Filtering and Ordering

Goals:

- Write SQL queries to filter and order results
- Order the results of your queries by using ORDER BY (ASC & DESC)
- Limit the number of records returned by a query using LIMIT
- Filter results using BETWEEN and IS NULL

Creating our Database

In [1]:
```
import pandas as pd
import sqlite3

conn = sqlite3.connect("pets_database.db")
cur = conn.cursor()
```

In [2]:
```
# checking the sql database
cur = cur.execute("""SELECT sql FROM sqlite_master;""")
pets_database = cur.fetchall()
pets_database
```

Out[2]:
```
[('CREATE TABLE cats ( id INTEGER PRIMARY KEY, name TEXT, age INTEGER, breed TEXT,
owner_id INTEGER )',),
 ('CREATE TABLE dogs (\n    id INTEGER PRIMARY KEY,\n    name TEXT\n)',)]
```

In [3]:
```
# select all data from cats data
all_cats = pd.read_sql("""SELECT * FROM cats;""", conn)
all_cats
```

Out[3]:

	id	name	age	breed	owner_id
0	1	Maru	3.0	Scottish Fold	1.0
1	2	Hana	1.0	Tabby	1.0
2	3	Lil' Bub	5.0	American Shorthair	NaN
3	4	Moe	10.0	Tabby	NaN
4	5	Patches	2.0	Calico	NaN
5	6	None	NaN	Tabby	NaN

ORDER BY clause

The ORDER BY clause in SQL is used to sort the results of a query based on one or more columns. It allows you to arrange the rows in either ascending (default) or descending order.

syntax

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column_name [ASC|DESC];
```

- `ASC` : Sorts the result in ascending order (default).
- `DESC` : Sorts the result in descending order.

Use Cases:

1. Displaying results in a specific order (e.g., top performers, recent dates).
2. Organizing data for easier analysis.
3. Providing input for further operations that require sorted data.

```
cur.execute('''SELECT column_name FROM table_name ORDER BY column_name
ASC|DESC;''').fetchall()
```

In [4]:
```
#  select all cats order by age
cats_age = pd.read_sql("""SELECT * FROM cats ORDER BY age;""", conn)
cats_age
```

Out[4]:

	id	name	age	breed	owner_id
0	6	None	NaN	Tabby	NaN
1	2	Hana	1.0	Tabby	1.0
2	5	Patches	2.0	Calico	NaN
3	1	Maru	3.0	Scottish Fold	1.0
4	3	Lil' Bub	5.0	American Shorthair	NaN
5	4	Moe	10.0	Tabby	NaN

In [5]:
```
#  select all of our cats and sort them by age in descending order
cats_age_desc = pd.read_sql("""SELECT * FROM cats ORDER BY age DESC;""", conn)
cats_age_desc
```

Out[5]:

	id	name	age	breed	owner_id
0	4	Moe	10.0	Tabby	NaN
1	3	Lil' Bub	5.0	American Shorthair	NaN
2	1	Maru	3.0	Scottish Fold	1.0
3	5	Patches	2.0	Calico	NaN
4	2	Hana	1.0	Tabby	1.0
5	6	None	NaN	Tabby	NaN

LIMIT clause

The `LIMIT` clause in SQL is used to restrict the number of rows returned by a query. It's especially useful when dealing with large datasets where you only need a subset of the results.

```
SELECT column1, column2, ...
FROM table_name
[WHERE condition]
ORDER BY column_name [ASC|DESC]
LIMIT number_of_rows;
```

`LIMIT` : Specifies the maximum number of rows to return

Use Cases:

1. *Pagination*: Retrieve a specific subset of rows for a page in an application.
2. *Top N Analysis*: Fetch the top-performing rows based on a metric (e.g., top 10 sales).
3. *Preview Data*: View a small sample of a large dataset.

