Selecting Data with SQL

Introduction

As a data scientist, the SQL query you'll likely use most often is `SELECT`. This lesson introduces how to use `SELECT` to subset and transform the columns of a database table.

Objectives

You will be able to:

- Retrieve a `subset of columns` from a table
- Create an `alias` in a SQL query
- Use SQL `CASE` statements to transform selected columns
- Use built-in `SQL functions` to transform selected columns

The Data

Below, we connect to a SQLite database using the Python `sqlite3` library (documentation here):

```
# import module
import sqlite3
# create a connection
conn = sqlite3.connect('data.sqlite')
```

The database that you've just connected to is the same database you have seen previously, containing data about orders, employees, etc. Here's an overview of the database:

For this first section we'll be focusing on the `employees` table.

If we want to get all information about the employee records, we might do something like this (`*` means all columns):

```
# module
import pandas as pd
# pandas-sql-query
pd.read_sql("""SELECT * FROM employees;""", conn)
```

```
   employeeNumber   lastName firstName extension  \
0            1002     Murphy     Diane     x5800
1            1056  Patterson      Mary     x4611
2            1076   Firrelli      Jeff     x9273
3            1088  Patterson   William     x4871
```

```
4            1102     Bondur    Gerard     x5408
5            1143        Bow   Anthony     x5428
6            1165   Jennings    Leslie     x3291
7            1166   Thompson    Leslie     x4065
8            1188   Firrelli     Julie     x2173
9            1216  Patterson     Steve     x4334
10           1286      Tseng  Foon Yue     x2248
11           1323     Vanauf    George     x4102
12           1337     Bondur      Loui     x6493
13           1370  Hernandez    Gerard     x2028
14           1401   Castillo    Pamela     x2759
15           1501       Bott     Larry     x2311
16           1504      Jones     Barry      x102
17           1611     Fixter      Andy      x101
18           1612      Marsh     Peter      x102
19           1619       King       Tom      x103
20           1621      Nishi      Mami      x101
21           1625       Kato   Yoshimi      x102
22           1702     Gerard    Martin     x2312

                             email officeCode reportsTo
jobTitle
0      dmurphy@classicmodelcars.com          1
President
1    mpatterso@classicmodelcars.com          1      1002
VP Sales
2    jfirrelli@classicmodelcars.com          1      1002          VP
Marketing
3   wpatterson@classicmodelcars.com          6      1056  Sales
Manager (APAC)
4      gbondur@classicmodelcars.com          4      1056   Sale
Manager (EMEA)
5         abow@classicmodelcars.com          1      1056    Sales
Manager (NA)
6    ljennings@classicmodelcars.com          1      1143
Sales Rep
7    lthompson@classicmodelcars.com          1      1143
Sales Rep
8    jfirrelli@classicmodelcars.com          2      1143
Sales Rep
9   spatterson@classicmodelcars.com          2      1143
Sales Rep
10     ftseng@classicmodelcars.com          3      1143
Sales Rep
11     gvanauf@classicmodelcars.com          3      1143
Sales Rep
12     lbondur@classicmodelcars.com          4      1102
Sales Rep
13   ghernande@classicmodelcars.com          4      1102
```

```
Sales Rep
14   pcastillo@classicmodelcars.com          4      1102
Sales Rep
15       lbott@classicmodelcars.com          7      1102
Sales Rep
16      bjones@classicmodelcars.com          7      1102
Sales Rep
17     afixter@classicmodelcars.com          6      1088
Sales Rep
18      pmarsh@classicmodelcars.com          6      1088
Sales Rep
19       tking@classicmodelcars.com          6      1088
Sales Rep
20      mnishi@classicmodelcars.com          5      1056
Sales Rep
21       ykato@classicmodelcars.com          5      1621
Sales Rep
22     mgerard@classicmodelcars.com          4      1102
Sales Rep
```

