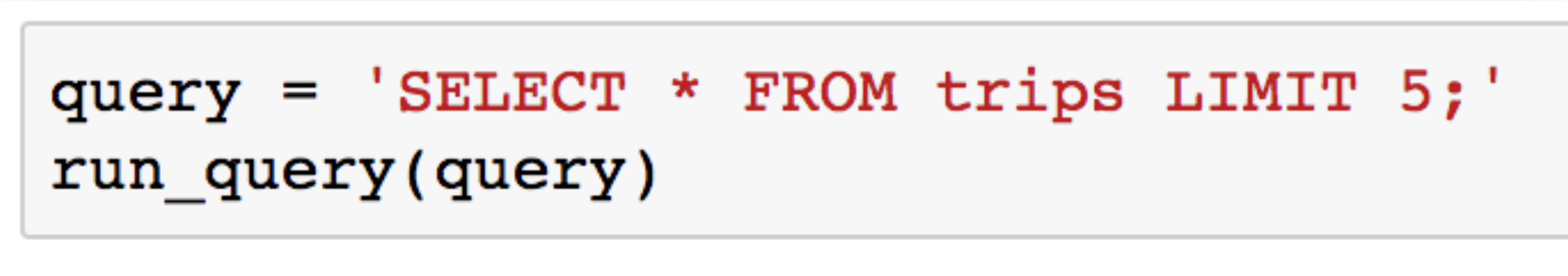


```
# "*" returns every column
query = "SELECT * FROM sqlite_master where type='table';"
#selects "name" column
run_query(query)
```

	type	name	tbl_name	rootpage	sql
0	table	trips	trips	2	CREATE TABLE trips (id INTEGER, duration INTEG...
1	table	stations	stations	33340	CREATE TABLE stations (id INTEGER, station TEX...

SQL: THE BASICS

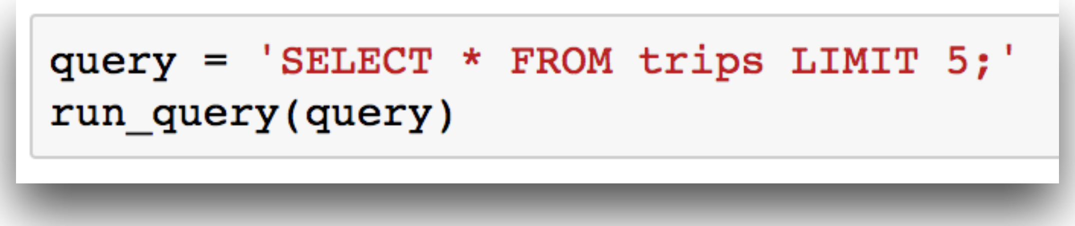
- As we saw, the database has two tables, TRIPS and STATIONS. We will first work with the TRIPS table. Let's see what kind of information it contains:



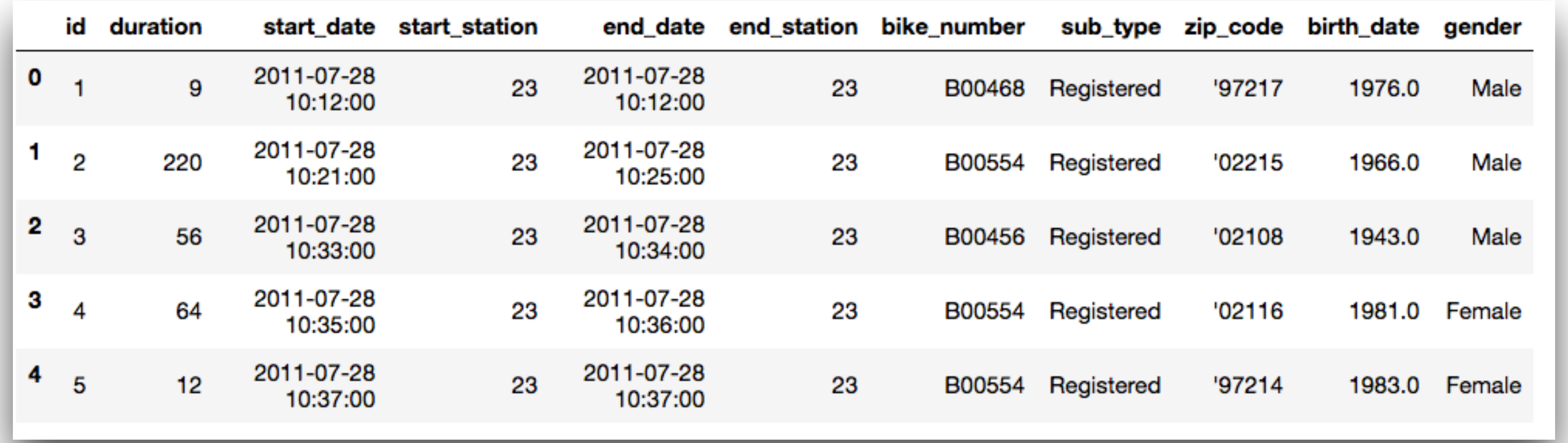
```
query = 'SELECT * FROM trips LIMIT 5;'
run_query(query)
```

SQL: THE BASICS

- See all columns:



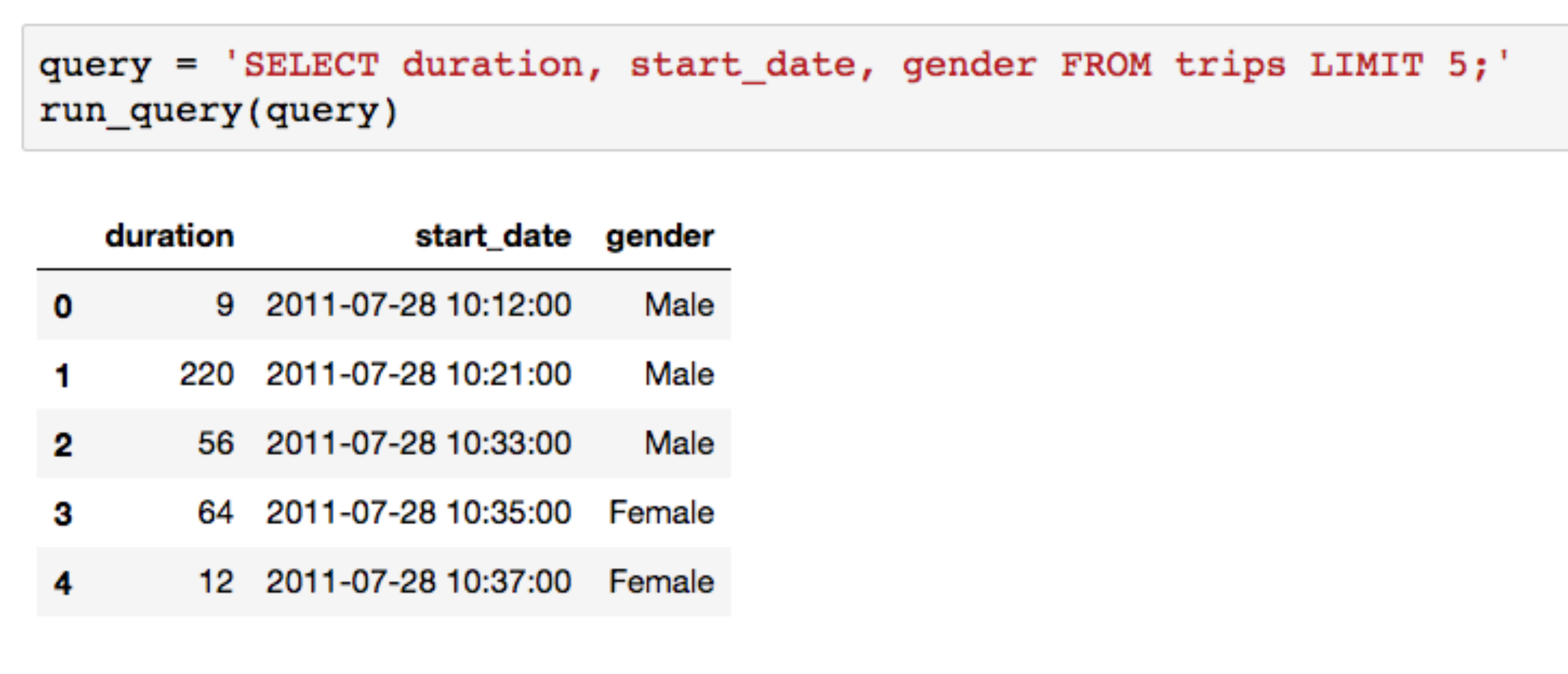
```
query = 'SELECT * FROM trips LIMIT 5;'
run_query(query)
```



	id	duration	start_date	start_station	end_date	end_station	bike_number	sub_type	zip_code	birth_date	gender
0	1	9	2011-07-28 10:12:00	23	2011-07-28 10:12:00	23	B00468	Registered	'97217	1976.0	Male
1	2	220	2011-07-28 10:21:00	23	2011-07-28 10:25:00	23	B00554	Registered	'02215	1966.0	Male
2	3	56	2011-07-28 10:33:00	23	2011-07-28 10:34:00	23	B00456	Registered	'02108	1943.0	Male
3	4	64	2011-07-28 10:35:00	23	2011-07-28 10:36:00	23	B00554	Registered	'02116	1981.0	Female
4	5	12	2011-07-28 10:37:00	23	2011-07-28 10:37:00	23	B00554	Registered	'97214	1983.0	Female

SQL: THE BASICS

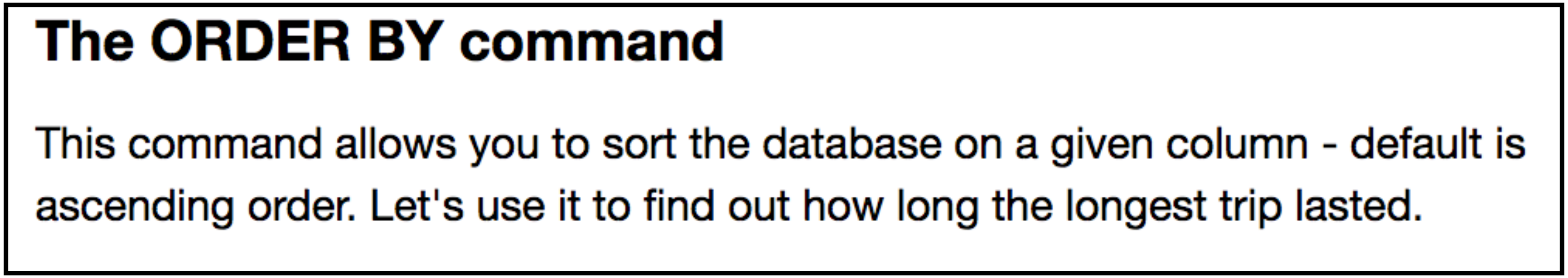
- See specific columns:



```
query = 'SELECT duration, start_date, gender FROM trips LIMIT 5;'
run_query(query)
```

	duration	start_date	gender
0	9	2011-07-28 10:12:00	Male
1	220	2011-07-28 10:21:00	Male
2	56	2011-07-28 10:33:00	Male
3	64	2011-07-28 10:35:00	Female
4	12	2011-07-28 10:37:00	Female

SQL: THE BASICS

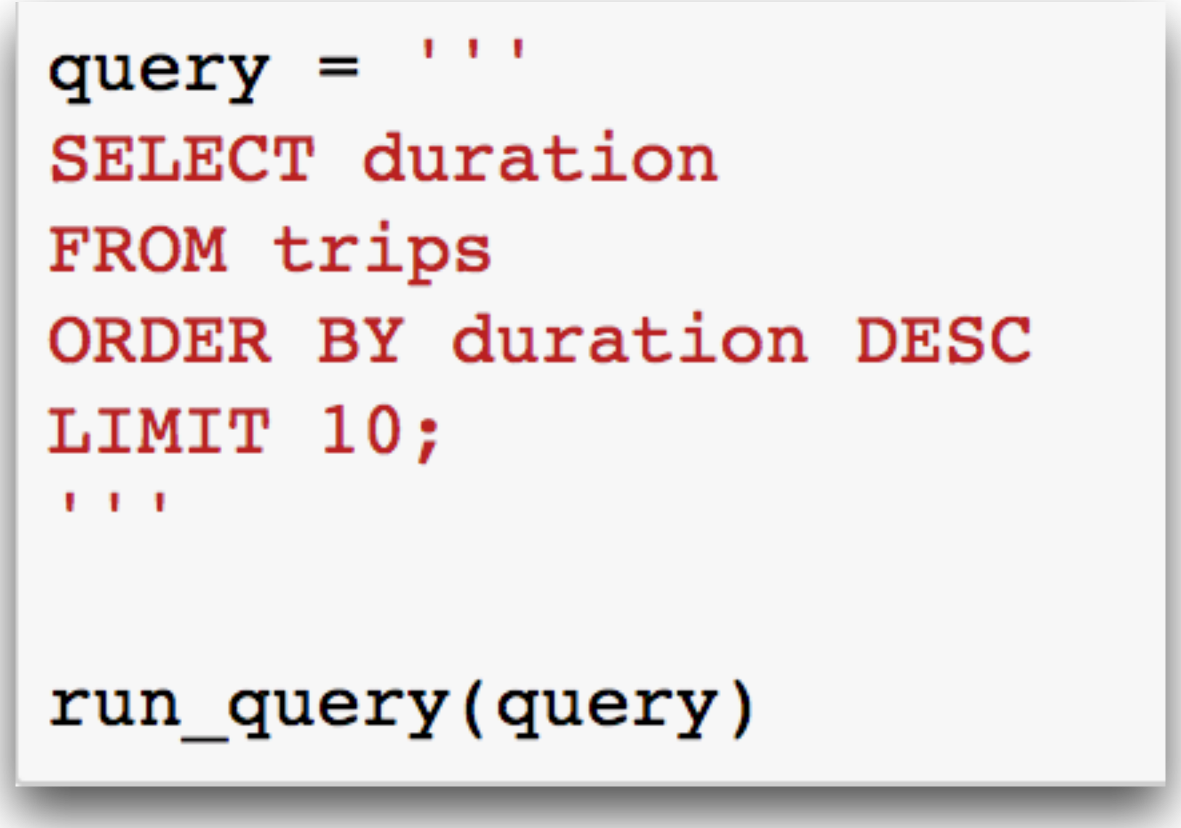


The ORDER BY command

This command allows you to sort the database on a given column - default is ascending order. Let's use it to find out how long the longest trip lasted.

