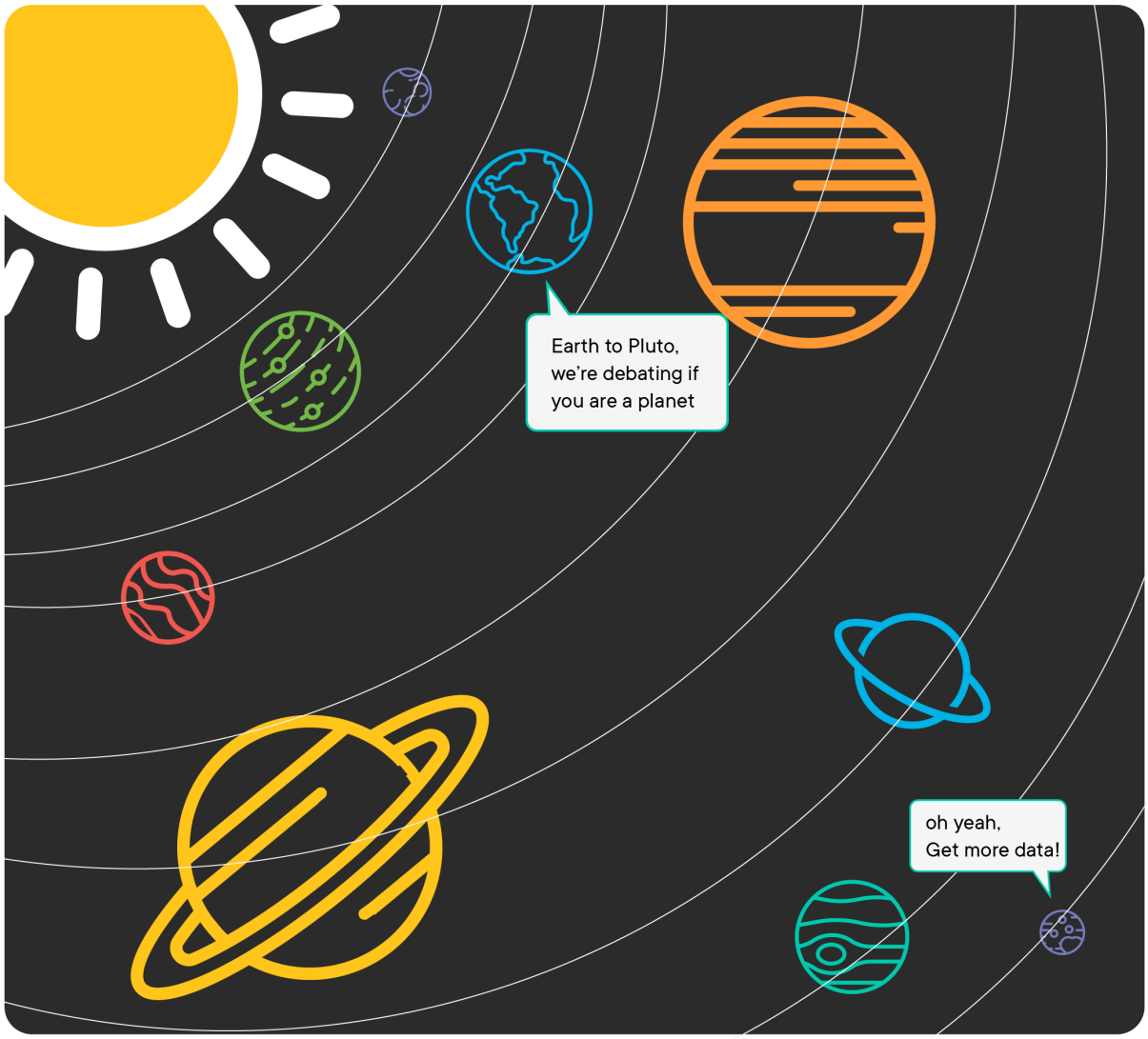
Filtering Data with SQL - Lab

Introduction

NASA wants to go to Mars! Before they build their rocket, NASA needs to track information about all of the planets in the Solar System. In this lab, you'll practice querying the database with various `SELECT` statements. This will include selecting different columns and implementing other SQL clauses like `WHERE` to return the data desired.



Objectives

You will practice the following:

- Retrieve a subset of records from a table using a `WHERE` clause
- Filter results using conditional operators such as `BETWEEN`, `IS NULL`, and `LIKE`
- Apply an aggregate function to the result of a filtered query

Connecting to the Database

To get started, import `sqlite3` as well as `pandas` for conveniently displaying results. Then, connect to the SQLite database located at `planets.db`.

```
import pandas as pd
import sqlite3
conn = sqlite3.connect('planets.db')
```

Database Schema

This database contains a single table, `planets`. This is the schema:

```
CREATE TABLE planets (
  id INTEGER PRIMARY KEY,
  name TEXT,
  color TEXT,
  num_of_moons INTEGER,
  mass REAL,
  rings BOOLEAN
);
```

The data looks something like this:

id	name	color	num_of_moons	mass	rings
1	Mercury	gray	0	0.55	FALSE
2	Venus	yellow	0	0.82	FALSE
3	Earth	blue	1	1.00	FALSE
4	Mars	red	2	0.11	FALSE
5	Jupiter	orange	67	317.90	FALSE
6	Saturn	hazel	62	95.19	TRUE
7	Uranus	light blue	27	14.54	TRUE
8	Neptune	dark blue	14	17.15	TRUE

SQL Queries

Write SQL queries for each of the statements below using the same pandas wrapping syntax from the previous lesson.