Quick Note on String Syntax

When working with strings, you may have previously seen a `'string'`, a `"string"`, a `'''string'''`, or a `"""string"""`. While all of these are strings, the triple quotes have the added functionality of being able to use multiple lines within the same string as well as to use single quotes within the string. Sometimes, SQL queries can be much longer than others, in which case it's helpful to use new lines for readability. Here's the same example, this time with the string spread out onto multiple lines:

```
pd.read_sql("""
            SELECT
              *
            FROM
                employees;
            """, conn
          )
```

```
   employeeNumber   lastName firstName extension  \
0            1002     Murphy     Diane     x5800
1            1056  Patterson      Mary     x4611
2            1076   Firrelli      Jeff     x9273
3            1088  Patterson   William     x4871
4            1102     Bondur    Gerard     x5408
5            1143        Bow   Anthony     x5428
6            1165   Jennings    Leslie     x3291
7            1166   Thompson    Leslie     x4065
8            1188   Firrelli     Julie     x2173
9            1216  Patterson     Steve     x4334
10           1286      Tseng  Foon Yue     x2248
```

```
11           1323     Vanauf    George     x4102
12           1337     Bondur      Loui     x6493
13           1370  Hernandez    Gerard     x2028
14           1401   Castillo    Pamela     x2759
15           1501       Bott     Larry     x2311
16           1504      Jones     Barry      x102
17           1611     Fixter      Andy      x101
18           1612      Marsh     Peter      x102
19           1619       King       Tom      x103
20           1621      Nishi      Mami      x101
21           1625       Kato   Yoshimi      x102
22           1702     Gerard    Martin     x2312

                              email officeCode reportsTo
jobTitle
0      dmurphy@classicmodelcars.com          1
President
1    mpatterso@classicmodelcars.com          1      1002
VP Sales
2    jfirrelli@classicmodelcars.com          1      1002          VP
Marketing
3   wpatterson@classicmodelcars.com          6      1056  Sales
Manager (APAC)
4      gbondur@classicmodelcars.com          4      1056   Sale
Manager (EMEA)
5         abow@classicmodelcars.com          1      1056    Sales
Manager (NA)
6    ljennings@classicmodelcars.com          1      1143
Sales Rep
7    lthompson@classicmodelcars.com          1      1143
Sales Rep
8    jfirrelli@classicmodelcars.com          2      1143
Sales Rep
9   spatterson@classicmodelcars.com          2      1143
Sales Rep
10     ftseng@classicmodelcars.com          3      1143
Sales Rep
11     gvanauf@classicmodelcars.com          3      1143
Sales Rep
12     lbondur@classicmodelcars.com          4      1102
Sales Rep
13   ghernande@classicmodelcars.com          4      1102
Sales Rep
14   pcastillo@classicmodelcars.com          4      1102
Sales Rep
15       lbott@classicmodelcars.com          7      1102
Sales Rep
16      bjones@classicmodelcars.com          7      1102
Sales Rep
```

```
17     afixter@classicmodelcars.com          6      1088
Sales Rep
18      pmarsh@classicmodelcars.com          6      1088
Sales Rep
19       tking@classicmodelcars.com          6      1088
Sales Rep
20      mnishi@classicmodelcars.com          5      1056
Sales Rep
21       ykato@classicmodelcars.com          5      1621
Sales Rep
22     mgerard@classicmodelcars.com          4      1102
Sales Rep
```

Unlike in Python, whitespace indentation in SQL is not used to indicate scope or any other important information. Therefore this:

```
SELECT *
  FROM employees;
```

(with two spaces in front of `FROM`)

is identical to this:

```
SELECT *
FROM employees;
```

(with zero spaces in front of `FROM`)

as far as SQL is concerned. However we will be aligning the right edge of the SQL keywords, using a "river" of whitespace down the center to improve legibility in this lesson, following this style guide. You will see multi-line SQL written with various different indentation styles, and you will want to check with your employer to learn what their style guide is.

Retrieving a Subset of Columns

Once we know what the column names are for a given table, we can select specific columns rather than using `*` to select all of them. This is achieved by replacing the `*` with the names of the columns, separated by commas.