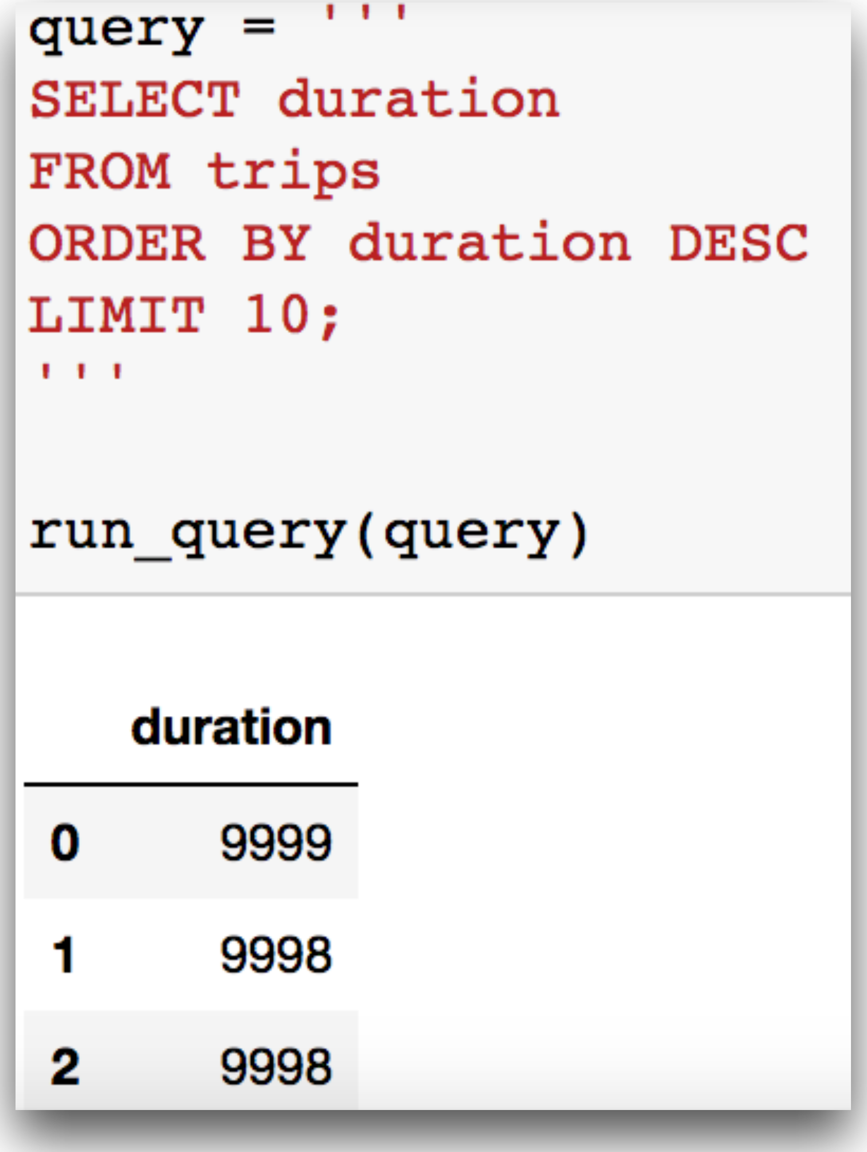
SQL: THE BASICS

✦ **DESC**: Descending



```
query = '''
SELECT duration
FROM trips
ORDER BY duration DESC
LIMIT 10;
'''


run_query(query)
```

SQL: THE BASICS



```
query = '''
SELECT duration
FROM trips
ORDER BY duration DESC
LIMIT 10;
'''

run_query(query)
```

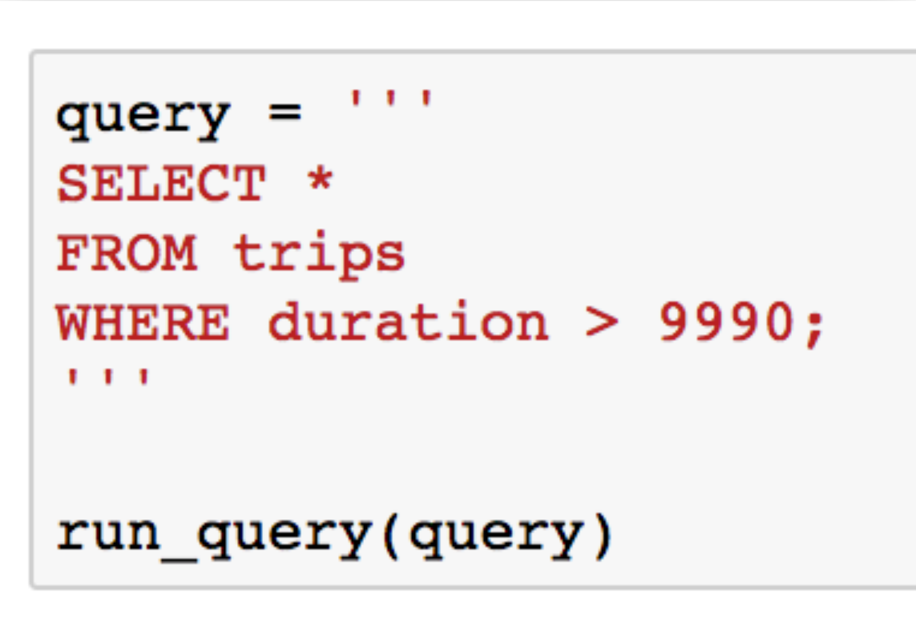
	duration
0	9999
1	9998
2	9998

- ✦ The longest trip lasts a bit less than 3 hours.

SQL: THE BASICS

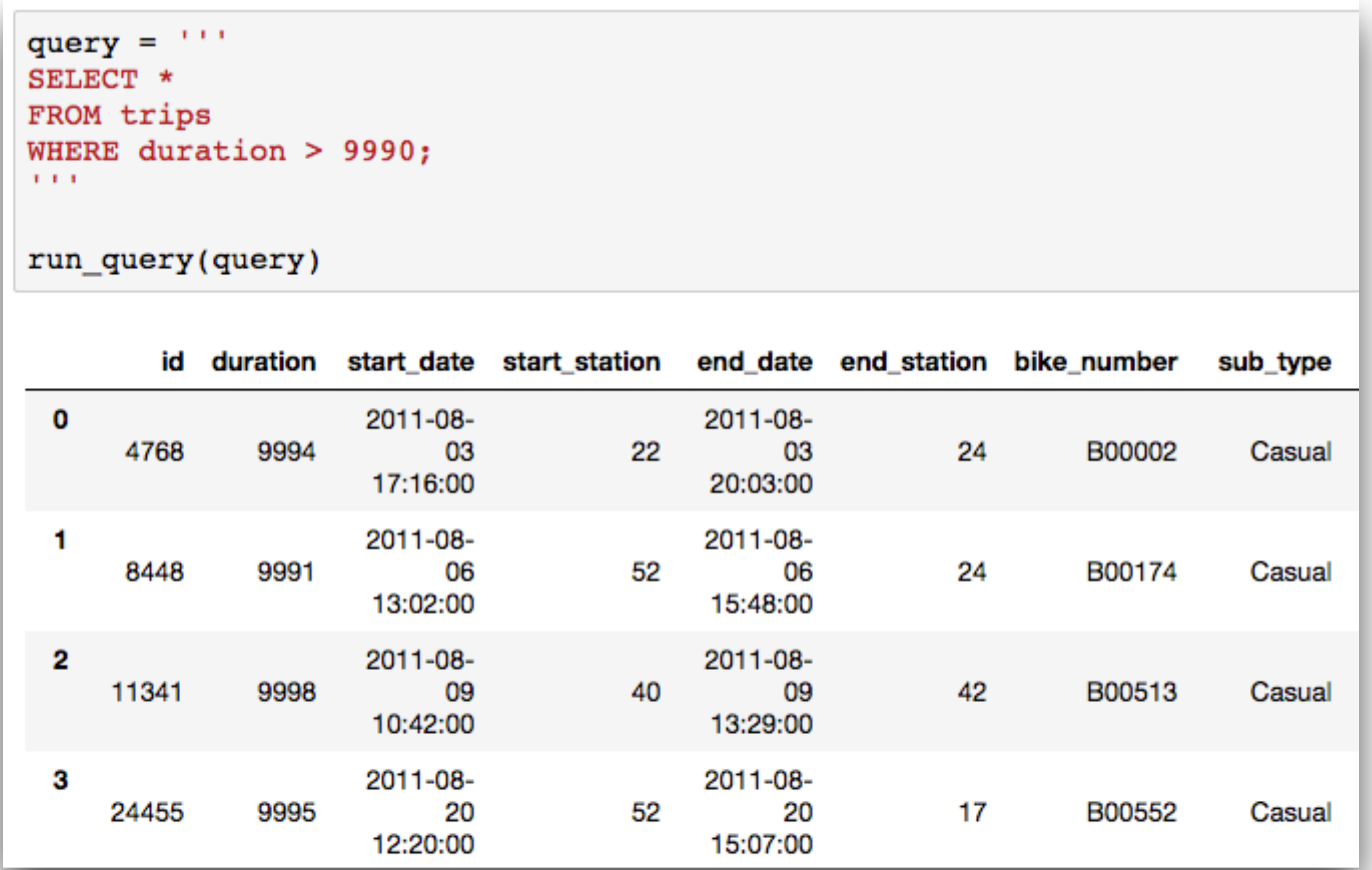
The WHERE command

The WHERE command is used to specify a certain subset of data. For example you could use the following command to return every trip with a duration longer than 9990 seconds:



```
query = '''
SELECT *
FROM trips
WHERE duration > 9990;
'''


run_query(query)
```

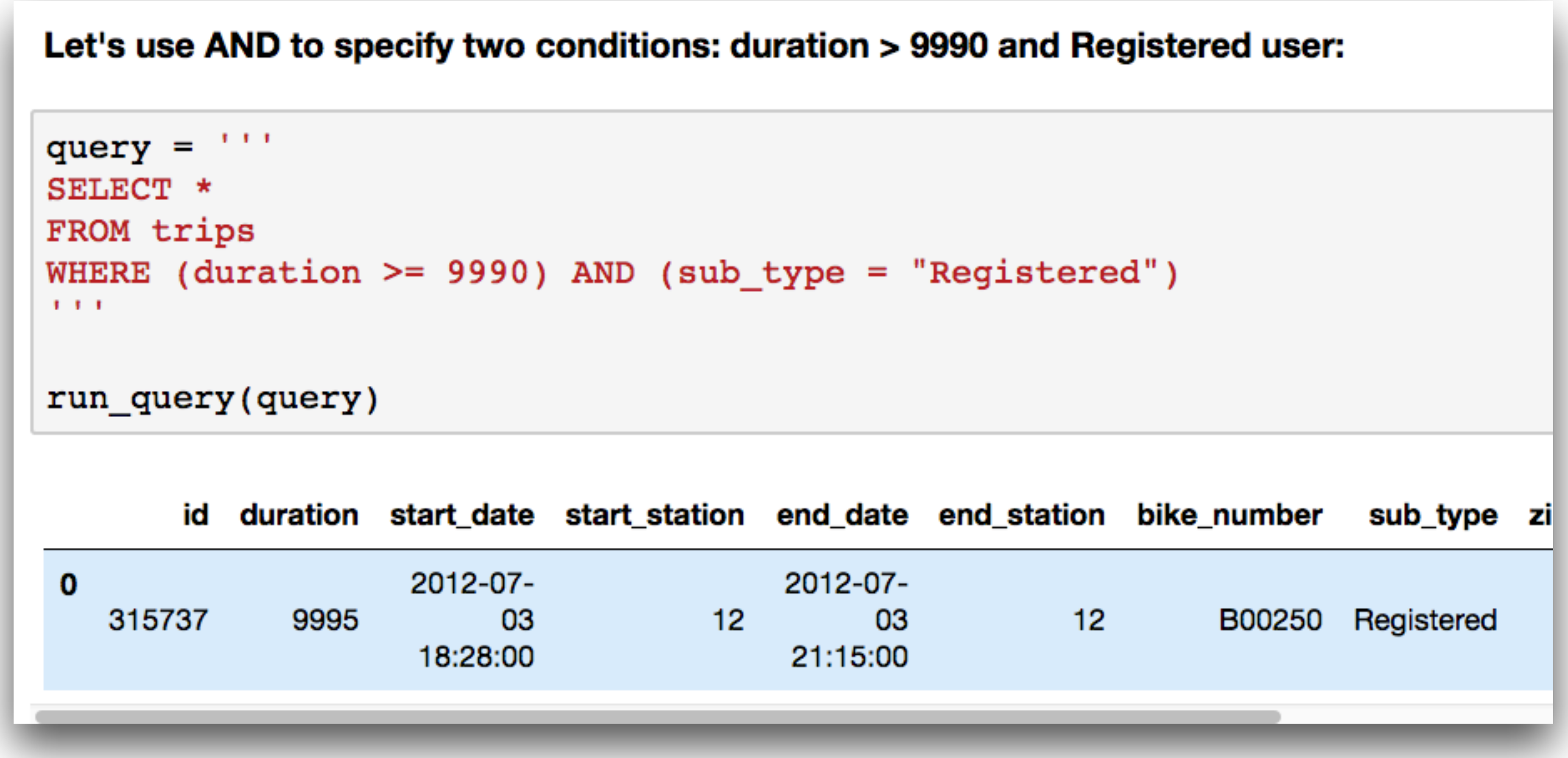
SQL: THE BASICS



```
query = '''
SELECT *
FROM trips
WHERE duration > 9990;
'''

run_query(query)
```

	id	duration	start_date	start_station	end_date	end_station	bike_number	sub_type
0	4768	9994	2011-08-03 17:16:00	22	2011-08-03 20:03:00	24	B00002	Casual
1	8448	9991	2011-08-06 13:02:00	52	2011-08-06 15:48:00	24	B00174	Casual
2	11341	9998	2011-08-09 10:42:00	40	2011-08-09 13:29:00	42	B00513	Casual
3	24455	9995	2011-08-20 12:20:00	52	2011-08-20 15:07:00	17	B00552	Casual