```
cur.execute('''SELECT * FROM cats ORDER BY age DESC LIMIT
1;''').fetchone()
```

In [6]:
```
# Let's get the two oldest cats
oldest_cats = pd.read_sql("""SELECT * FROM cats ORDER  BY age DESC LIMIT 2;""", con
oldest_cats
```

Out[6]:

	id	name	age	breed	owner_id
0	4	Moe	10	Tabby	None
1	3	Lil' Bub	5	American Shorthair	None

BETWEEN clause

The BETWEEN clause in SQL is used to filter the rows where a column's value falls within a specified range. It is inclusive, meaning it includes both the starting and ending values of the range.

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

- value1: The lower bound of the range.
- value2: The upper bound of the range.

Key Points:

1. Inclusive: The range includes both boundary values.
2. Supports Numbers, Dates, and Text: You can use BETWEEN for numeric, date, or string data types.
3. Readability: Simplifies filtering ranges compared to using multiple conditions:

```
WHERE column_name >= value1 AND column_name <= value2
```

In [7]:
```
# find all records of cats between ages 1 and 3
cats_age_bwn_1_and_3 = pd.read_sql("""SELECT name FROM cats WHERE age BETWEEN 1 AND
cats_age_bwn_1_and_3
```

Out[7]:

	name
0	Maru
1	Hana
2	Patches

NULL

In SQL, `NULL` represents a missing, unknown, or undefined value in a column. It is not equivalent to zero, an empty string, or any other value—it is simply the absence of a value.

Key Concepts:

NULL Value:

- `NULL` indicates missing or undefined data.
- A column can contain `NULL` if no value is provided during data entry and the column allows `NULL`.

Not Comparable:

- `NULL` cannot be compared using standard operators like `=` or `!=`.
- Special keywords (`IS NULL or IS NOT NULL`) are required to handle NULL values.

In [8]:
```
# select cats where the name field is null
cats_null_name = pd.read_sql("""SELECT * FROM cats WHERE name IS NULL;""", conn)
cats_null_name
```

Out[8]:

	id	name	age	breed	owner_id
0	6	None	None	Tabby	None

COUNT clause

The `COUNT` function in SQL is an aggregate function used to calculate the number of rows in a table or the number of non-`NULL` values in a specific column. It is commonly used for summarizing data.

syntax

```
SELECT COUNT(column_name)
FROM table_name
[WHERE condition];
```

- `COUNT(column_name)` : Counts only non-`NULL` values in the column.
- `COUNT(*):` Counts all rows, including those with `NULL` values.

Use Cases:

- Basic Counts: Count all rows, filtered rows, or unique entries in a column.

*Group Counts: Use `COUNT` with `GROUP BY` to calculate counts for each group

```
SELECT color, COUNT(*)
FROM planets
GROUP BY color;
```

- `COUNT(*)` : Counts all rows.
- `COUNT(column_name)` : Counts non-NULL values in a column.
- `COUNT(DISTINCT column_name)` : Counts unique non-NULL values.

In [9]:
```
# count the number of cats who have an owner_id of 1
owner_id_1 = pd.read_sql("""SELECT COUNT(owner_id) FROM cats  WHERE  owner_id = 1;"
owner_id_1
```

Out[9]:

	COUNT(owner_id)
0	2

GROUP BY

The `GROUP BY` clause in SQL is used to arrange rows with the same values in specified columns into groups. It is often used in combination with aggregate functions (like `COUNT`,

`SUM`, `AVG`, `MAX`, `MIN`) to perform calculations for each group.

```sql
SELECT column1, aggregate_function(column2)
FROM table_name
[WHERE condition]
GROUP BY column1;

SELECT color, COUNT(*)
FROM planets
GROUP BY color;

SELECT color, rings, COUNT(*)
FROM planets
GROUP BY color, rings;

SELECT color, SUM(num_of_moons) AS total_moons
FROM planets
GROUP BY color;

SELECT color, COUNT(*) AS planet_count
FROM planets
GROUP BY color
HAVING COUNT(*) > 1;
```

- `column1`: The column by which rows are grouped.
- `aggregate_function`: Performs calculations (e.g., `COUNT`, `SUM`, `AVG`) on each group.

Key Differences Between WHERE and HAVING:

- `WHERE`: Filters rows before grouping.
- `HAVING`: Filters groups after aggregation.

Use Cases:

- **Summarizing Data**: Calculate totals, averages, or counts for categories.
- **Categorical Analysis**: Identify trends within grouped data.
- **Post-Aggregate Filtering**: Apply conditions on aggregate results.

In [10]:
```
# group cats by breed
breeds = pd.read_sql("""SELECT breed, owner_id,  COUNT(breed) FROM cats GROUP BY br
breeds
```

Out[10]:

	breed	owner_id	COUNT(breed)
0	American Shorthair	NaN	1
1	Calico	NaN	1
2	Scottish Fold	1.0	1
3	Tabby	NaN	2
4	Tabby	1.0	1

note on SELECT clause

When using multiple tables, specify the table name.

syntax