For example, if we just wanted to select the last and first names of the employees:

```
pd.read_sql("""
      SELECT
            lastName, firstName
      FROM
            employees;
      """, conn
).head()
```

```
    lastName firstName
0     Murphy     Diane
1  Patterson      Mary
2   Firrelli      Jeff
3  Patterson   William
4     Bondur    Gerard
```

We can also specify the columns in a different order than they appear in the database, in order to reorder the columns in the resulting dataframe:

```
pd.read_sql("""
    SELECT
          firstName, lastName
    FROM
          employees;
""", conn
).head()
```

```
  firstName   lastName
0     Diane     Murphy
1      Mary  Patterson
2      Jeff   Firrelli
3   William  Patterson
4    Gerard     Bondur
```

Additionally, we can use **aliases** (`AS` keyword) to change the column names in our query result:

```
pd.read_sql("""
    SELECT
        firstName AS name
    FROM
        employees;
""", conn
).head()
```

```
      name
0    Diane
1     Mary
2     Jeff
3  William
4   Gerard
```

Note: the `AS` keyword is technically optional when assigning an alias in SQL, so you may see examples that don't include it. In other words, you could just say `SELECT firstName name` and it would work the same as `SELECT firstName AS name`. However we recommend being more explicit and including the `AS`, so that it's clearer what your code is doing.

Using SQL `CASE` Statements

`CASE` statements appear very frequently in SQL technical interview questions. They are a type of conditional statement, similar to `if` statements in Python. Whereas Python uses the keywords `if`, `elif`, and `else`, SQL uses `CASE`, `WHEN`, `THEN`, `ELSE`, and `END`.

`CASE` indicates that a conditional statement has begun, and `END` indicates that it has ended.

`WHEN` is similar to `if`, and then instead of a colon and an indented block, `THEN` indicates what should happen if the condition is true. After the first `THEN` has executed, it skips to the end, so each subsequent `WHEN` is more like `elif` in Python.

`ELSE` is essentially the same as `else` in Python.

`CASE` to Bin Column Values

One of the most common use cases for `CASE` statements is to bin the column values. This is true for both numeric and categorical columns.

In the example below, we use the `jobTitle` field to bin all employees into `role` categories based on whether or not their job title is "Sales Rep":

```
# Trial-test # brian-added
pd.read_sql("""SELECT firstName, lastName, jobTitle From employees
WHERE jobTitle = 'Sales Rep';""", conn).head()
```

	firstName	lastName	jobTitle
0	Leslie	Jennings	Sales Rep
1	Leslie	Thompson	Sales Rep
2	Julie	Firrelli	Sales Rep
3	Steve	Patterson	Sales Rep
4	Foon Yue	Tseng	Sales Rep

```
pd.read_sql("""
SELECT firstName, lastName, jobTitle,
       CASE
       WHEN jobTitle = "Sales Rep" THEN "Sales Rep"
       ELSE "Not Sales Rep"
       END AS role
  FROM employees;
""", conn).head(10)
```

	firstName	lastName	jobTitle	role
0	Diane	Murphy	President	Not Sales Rep
1	Mary	Patterson	VP Sales	Not Sales Rep
2	Jeff	Firrelli	VP Marketing	Not Sales Rep
3	William	Patterson	Sales Manager (APAC)	Not Sales Rep
4	Gerard	Bondur	Sale Manager (EMEA)	Not Sales Rep
5	Anthony	Bow	Sales Manager (NA)	Not Sales Rep
6	Leslie	Jennings	Sales Rep	Sales Rep
7	Leslie	Thompson	Sales Rep	Sales Rep

```
8     Julie   Firrelli            Sales Rep      Sales Rep
9     Steve  Patterson            Sales Rep      Sales Rep
```

`CASE` to Make Values Human-Readable

Another typical way to use `CASE` is to translate the column values into something that your eventual audience will understand. This is especially true of data that is entered into the database as a "code" or "ID" rather than a human-readable name.