SQL: THE BASICS

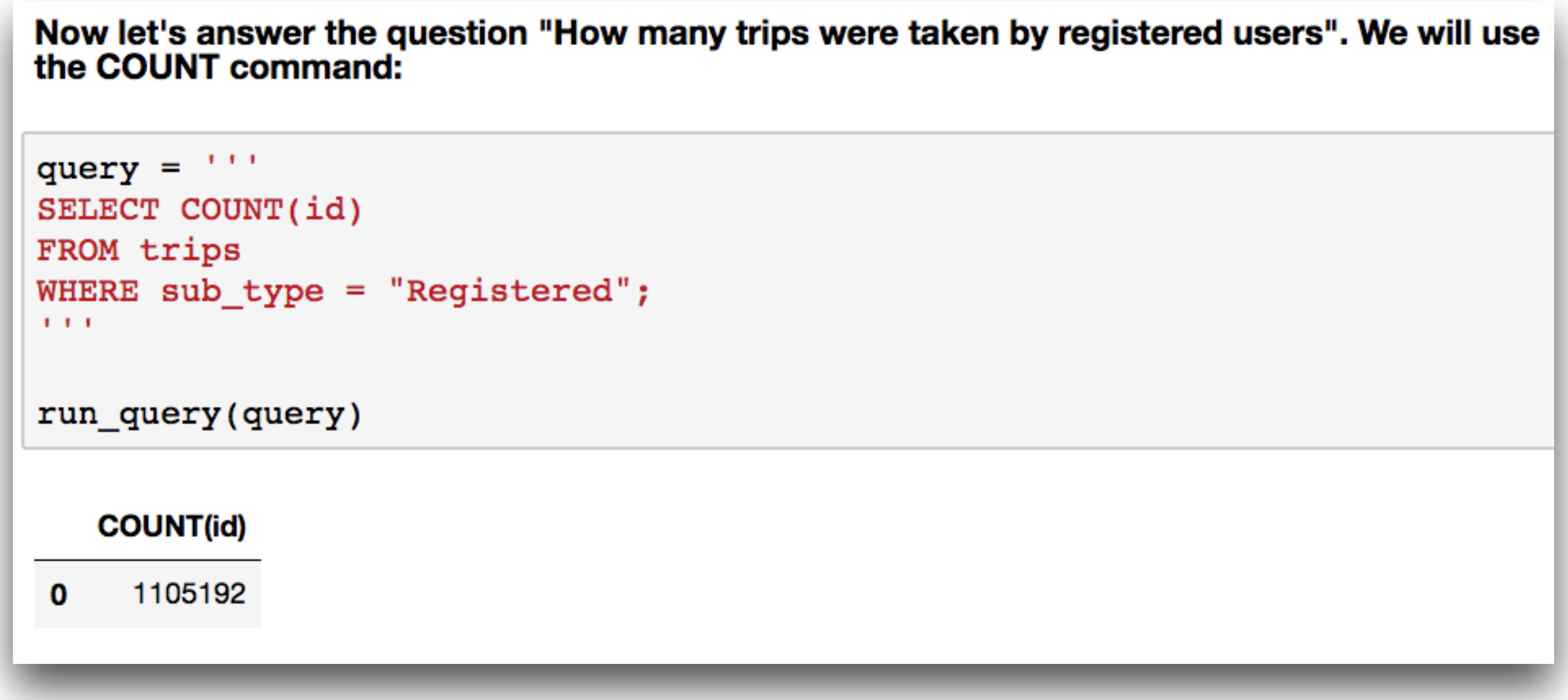


Let's use AND to specify two conditions: duration > 9990 and Registered user:

```
query = '''
SELECT *
FROM trips
WHERE (duration >= 9990) AND (sub_type = "Registered")
'''

run_query(query)
```

	id	duration	start_date	start_station	end_date	end_station	bike_number	sub_type	zi
0	315737	9995	2012-07-03 18:28:00	12	2012-07-03 21:15:00	12	B00250	Registered	

SQL: THE BASICS

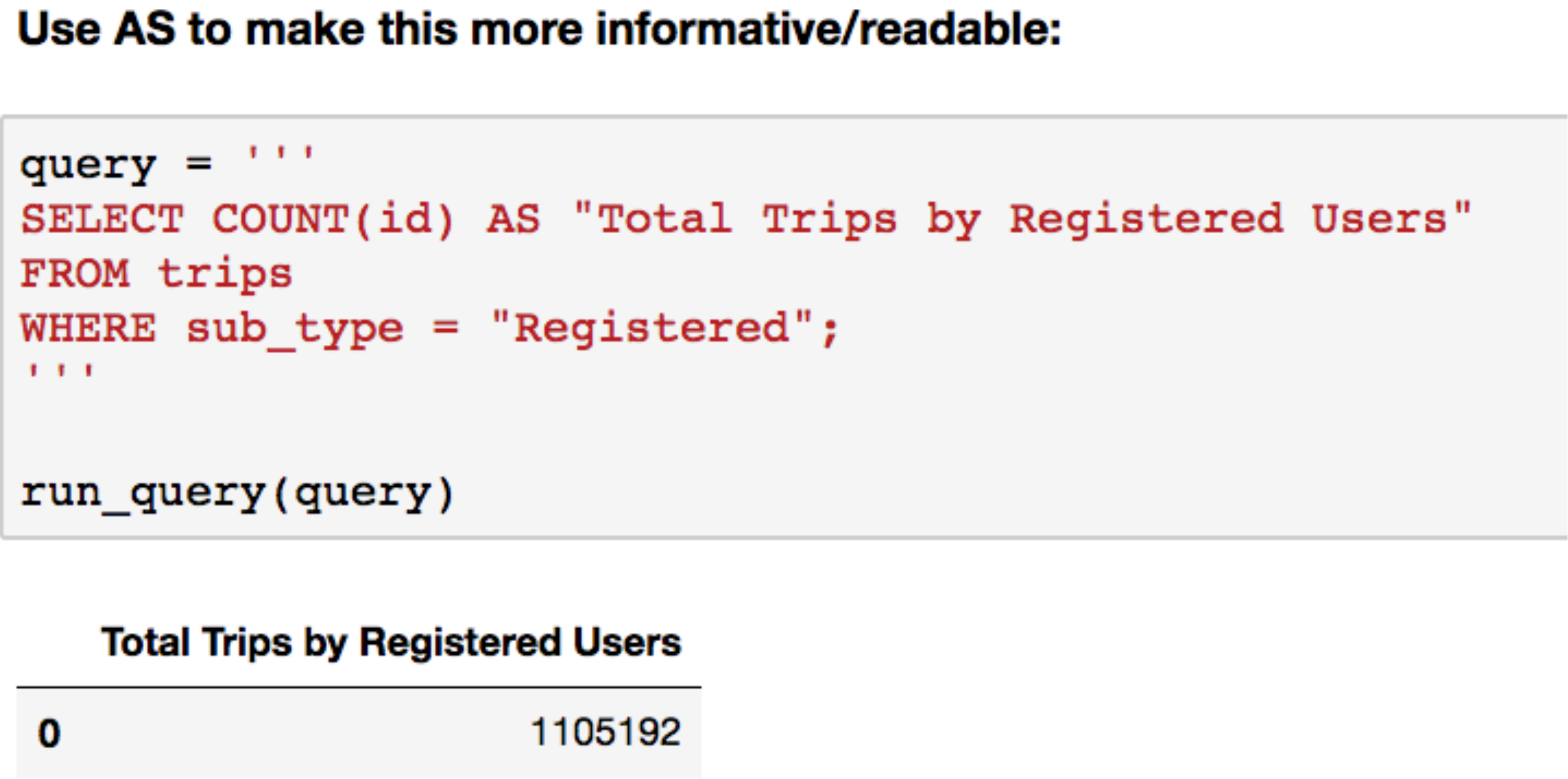


Now let's answer the question "How many trips were taken by registered users". We will use the COUNT command:

```
query = '''
SELECT COUNT(id)
FROM trips
WHERE sub_type = "Registered";
'''


run_query(query)
```

```
     COUNT(id)
0      1105192
```

SQL: THE BASICS

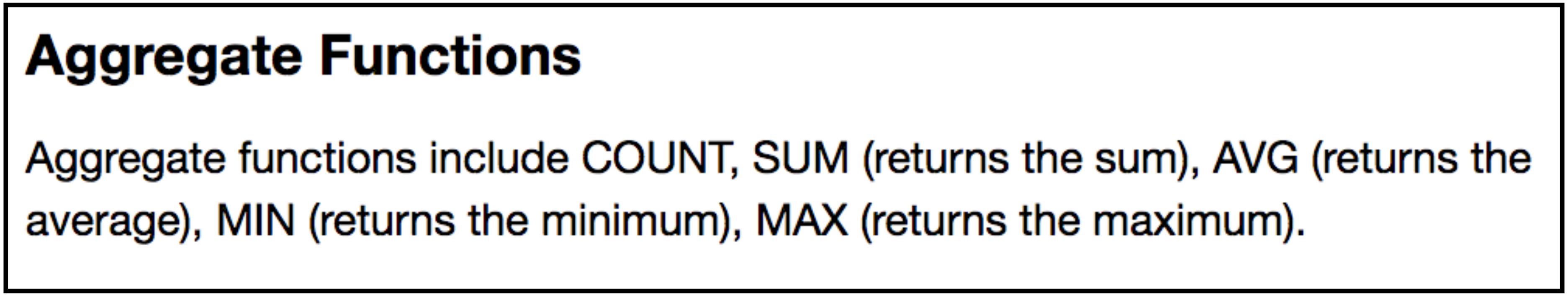


Use AS to make this more informative/readable:

```
query = '''
SELECT COUNT(id) AS "Total Trips by Registered Users"
FROM trips
WHERE sub_type = "Registered";
'''

run_query(query)
```

	Total Trips by Registered Users
0	1105192

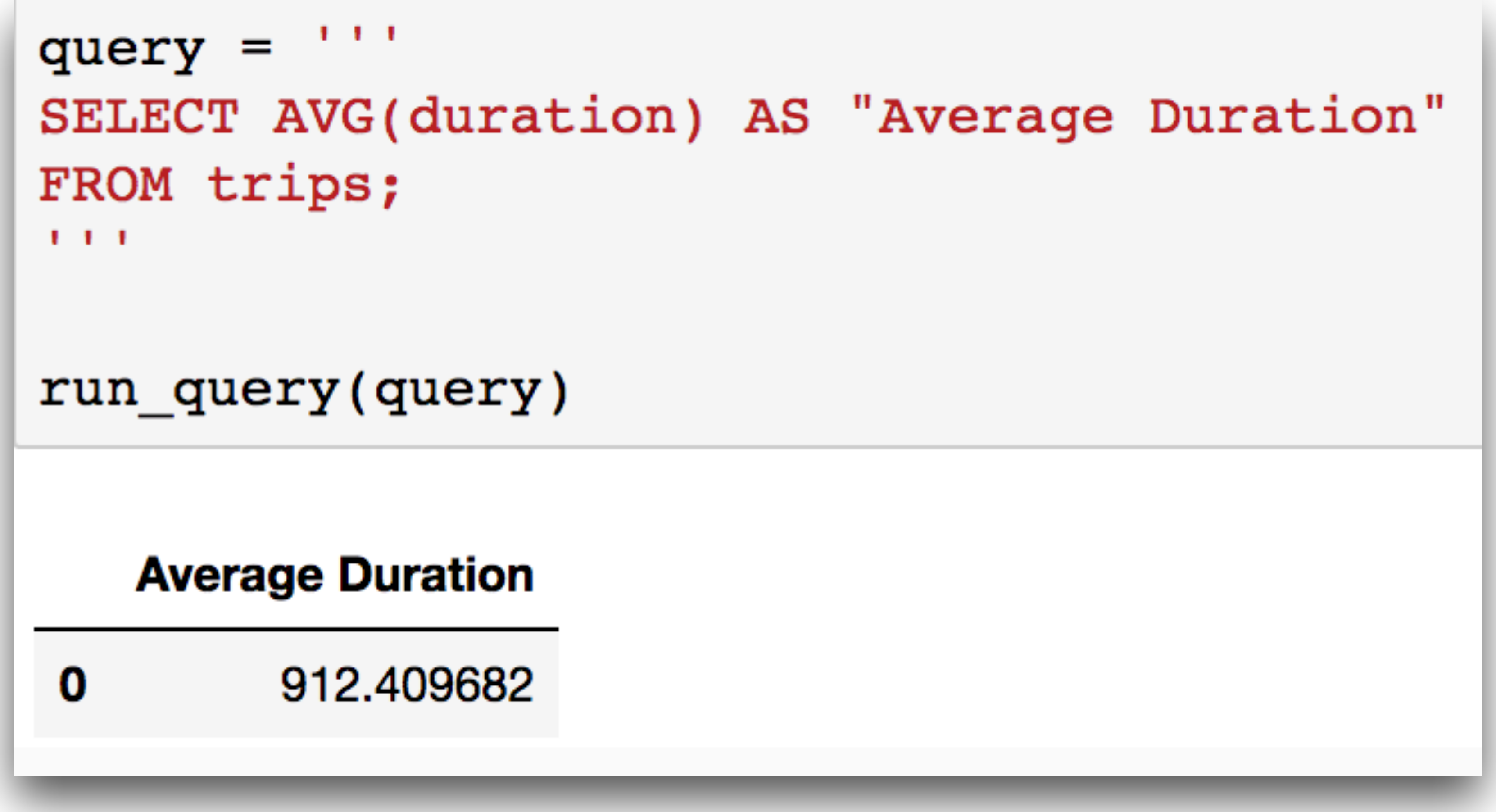
SQL: THE BASICS



Aggregate Functions

Aggregate functions include COUNT, SUM (returns the sum), AVG (returns the average), MIN (returns the minimum), MAX (returns the maximum).