1. Select just the `name` and `color` of each planet

```
pd.read_sql("""
SELECT name, color
```

```
  FROM planets;
""", conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	name	color
0	Mercury	gray
1	Venus	yellow
2	Earth	blue
3	Mars	red
4	Jupiter	orange
5	Saturn	hazel
6	Uranus	light blue
7	Neptune	dark blue

2. Select all columns for each planet whose `num_of_moons` is 0

```
pd.read_sql("""
SELECT *
  FROM planets
 WHERE num_of_moons = 0;
""", conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}
```

```
.dataframe thead th {
    text-align: right;
}
```

</style>

	id	name	color	num_of_moons	mass	rings
0	1	Mercury	gray	0	0.55	0
1	2	Venus	yellow	0	0.82	0

3. Select the `name` and `mass` of each planet whose `name` has exactly 7 letters

```
pd.read_sql("""
SELECT name, mass
  FROM planets
 WHERE length(name) = 7;
""", conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	name	mass
0	Mercury	0.55
1	Jupiter	317.90
2	Neptune	17.15

4. Select all columns for each planet whose `mass` is greater than 1.00

```
pd.read_sql("""
SELECT *
```

```
  FROM planets
 WHERE mass > 1.00;
""", conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	id	name	color	num_of_moons	mass	rings
0	5	Jupiter	orange	68	317.90	0
1	6	Saturn	hazel	62	95.19	1
2	7	Uranus	light blue	27	14.54	1
3	8	Neptune	dark blue	14	17.15	1

5. Select the `name` and `mass` of each planet whose `mass` is less than or equal to 1.00

```
pd.read_sql("""
SELECT name, mass
  FROM planets
 WHERE mass <= 1.00;
""", conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	name	mass
0	Mercury	0.55
1	Venus	0.82
2	Earth	1.00
3	Mars	0.11

6. Select the `name` and `mass` of each planet whose `mass` is between 0 and 50

```
# Technically this would also work if you said
# 0 <= mass AND mass <= 50
# but using BETWEEN is simpler and more efficient

pd.read_sql("""
SELECT name, mass
  FROM planets
 WHERE mass BETWEEN 0 AND 50;
""", conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	name	mass
0	Mercury	0.55
1	Venus	0.82
2	Earth	1.00
3	Mars	0.11

	name	mass
4	Uranus	14.54
5	Neptune	17.15

7. Select all columns for planets that have at least one moon and a `mass` less than 1.00

Hint: You can use `AND` to chain together two conditions in SQL, similar to `and` in Python

```
# "at least one moon", i.e. "1 or more moons",
# i.e. num_of_moons >= 1

pd.read_sql("""
SELECT *
  FROM planets
 WHERE num_of_moons >= 1
   AND mass < 1.00;
""", conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	id	name	color	num_of_moons	mass	rings
0	4	Mars	red	2	0.11	0

8. Select the `name` and `color` of planets that have a `color` containing the string "blue"

```
# For this particular dataset, we also could have done:
# WHERE color = 'blue'
#    OR color = 'light blue'
#    OR color = 'dark blue'
```

```
# but that would not be flexible if the underlying data ever changed

# Also, the string of "%blue" would work for this dataset
# since the available colors all end with "blue", but putting
# % on either side is a better match for the prompt

pd.read_sql("""
SELECT name, color
  FROM planets
 WHERE color LIKE "%blue%";
""", conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	name	color
0	Earth	blue
1	Uranus	light blue
2	Neptune	dark blue

9. Select the count of planets that don't have rings as `planets_without_rings`

Note: even though the schema states that `rings` is a `BOOLEAN` and the example table shows values `TRUE` and `FALSE`, SQLite does not actually support booleans natively. From the documentation:

> SQLite does not have a separate Boolean storage class. Instead, Boolean values are stored as integers 0 (false) and 1 (true).

```
# We could also say
# WHERE NOT rings

pd.read_sql("""
```

```
  SELECT COUNT(*) AS planets_without_rings
    FROM planets
   WHERE rings = 0;
  """, conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
  .dataframe tbody tr th {
      vertical-align: top;
  }

  .dataframe thead th {
      text-align: right;
  }
```

</style>

	planets_without_rings
0	5

10. Select the name of all planets, along with a value `has_rings` that returns "Yes" if the planet does have rings, and "No" if it does not

```
  pd.read_sql("""
  SELECT name,
         CASE rings
         WHEN 1 THEN "Yes"
         WHEN 0 THEN "No"
         END AS has_rings
    FROM planets;
  """, conn)
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
  .dataframe tbody tr th {
      vertical-align: top;
  }

  .dataframe thead th {
      text-align: right;
  }
```

</style>

	name	has_rings
0	Mercury	No
1	Venus	No
2	Earth	No
3	Mars	No
4	Jupiter	No
5	Saturn	Yes
6	Uranus	Yes
7	Neptune	Yes

Summary

Congratulations! NASA is one step closer to embarking upon its mission to Mars. In this lab, You practiced writing `SELECT` statements that query a single table to get specific information. You also used other clauses and specified column names to cherry-pick the data we wanted to retrieve.

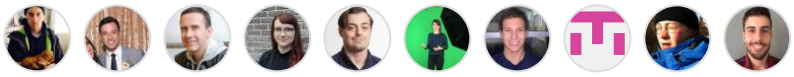
Releases

No releases published

Packages

No packages published

Contributors 10



Languages

- **Jupyter Notebook** 50.9%
- **Python** 49.1%