In the example below, we use a `CASE` statement with multiple `WHEN`'s in order to transform the `officeCode` column into an `office` column that uses a more meaningful name for the office:

```
pd.read_sql("""
SELECT firstName, lastName, officeCode,
       CASE
       WHEN officeCode = "1" THEN "San Francisco, CA"
       WHEN officeCode = "2" THEN "Boston, MA"
       WHEN officeCode = "3" THEN "New York, NY"
       WHEN officeCode = "4" THEN "Paris, France"
       WHEN officeCode = "5" THEN "Tokyo, Japan"
       END AS office
  FROM employees;
""", conn).head(10)
```

```
  firstName   lastName officeCode            office
0     Diane     Murphy          1  San Francisco, CA
1      Mary  Patterson          1  San Francisco, CA
2      Jeff   Firrelli          1  San Francisco, CA
3   William  Patterson          6              None
4    Gerard     Bondur          4     Paris, France
5   Anthony        Bow          1  San Francisco, CA
6    Leslie   Jennings          1  San Francisco, CA
7    Leslie   Thompson          1  San Francisco, CA
8     Julie   Firrelli          2        Boston, MA
9     Steve  Patterson          2        Boston, MA
```

`Note` that because **we did not specify a name for `officeCode` "6"**, and did not include an `ELSE`, the associated office value for William Patterson is `NULL` (represented as `None` in Python).

There is also a shorter syntax possible if all of the `WHEN`s are just checking if a value is equal to another value (e.g. in this case where we are repeating `officeCode =` over and over). Instead we can specify `officeCode` right after `CASE`, then only specify the potential matching values:

```
pd.read_sql("""
SELECT firstName, lastName, officeCode,
       CASE officeCode
       WHEN "1" THEN "San Francisco, CA"
       WHEN "2" THEN "Boston, MA"
       WHEN "3" THEN "New York, NY"
```

```
       WHEN "4" THEN "Paris, France"
       WHEN "5" THEN "Tokyo, Japan"
       END AS office
  FROM employees;
""", conn).head(10)
```

	firstName	lastName	officeCode	office
0	Diane	Murphy	1	San Francisco, CA
1	Mary	Patterson	1	San Francisco, CA
2	Jeff	Firrelli	1	San Francisco, CA
3	William	Patterson	6	None
4	Gerard	Bondur	4	Paris, France
5	Anthony	Bow	1	San Francisco, CA
6	Leslie	Jennings	1	San Francisco, CA
7	Leslie	Thompson	1	San Francisco, CA
8	Julie	Firrelli	2	Boston, MA
9	Steve	Patterson	2	Boston, MA

Using Built-in SQL Functions

Similar to the Python built-in functions, SQL also has built-in functions. The available functions will differ somewhat by the type of SQL you are using, but in general you should be able to find functions for:

- String manipulation
- Math operations
- Date and time operations

For SQLite in particular, if you are looking for a built-in function, start by checking the core functions page, mathematical functions page, and/or date and time functions page.

Built-in SQL Functions for String Manipulation

`length`

Let's start with an example of a SQL built-in function that is very similar to one we have in Python: `length` (documentation here). This works very similarly to the `len` built-in function in Python. For a string, it returns the number of characters.

If we wanted to find the length of the first names of all employees, that would look like this:

```
pd.read_sql("""
  SELECT
        firstName, length(firstName) AS length_of_name
  FROM
      employees;
""", conn).head()
```

	firstName	length_of_name
0	Diane	5

```
1      Mary                4
2      Jeff                4
3   William                7
4    Gerard                6
```

`upper`

Now let's say we wanted to return all of the employee names in all caps. Similar to the Python string method, this SQL function is called `upper` (documentation here). However, since it's a built-in function and not a method, the syntax looks like:

```
upper(column_name)
```

and not `column_name.upper()`.

As you get more comfortable with Python and SQL, distinctions like this will get more intuitive, but for now don't worry if you have to look it up every time!