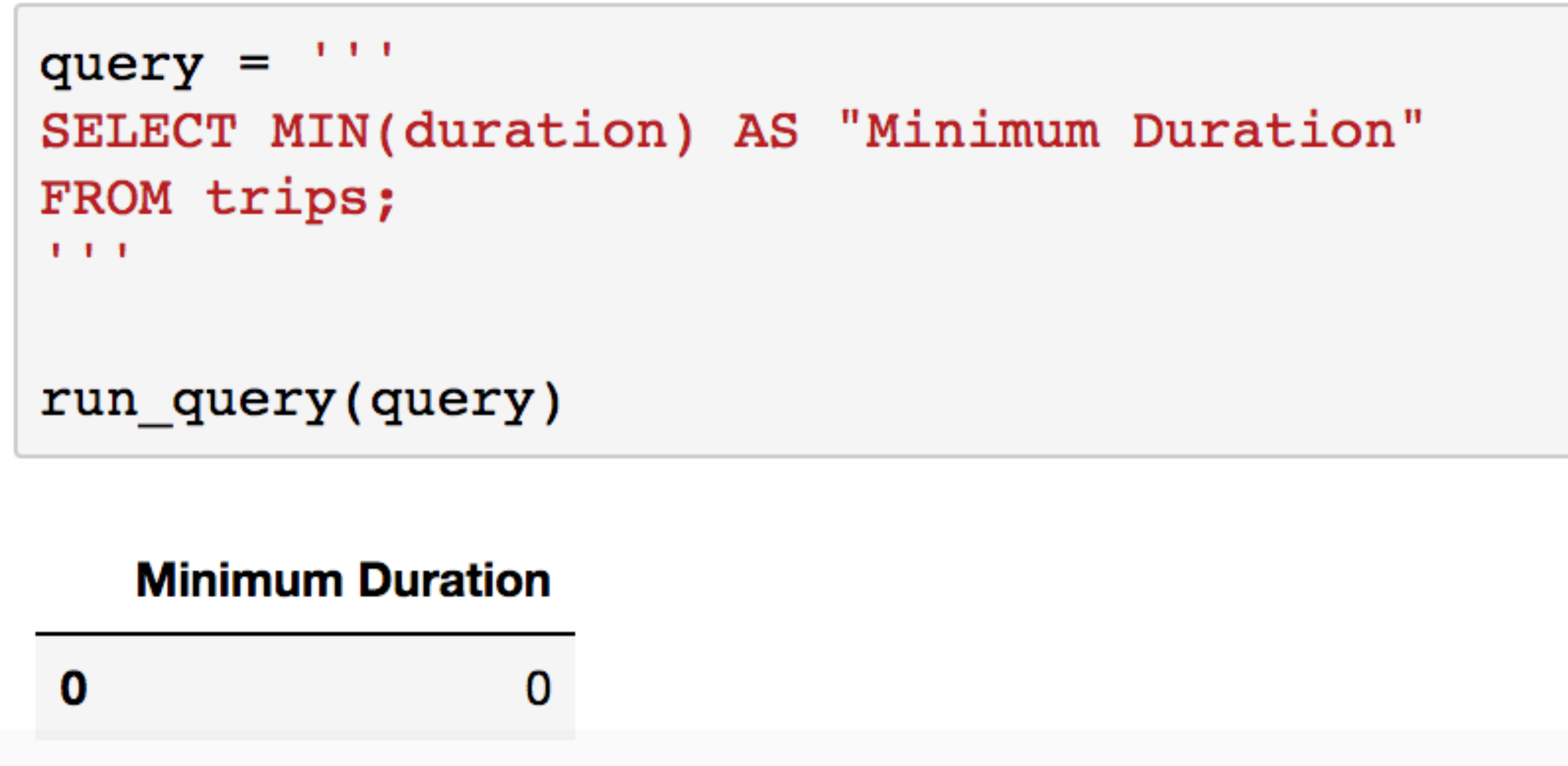
SQL: THE BASICS



```
query = '''
SELECT AVG(duration) AS "Average Duration"
FROM trips;
'''


run_query(query)
```

```
   Average Duration
0        912.409682
```

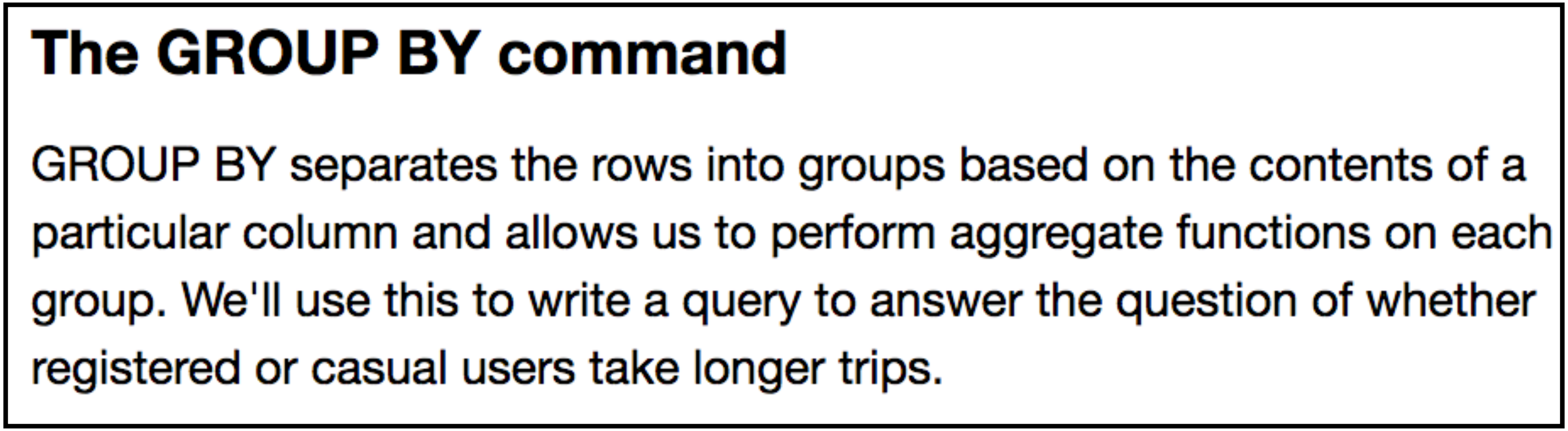
SQL: THE BASICS



```
query = '''
SELECT MIN(duration) AS "Minimum Duration"
FROM trips;
'''


run_query(query)
```

	Minimum Duration
0	0

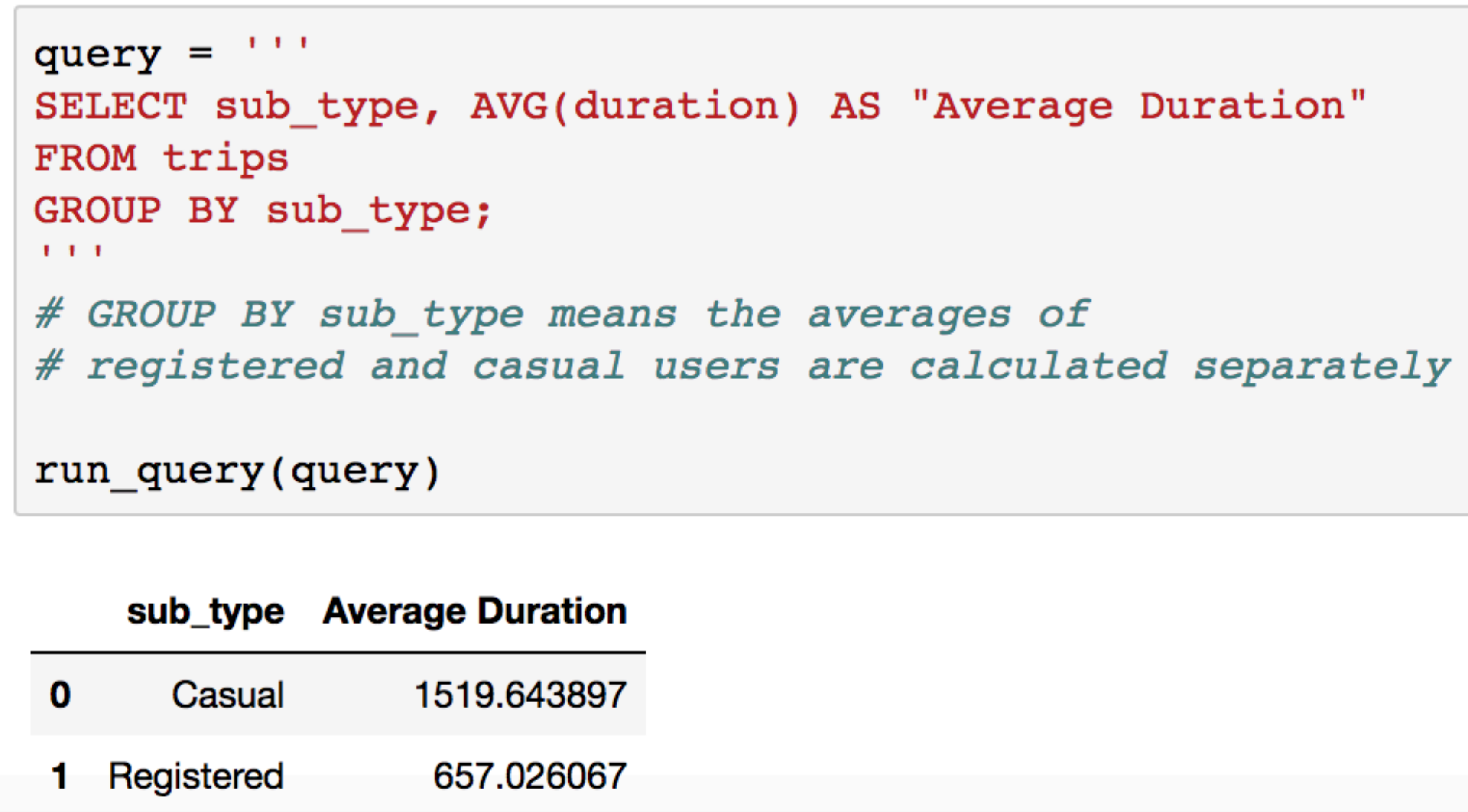
SQL: THE BASICS



The GROUP BY command

GROUP BY separates the rows into groups based on the contents of a particular column and allows us to perform aggregate functions on each group. We'll use this to write a query to answer the question of whether registered or casual users take longer trips.

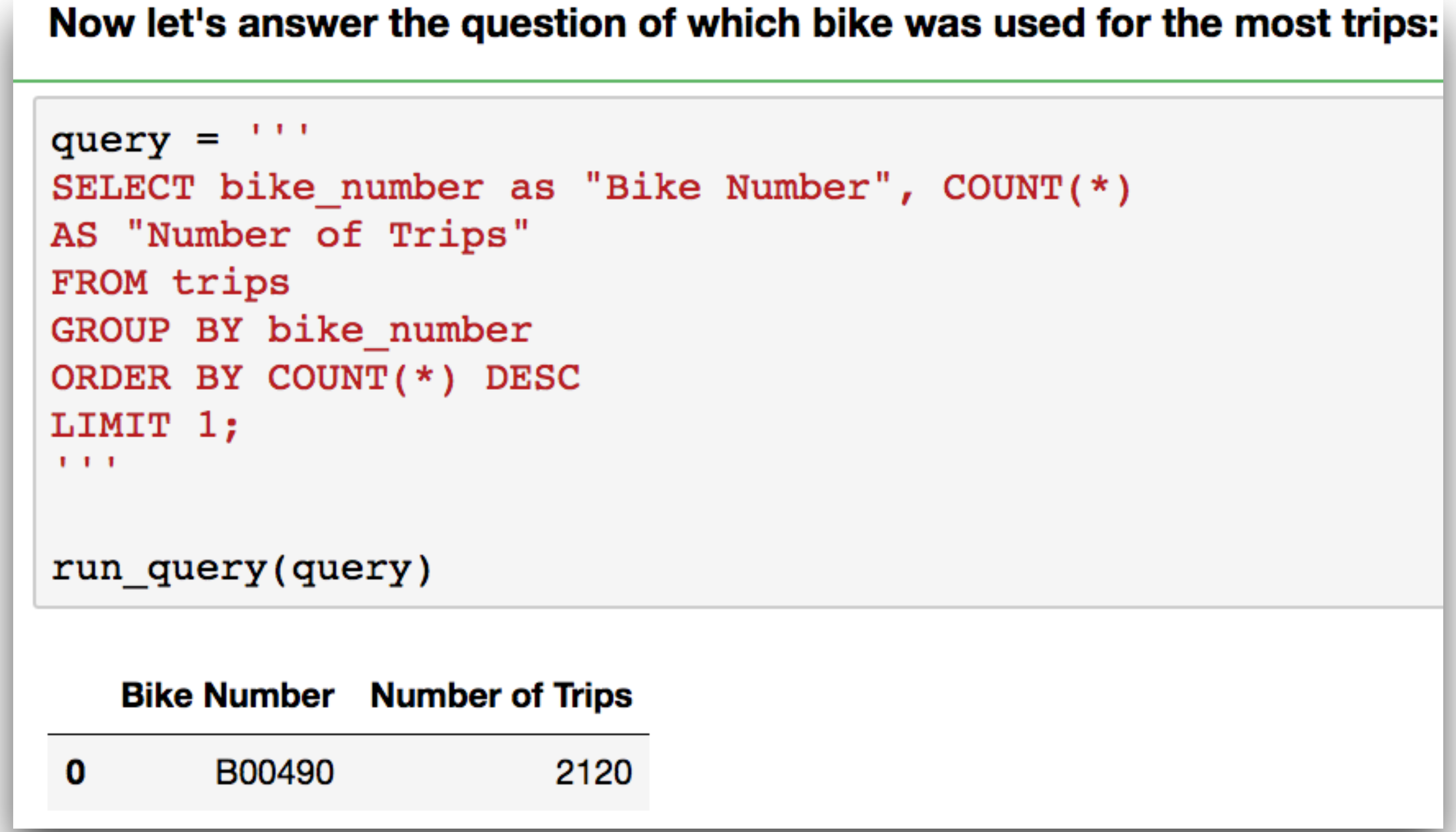
SQL: THE BASICS



```
query = '''
SELECT sub_type, AVG(duration) AS "Average Duration"
FROM trips
GROUP BY sub_type;
'''

# GROUP BY sub_type means the averages of
# registered and casual users are calculated separately

run_query(query)
```

	sub_type	Average Duration
0	Casual	1519.643897
1	Registered	657.026067

SQL: THE BASICS

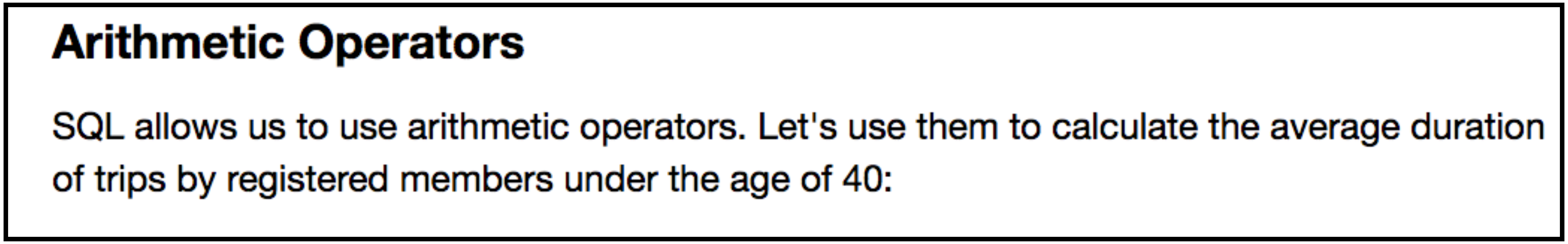


Now let's answer the question of which bike was used for the most trips:

```
query = '''
SELECT bike_number as "Bike Number", COUNT(*)
AS "Number of Trips"
FROM trips
GROUP BY bike_number
ORDER BY COUNT(*) DESC
LIMIT 1;
'''

run_query(query)
```