```
SELECT table1.column_name, table2.column_name
FROM table1, table2
JOIN table2 ON table1.id = table2.id;
```

In [11]:
```
# SQL statement to create the dogs table
create_table_query = """
CREATE TABLE IF NOT EXISTS dogs (
    id INTEGER PRIMARY KEY,
    name TEXT
);
"""
cur.execute(create_table_query)
conn.commit()  # Commit the table creation""
```

In [12]:
```
insert_query = """
INSERT INTO dogs (name) VALUES (?);
"""
names = ["Clifford", "Tana", "Tom"]

# Loop through each name and execute the insert query
for name in names:
    cur.execute(insert_query, (name,))

conn.commit()  # Commit changes
```

In [13]:
```
dog_name = pd.read_sql("""SELECT dogs.name FROM dogs;""",conn)
dog_name
```

Out[13]:

	name
0	Clifford
1	Clifford
2	Tana
3	Tom
4	Clifford
5	Tana
6	Tom
7	Clifford
8	Tana
9	Tom

Bonus , Calender 2025

In [14]:

```
from calendar import TextCalendar  # Import the TextCalendar class from the calenda

year = int(input('Enter Year'))  # Prompt the user to input a year and convert it t

cal = TextCalendar()  # Create an instance of TextCalendar, which generates text-ba

# Print the formatted year calendar
# formatyear parameters:
# - year: The year for which the calendar is generated
# - width: (2) The width of each date column
# - length: (1) The number of lines for each week
# - c: (8) The spacing between months
# - m: (3) The number of months per row
print(cal.formatyear(year, 2, 1, 8, 3))
```

```
Enter Year2025
```

```
                                  2025

      January                   February                  March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
       1  2  3  4  5                      1  2                      1  2
 6  7  8  9 10 11 12       3  4  5  6  7  8  9       3  4  5  6  7  8  9
13 14 15 16 17 18 19      10 11 12 13 14 15 16      10 11 12 13 14 15 16
20 21 22 23 24 25 26      17 18 19 20 21 22 23      17 18 19 20 21 22 23
27 28 29 30 31            24 25 26 27 28            24 25 26 27 28 29 30
                                                    31

       April                     May                       June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6                1  2  3  4                         1
 7  8  9 10 11 12 13       5  6  7  8  9 10 11       2  3  4  5  6  7  8
14 15 16 17 18 19 20      12 13 14 15 16 17 18       9 10 11 12 13 14 15
21 22 23 24 25 26 27      19 20 21 22 23 24 25      16 17 18 19 20 21 22
28 29 30                  26 27 28 29 30 31         23 24 25 26 27 28 29
                                                    30

        July                    August                  September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6                   1  2  3       1  2  3  4  5  6  7
 7  8  9 10 11 12 13       4  5  6  7  8  9 10       8  9 10 11 12 13 14
14 15 16 17 18 19 20      11 12 13 14 15 16 17      15 16 17 18 19 20 21
21 22 23 24 25 26 27      18 19 20 21 22 23 24      22 23 24 25 26 27 28
28 29 30 31               25 26 27 28 29 30 31      29 30

      October                   November                  December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
       1  2  3  4  5                      1  2       1  2  3  4  5  6  7
 6  7  8  9 10 11 12       3  4  5  6  7  8  9       8  9 10 11 12 13 14
13 14 15 16 17 18 19      10 11 12 13 14 15 16      15 16 17 18 19 20 21
20 21 22 23 24 25 26      17 18 19 20 21 22 23      22 23 24 25 26 27 28
27 28 29 30 31            24 25 26 27 28 29 30      29 30 31
```

In [17]:

```python
from datetime import datetime  # Import the datetime class from the datetime module

# Get the current time
current_time = datetime.now()

# Format and print the current time
print("Current Time:", current_time.strftime("%H:%M:%S"))
```

```
Current Time: 22:05:40
```