Here is an example using `upper`:

```
pd.read_sql("""
  SELECT
        firstName, upper(firstName) AS name_in_CAPS
  FROM
        employees;
""", conn).head()
```

```
  firstName name_in_CAPS
0     Diane        DIANE
1      Mary         MARY
2      Jeff         JEFF
3   William      WILLIAM
4    Gerard       GERARD
```

`substr`

Another form of string manipulation you might need is finding a substring (subset of a string). In Python, we do this with string slicing. In SQL, there is a built-in function that does this instead. For SQLite specifically, this is called `substr` (documentation here).

Let's say we wanted just the first initial (first letter of the first name) for each employee:

```
pd.read_sql("""
    SELECT
          firstName, substr(firstName, 1, 1) AS first_initial,
length(substr(firstName, 1, 1)) AS Length_of_Substr
    FROM
          employees;
""", conn).head()
```

```
  firstName first_initial  Length_of_Substr
0     Diane             D                 1
1      Mary             M                 1
2      Jeff             J                 1
3   William             W                 1
4    Gerard             G                 1
```

If we wanted to add a `.` after each first initial, we could use the SQLite `||` (concatenate) operator. This works similarly to `+` with strings in Python:

```
pd.read_sql("""
  SELECT
        substr(firstName, 1, 1) || "." AS first_initial
    FROM
        employees;
""", conn).head()
```

```
  first_initial
0            D.
1            M.
2            J.
3            W.
4            G.
```

We can also combine multiple column values, not just string literals. For example, below we combine the first and last name:

```
pd.read_sql("""
  SELECT
        firstName || lastName AS full_name
  FROM
      employees;
""", conn).head()
```

```
          full_name
0       DianeMurphy
1     MaryPatterson
2      JeffFirrelli
3  WilliamPatterson
4      GerardBondur
```

Hmm, that looks a bit odd. Let's concatenate those column values with a space (`" "`) string literal:

```
pd.read_sql("""
  SELECT
        firstName || " " || lastName AS full_name
  FROM
```

```
        employees;
""", conn).head()
```

```
           full_name
0       Diane Murphy
1     Mary Patterson
2      Jeff Firrelli
3  William Patterson
4      Gerard Bondur
```

That looks better!

Built-in SQL Functions for Math Operations

For these examples, let's switch over to using the `orderDetails` table:

```
pd.read_sql("""SELECT * FROM orderDetails;""", conn).head()
```

```
  orderNumber productCode quantityOrdered priceEach orderLineNumber
0       10100    S18_1749              30    136.00               3
1       10100    S18_2248              50     55.09               2
2       10100    S18_4409              22     75.46               4
3       10100    S24_3969              49     35.29               1
4       10101    S18_2325              25    108.06               4
```

`round`

Let's say we wanted to round the price to the nearest dollar. We could use the SQL `round` function (documentation here), which is very similar to the the Python `round`:

```
pd.read_sql("""
  SELECT
        priceEach, round(priceEach) AS rounded_price
  FROM
        orderDetails;
""", conn).head()
```

```
  priceEach  rounded_price
0    136.00          136.0
1     55.09           55.0
2     75.46           75.0
3     35.29           35.0
4    108.06          108.0
```

`CAST`

The previous result looks ok, but it's returning floating point numbers. What if we want integers instead?

In Python, we might apply the `int` built-in function. In SQLite, we can use a `CAST` expression (documentation here):

```
pd.read_sql("""
SELECT
      CAST(round(priceEach) AS INTEGER) AS rounded_price_int
FROM
      orderDetails;
""", conn).head()
```

```
   rounded_price_int
0                136
1                 55
2                 75
3                 35
4                108
```

Basic Math Operations

Just like when performing math operations with Python, you don't always need to use a function. Sometimes all you need is an operator like `+`, `-`, `/`, or `*`. For example, below we multiply the price times the quantity ordered to find the total price:

```
pd.read_sql("""
SELECT
      priceEach * quantityOrdered AS Total_Price

FROM
      orderDetails;
""", conn).head()
# alternative
## CAST((priceEach * quantityOrdered) AS INTEGER) AS Total_Price
```

```
   Total_Price
0      4080.00
1      2754.50
2      1660.12
3      1729.21
4      2701.50
```