	Bike Number	Number of Trips
0	B00490	2120

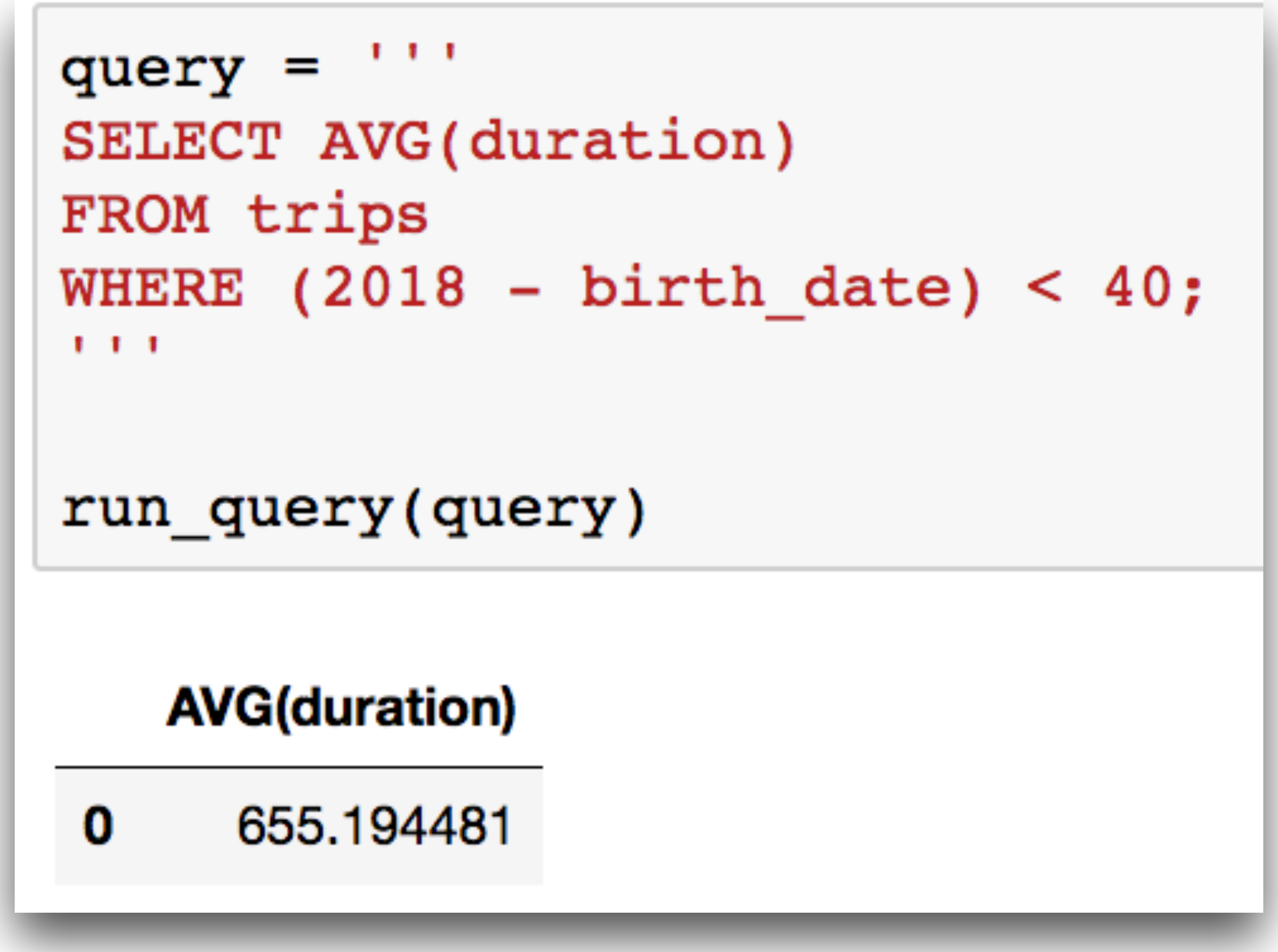
SQL: THE BASICS



Arithmetic Operators

SQL allows us to use arithmetic operators. Let's use them to calculate the average duration of trips by registered members under the age of 40:

SQL: THE BASICS



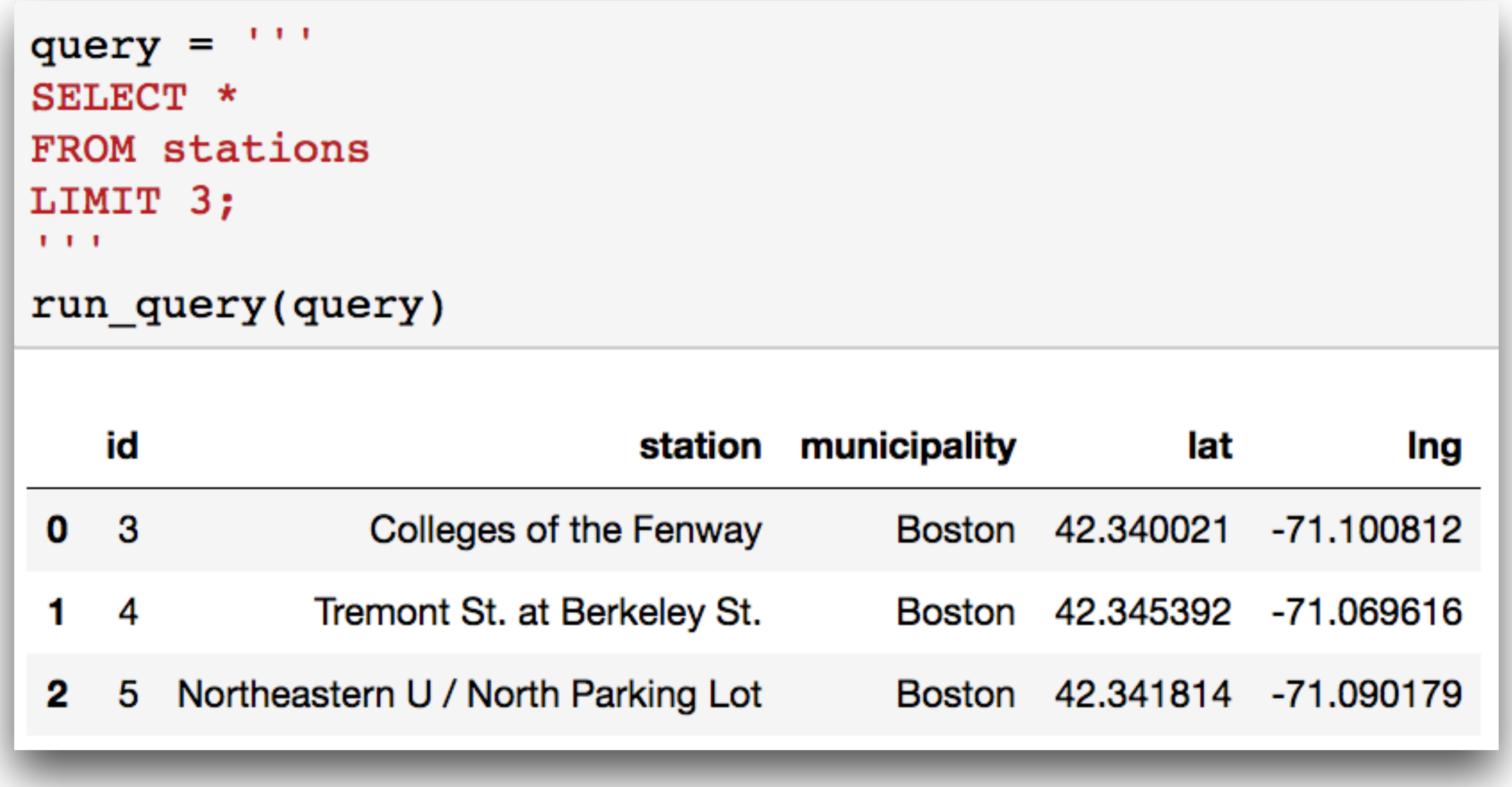
```
query = '''
SELECT AVG(duration)
FROM trips
WHERE (2018 - birth_date) < 40;
'''

run_query(query)
```

```
   AVG(duration)
0     655.194481
```

SQL: THE BASICS

- So far we’ve been looking at queries pulling data from the TRIPS table
- But as you might remember there’s also the STATIONS table
- The STATIONS table contains information about every station in the Hubway network
- It also includes an id column referenced by the TRIPS table
- So these tables can be combined to extract useful information
- Let’s have a look…

SQL: THE BASICS



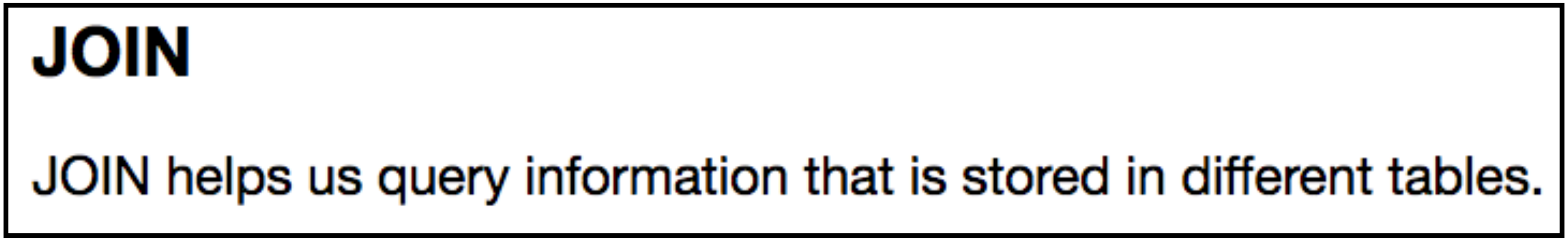
```
query = '''
SELECT *
FROM stations
LIMIT 3;
'''
run_query(query)
```

	id	station	municipality	lat	lng
0	3	Colleges of the Fenway	Boston	42.340021	-71.100812
1	4	Tremont St. at Berkeley St.	Boston	42.345392	-71.069616
2	5	Northeastern U / North Parking Lot	Boston	42.341814	-71.090179

ID is a unique identifier for each station, corresponding to the start_station and end_station columns in the TRIPS table

SQL: THE BASICS

- **Let’s say we want to know which station is the most popular starting point**
- For that we need to combine information from both the TRIPS and STATIONS tables
- We will use the JOIN command



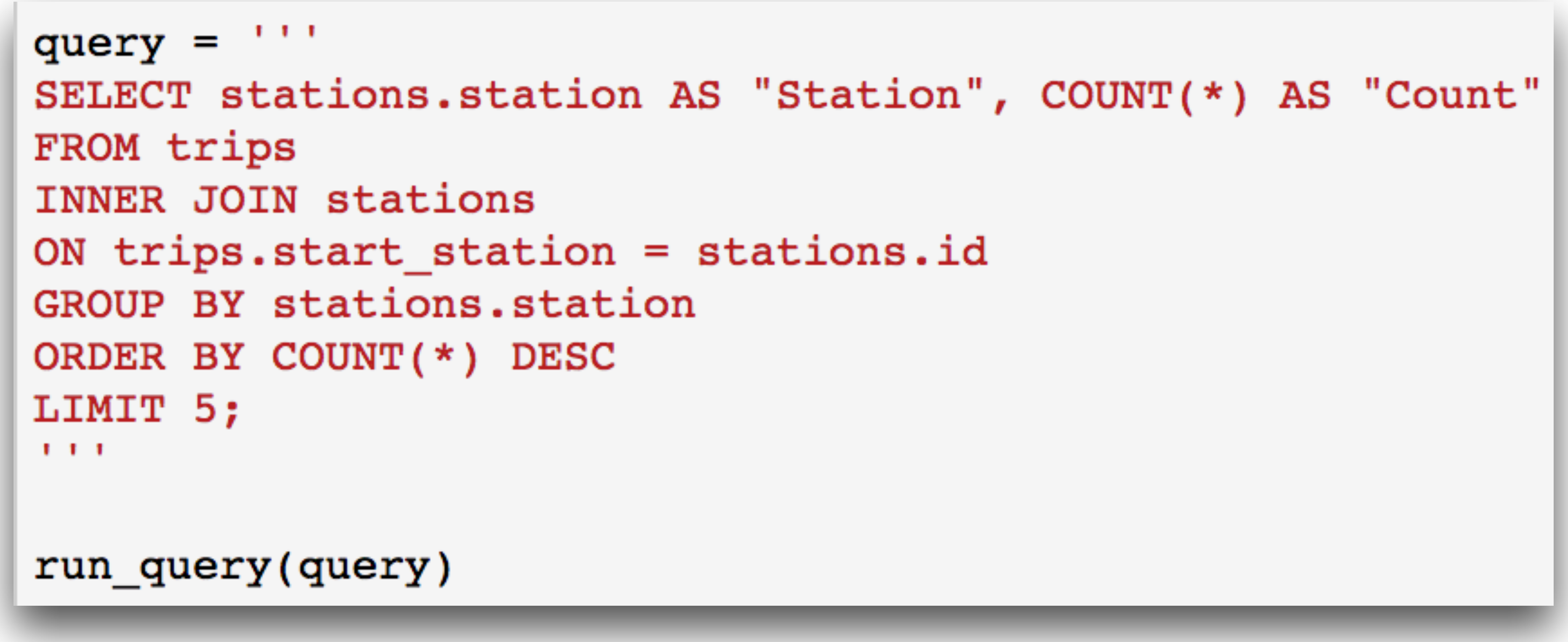
JOIN

JOIN helps us query information that is stored in different tables.

SQL: THE BASICS

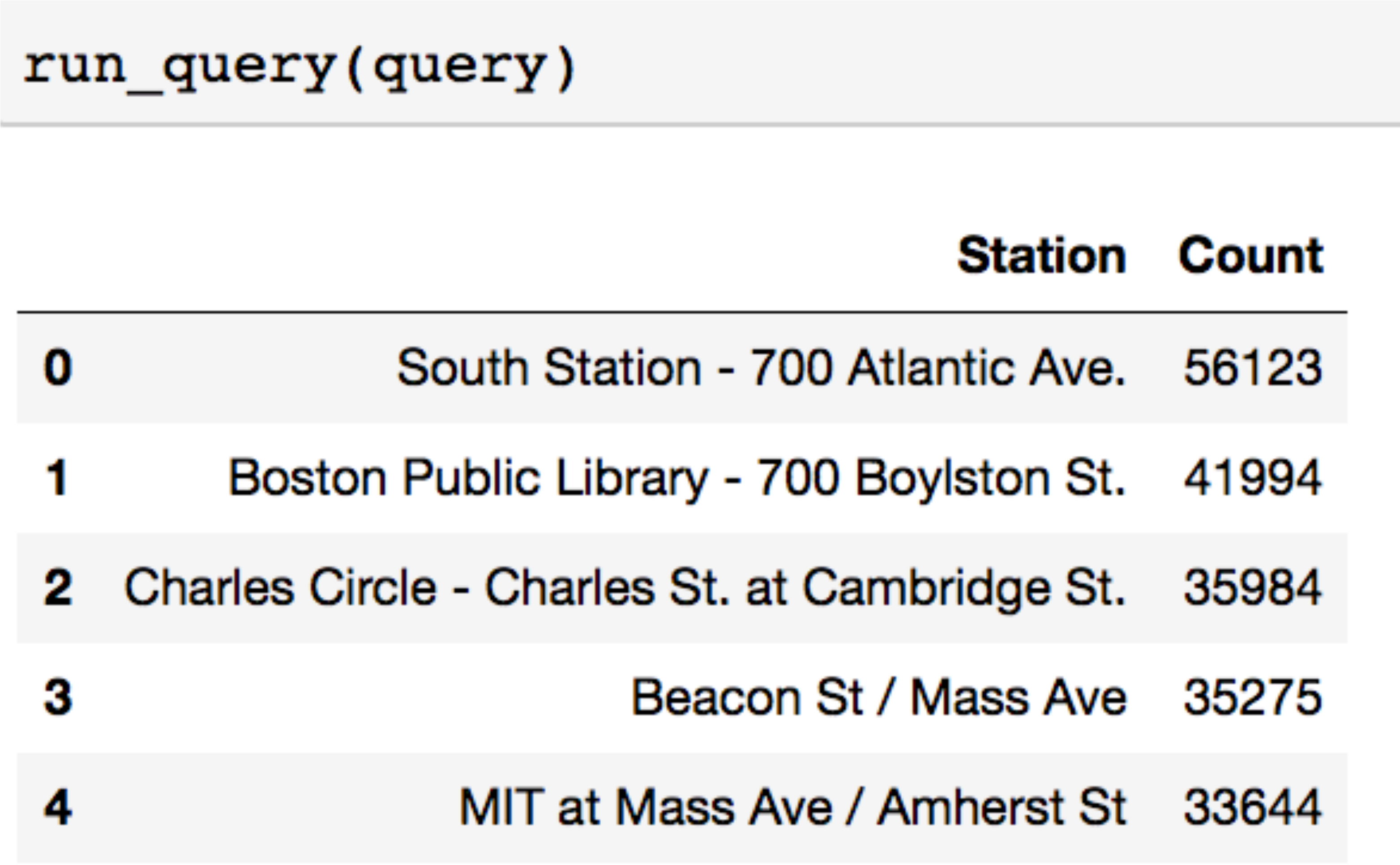
- We will use SELECT to return the station column from the stations table using the **table.column** syntax, i.e. **stations.station** in our case
- We also return the COUNT of the number of rows from the trips table
- To tell the database how the stations and trips tables are connected, we'll use JOIN and ON.
- JOIN specifies which tables should be connected
- ON specifies which columns in each table are related
- INNER JOIN means rows will only be returned where there is a much in the columns specified by ON
- Tables are connected ON trips.start_station = stations.id
- Then we group by the station column so that COUNT will give the number of trips for each station separately
- Finally we ORDER BY descending order

SQL: THE BASICS



```
query = '''
SELECT stations.station AS "Station", COUNT(*) AS "Count"
FROM trips
INNER JOIN stations
ON trips.start_station = stations.id
GROUP BY stations.station
ORDER BY COUNT(*) DESC
LIMIT 5;
'''


run_query(query)
```

SQL: THE BASICS

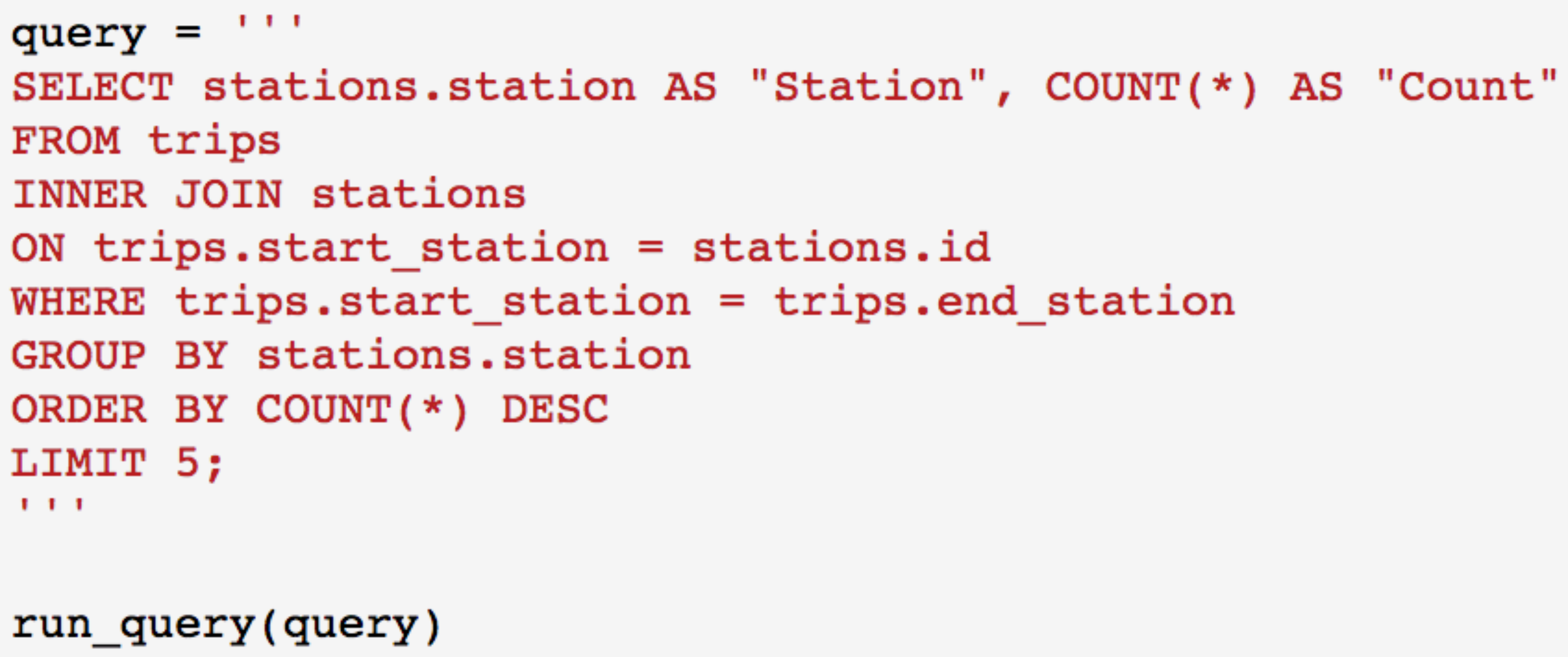


```
run_query(query)
```

	Station	Count
0	South Station - 700 Atlantic Ave.	56123
1	Boston Public Library - 700 Boylston St.	41994
2	Charles Circle - Charles St. at Cambridge St.	35984
3	Beacon St / Mass Ave	35275
4	MIT at Mass Ave / Amherst St	33644

SQL: THE BASICS

- **Let’s slightly expand this query to see which are the most popular round-trip stations:**



```
query = '''
SELECT stations.station AS "Station", COUNT(*) AS "Count"
FROM trips
INNER JOIN stations
ON trips.start_station = stations.id
WHERE trips.start_station = trips.end_station
GROUP BY stations.station
ORDER BY COUNT(*) DESC
LIMIT 5;
'''


run_query(query)
```

EXERCISES/TASKS

- Code up the queries we just learned in Jupyter and reproduce the results
- How many trips lasted more than half an hour? (this induces extra charges)
- Which bike was used for the least total time?
- Did registered or casual users take more round trips?
- Pick up any publicly available database and play with it!

PostgreSQL

- **PostgreSQL:** “the world’s most advanced open source relational database
- Active development for 30 years now!
- www.postgresql.org
- **Installation:** For MAC OS I strongly recommend using Postgress.app, see https://www.calhoun.io/how-to-install-postgresql-9-6-on-mac-os-x/
- For other systems, see https://www.dataquest.io/blog/sql-intermediate/ and https://www.systems.ethz.ch/sites/default/files/ex1a_postgresql_jupyter_setup.pdf (not tested…)

PostgreSQL

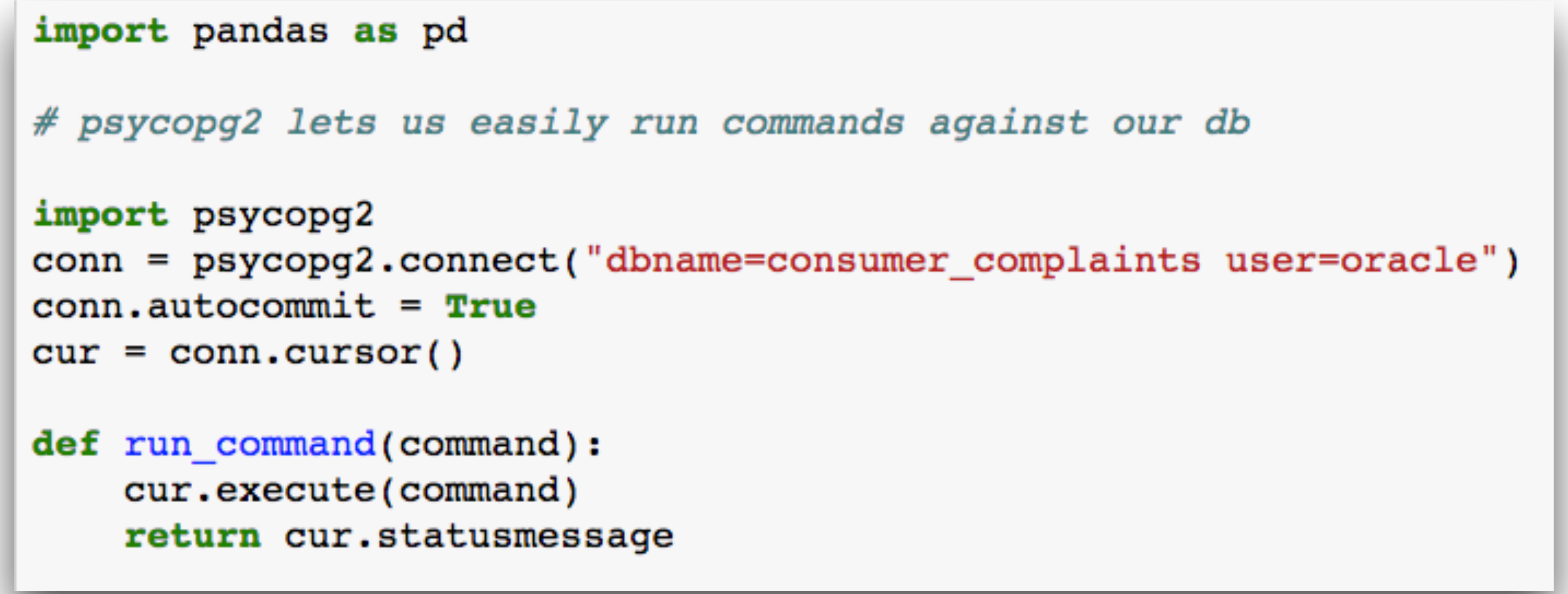
- First we need to **create new user, database, and tables**
- Follow the instructions in https://www.dataquest.io/blog/sql-intermediate/ to run psql, create a new user named ‘oracle’ (or another name of your preference) and a new database
- The new database contains consumer complaints
- **It has two tables:** one for bank account complaints and one for credit card complaints

PostgreSQL

- We need to populate these with actual data!
- We will use data from here https://data.world/dataquest/bank-and-credit-card-complaints
- Again, follow the instructions in https://www.dataquest.io/blog/sql-intermediate/ to load the data
- They are CSV files
    - They have identical fields: complaint_id, date_received, product, …, issue, consumer_complaint_narrative, etc.