Built-in SQL Functions for Date and Time Operations

For these examples, we'll look at yet another table within the database, this time the `orders` table:

```
pd.read_sql("""SELECT * FROM orders;""", conn).head()
```

```
  orderNumber   orderDate requiredDate shippedDate   status  \
0       10100  2003-01-06   2003-01-13  2003-01-10  Shipped
```

```
1       10101  2003-01-09   2003-01-18  2003-01-11  Shipped
2       10102  2003-01-10   2003-01-18  2003-01-14  Shipped
3       10103  2003-01-29   2003-02-07  2003-02-02  Shipped
4       10104  2003-01-31   2003-02-09  2003-02-01  Shipped
```

```
                  comments customerNumber
0                                     363
1  Check on availability.             128
2                                     181
3                                     121
4                                     141
```

What if we wanted to know how many days there are between the `requiredDate` and the `orderDate` for each order? Intuitively you might try something like this:

```
pd.read_sql("""
    SELECT
          requiredDate - orderDate
    FROM
          orders;
""", conn)
```

```
     requiredDate - orderDate
0                           0
1                           0
2                           0
3                           0
4                           0
..                        ...
321                         0
322                         0
323                         0
324                         0
325                         0

[326 rows x 1 columns]
```

Clearly that didn't work.

It turns out that we need to specify that we want the difference in *days*. One way to do this is using the `julianday` function (documentation here):

```
pd.read_sql("""
SELECT
      CAST(julianday(requiredDate) - julianday(orderDate) AS INTEGER)
AS days_from_order_to_required
FROM
    orders;
""", conn)
```

```
     days_from_order_to_required
0                              7
1                              9
2                              8
3                              9
4                              9
..                           ...
321                            8
322                           12
323                            6
324                            8
325                            7

[326 rows x 1 columns]
```

If we wanted to select the order dates as well as dates 1 week after the order dates, that would look like this:

```
pd.read_sql("""
    SELECT
          orderDate, date(orderDate, "+7 days") AS one_week_later,
date(orderDate, "+14 days") AS two_week_later
    FROM
        orders;
""", conn)
```

```
      orderDate one_week_later two_week_later
0    2003-01-06     2003-01-13     2003-01-20
1    2003-01-09     2003-01-16     2003-01-23
2    2003-01-10     2003-01-17     2003-01-24
3    2003-01-29     2003-02-05     2003-02-12
4    2003-01-31     2003-02-07     2003-02-14
..          ...            ...            ...
321  2005-05-29     2005-06-05     2005-06-12
322  2005-05-30     2005-06-06     2005-06-13
323  2005-05-30     2005-06-06     2005-06-13
324  2005-05-31     2005-06-07     2005-06-14
325  2005-05-31     2005-06-07     2005-06-14

[326 rows x 3 columns]
```

You can also use the `strftime` function, which is very similar to the Python version. This is useful if you want to split apart a date or time value into different sub-parts. For example, here we extract the year, month, and day of month from the order date:

```
pd.read_sql("""
    SELECT orderDate,
          strftime("%m", orderDate) AS month,
          strftime("%Y", orderDate) AS year,
```

```
          strftime("%d", orderDate) AS day
    FROM orders;
""", conn).head()
```

```
    orderDate month  year day
0  2003-01-06    01  2003  06
1  2003-01-09    01  2003  09
2  2003-01-10    01  2003  10
3  2003-01-29    01  2003  29
4  2003-01-31    01  2003  31
```

Now that we are finished with our queries, we can close the database connection.

```
conn.close()
```

Summary

In this lesson, you saw how to execute several kinds of SQL `SELECT` queries. First, there were examples of specifying the selection of particular columns, rather than always using `SELECT *` to select all columns. Then you saw some examples of how to use `CASE` to transform column values using conditional logic. Finally, we walked through how to use built-in SQL functions, particularly for string, numeric, and date/time fields.