PostgreSQL

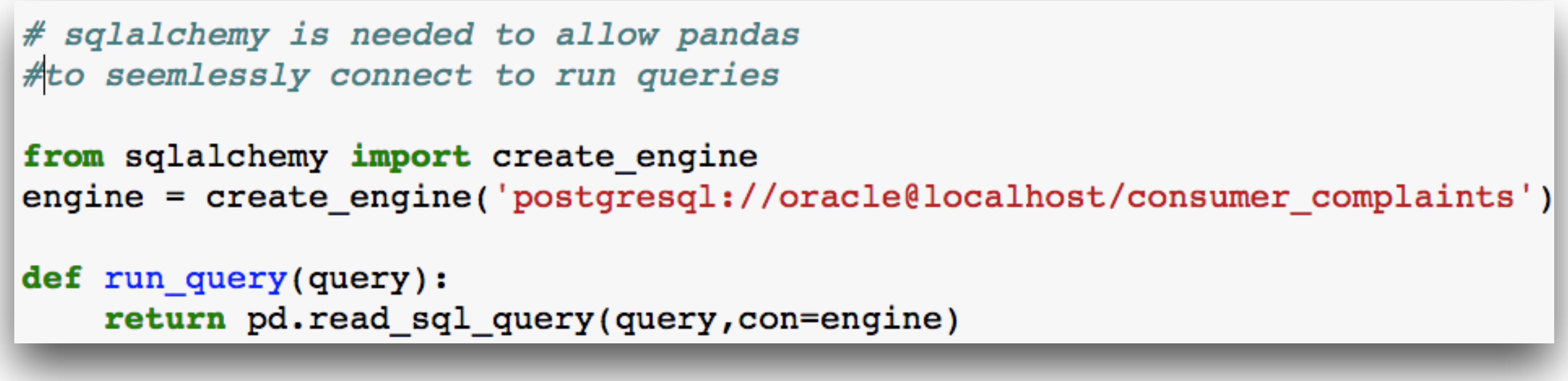
- Before having a look and playing with the data, we need to create two helper functions
- One to run queries and one to run commands



```
import pandas as pd

# psycopg2 lets us easily run commands against our db

import psycopg2
conn = psycopg2.connect("dbname=consumer_complaints user=oracle")
conn.autocommit = True
cur = conn.cursor()

def run_command(command):
    cur.execute(command)
    return cur.statusmessage
```

PostgreSQL

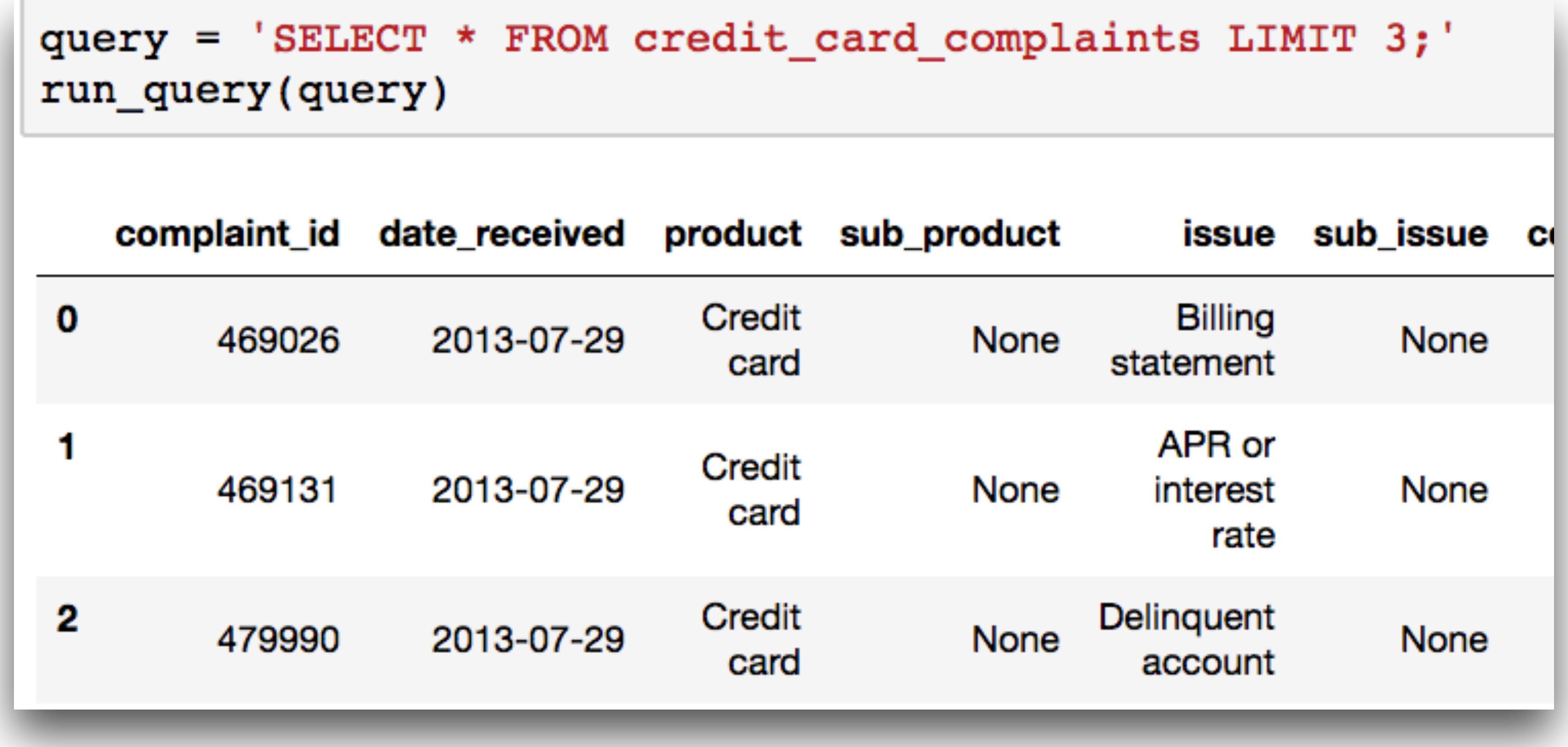
- Before having a look and playing with the data, we need to create two helper functions
- One to run queries and one to run commands



```
# sqlalchemy is needed to allow pandas
#to seemlessly connect to run queries

from sqlalchemy import create_engine
engine = create_engine('postgresql://oracle@localhost/consumer_complaints')

def run_query(query):
    return pd.read_sql_query(query,con=engine)
```

PostgreSQL

- OK, now let’s test everything works OK.
- First let’s see how the credit card complaints table looks like.



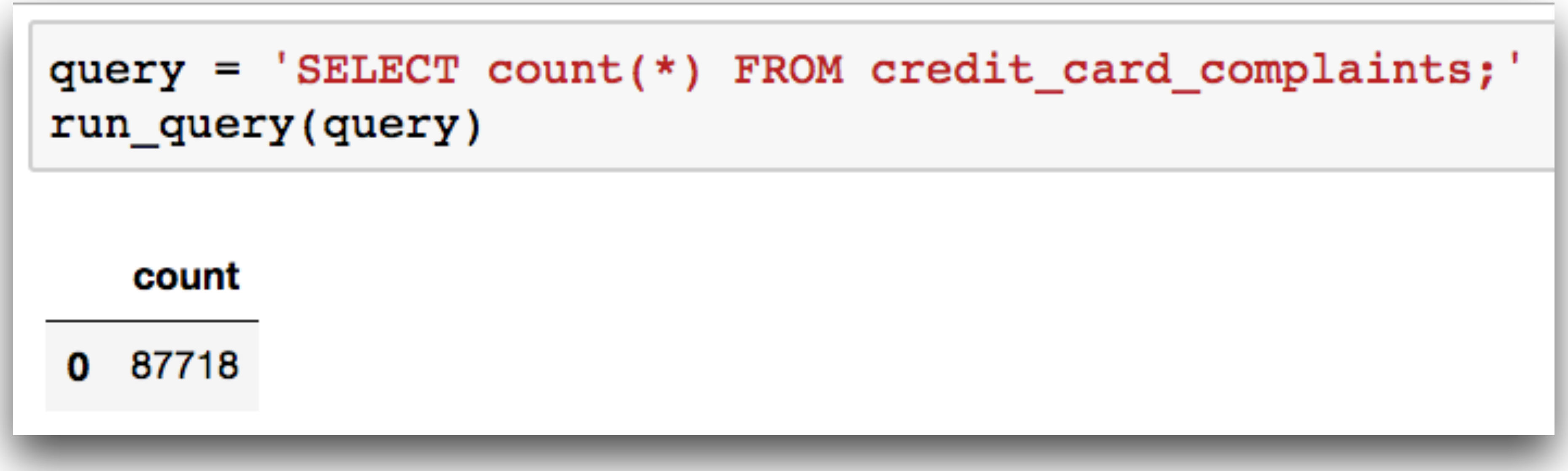
```
query = 'SELECT * FROM credit_card_complaints LIMIT 3;'
run_query(query)
```

	complaint_id	date_received	product	sub_product	issue	sub_issue	c
0	469026	2013-07-29	Credit card	None	Billing statement	None	
1	469131	2013-07-29	Credit card	None	APR or interest rate	None	
2	479990	2013-07-29	Credit card	None	Delinquent account	None	

PostgreSQL

- Then let's get the number of records using the COUNT function

- Works well! (try the bank account complaints table too)



```
query = 'SELECT count(*) FROM credit_card_complaints;'
run_query(query)
```

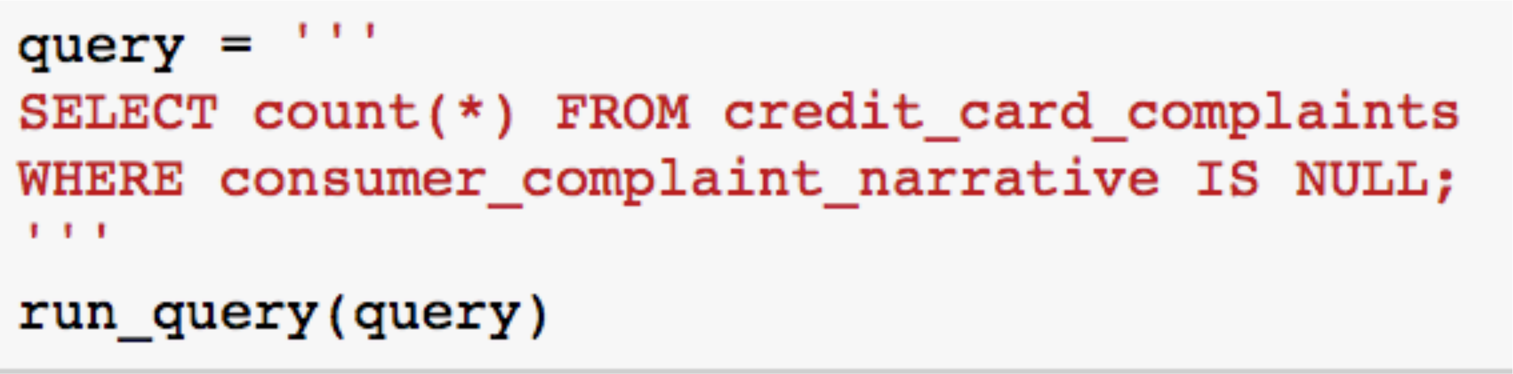
```
   count
0  87718
```

PostgreSQL

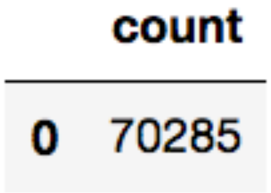
- How to deal with **NULL values**

- Let’s see how many records in each table have null values for the consumer complaint narrative field

- When comparing a column to null (no value), we cannot use arithmetic operators. Instead we use IS NULL / IS NOT NULL.



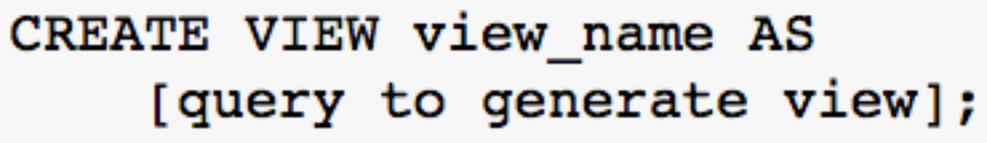
```
query = '''
SELECT count(*) FROM credit_card_complaints
WHERE consumer_complaint_narrative IS NULL;
'''
run_query(query)
```



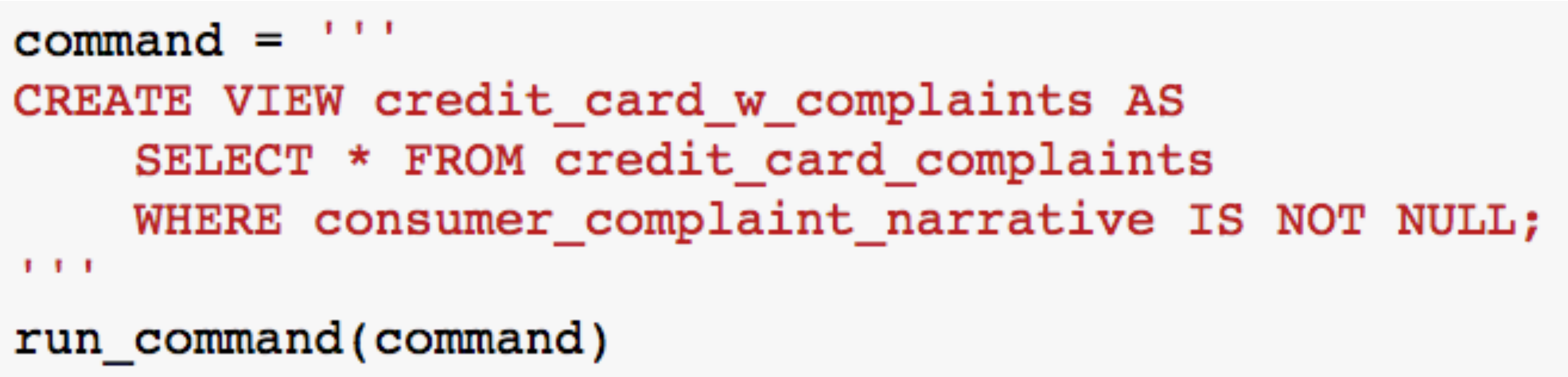
```
   count
0  70285
```

PostgreSQL: Views

- So we just saw a large amount of records had null values for the consumer complaint narrative field.
- Instead of having to filter on this field later, we’ll create a view with this subset only.
- Syntax is simple:



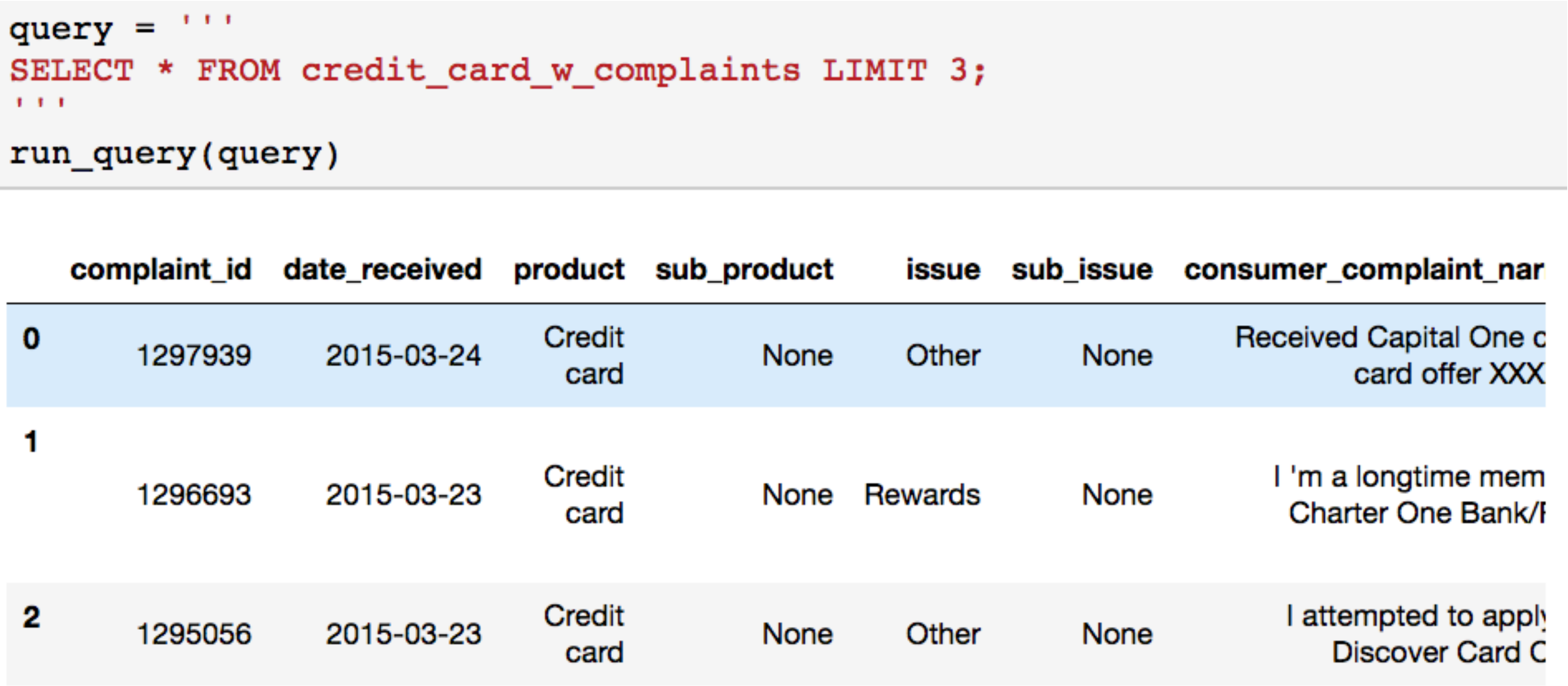
```
CREATE VIEW view_name AS
    [query to generate view];
```



```
command = '''
CREATE VIEW credit_card_w_complaints AS
    SELECT * FROM credit_card_complaints
    WHERE consumer_complaint_narrative IS NOT NULL;
'''
run_command(command)
```

PostgreSQL: Views

- Let’s have a look:



```
query = '''
SELECT * FROM credit_card_w_complaints LIMIT 3;
'''
run_query(query)
```

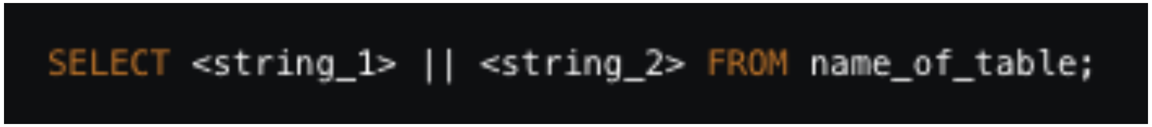
	complaint_id	date_received	product	sub_product	issue	sub_issue	consumer_complaint_nar
0	1297939	2015-03-24	Credit card	None	Other	None	Received Capital One c card offer XXX
1	1296693	2015-03-23	Credit card	None	Rewards	None	I 'm a longtime mem Charter One Bank/
2	1295056	2015-03-23	Credit card	None	Other	None	I attempted to appl Discover Card C

PostgreSQL: String Concatenation

- Extremely useful, combines two or more strings (text values) together to form a single string

- For example say we have a “month” field and a “year” field but we need to show “month-year” instead

- Syntax:

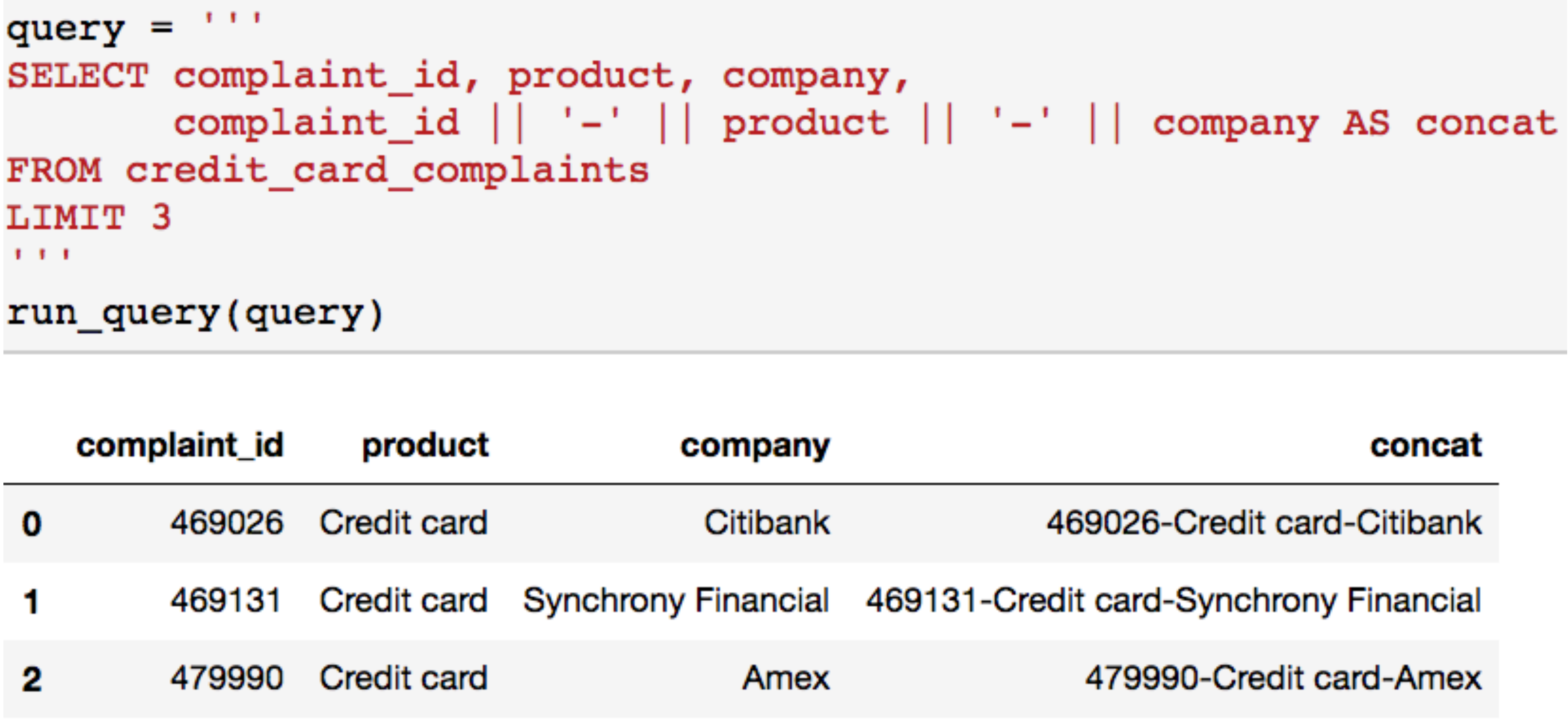


```
SELECT <string_1> || <string_2> FROM name_of_table;
```

- Let’s try it out with our credit card complaints table

- Let’s select *complaint_id, product, company*, and concatenate separated by a hyphen

PostgreSQL: String Concatenation

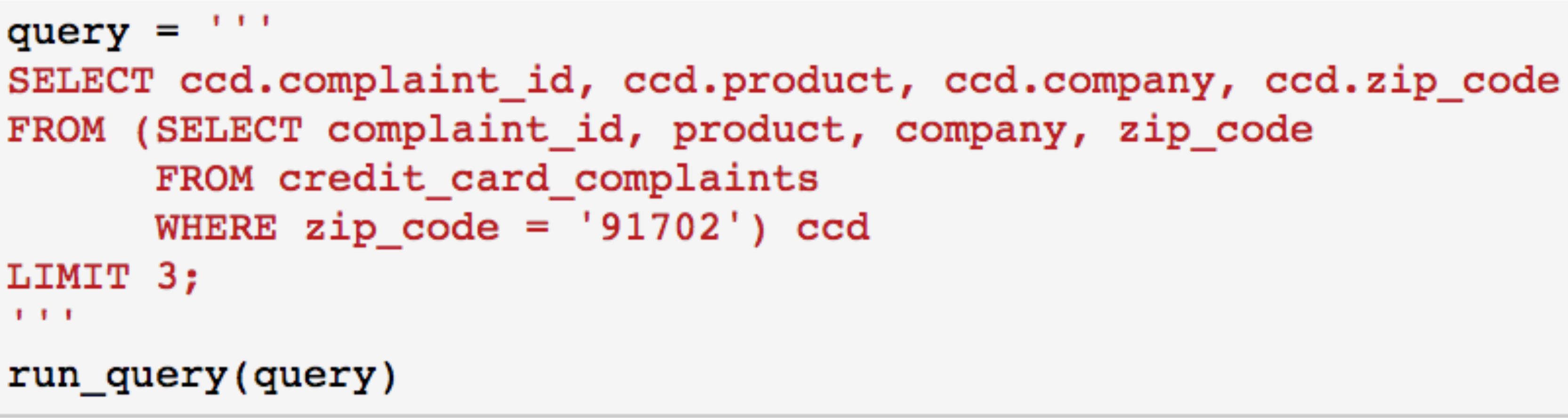


```
query = '''
SELECT complaint_id, product, company,
       complaint_id || '-' || product || '-' || company AS concat
FROM credit_card_complaints
LIMIT 3
'''
run_query(query)
```

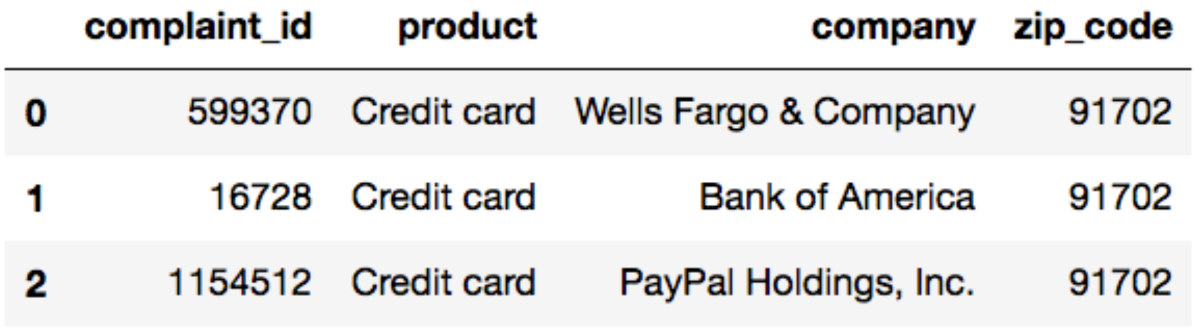
	complaint_id	product	company	concat
0	469026	Credit card	Citibank	469026-Credit card-Citibank
1	469131	Credit card	Synchrony Financial	469131-Credit card-Synchrony Financial
2	479990	Credit card	Amex	479990-Credit card-Amex

PostgreSQL: Subqueries

- Subqueries (“inline views”) create a mini view within a single query
- The best way to understand how they work is via an example:



```
query = '''
SELECT ccd.complaint_id, ccd.product, ccd.company, ccd.zip_code
FROM (SELECT complaint_id, product, company, zip_code
      FROM credit_card_complaints
      WHERE zip_code = '91702') ccd
LIMIT 3;
'''
run_query(query)
```



	complaint_id	product	company	zip_code
0	599370	Credit card	Wells Fargo & Company	91702
1	16728	Credit card	Bank of America	91702
2	1154512	Credit card	PayPal Holdings, Inc.	91702

TASKS

- ✦ Read about **UNION/UNION ALL**, and put them in action using different views of the banking data
- ✦ Same with **INTERSECT/EXCEPT**
- ✦ Explore subqueries, for example by reproducing the “Subqueries in action” examples in https://www.dataquest.io/blog/sql-intermediate/
- ✦ Go through the SQL/Pandas tutorial in https://www.dataquest.io/blog/python-pandas-databases/

REFERENCES

- Introduction to SQL https://www.stat.berkeley.edu/~spector/sql.pdf
- https://www.w3schools.com/sql/
- http://www.sql-tutorial.net/
- https://www.kaggle.com/learn/sql
- https://www.dataquest.io/blog/sql-basics/
- https://www.dataquest.io/blog/sql-intermediate/
- https://www.dataquest.io/blog/python-pandas-databases/