```sql
-- SQL Aggregations Lesson Overview
-- In this lesson, we will cover and you will be able to:
/*
    Deal with NULL values
    Create aggregations in your SQL Queries including:
        COUNT
        SUM
        MIN & MAX
        AVG
        GROUP BY
        DISTINCT
        HAVING
    Create DATE functions
    Implement CASE statements
*/

-- NULLs
-- NULLs are a datatype that specifies where no data exists in SQL.
-- They are often ignored in our aggregation functions like COUNT SUM AVG MIN MAX

-- NULLs and Aggregation

-- Notice that NULLs are different than a zero, they are cells where data does not exist.
SELECT *
    FROM accounts
    WHERE id > 1500 and id < 1600
    LIMIT 5;

-- When identifying NULLs in a WHERE clause, we write IS NULL or IS NOT NULL.
-- We don't use =, because NULL isn't considered a value in SQL. Rather,
-- it is a property of the data.

-- IS NULL
SELECT *
    FROM accounts
    WHERE primary_poc IS NULL
    LIMIT 5;

-- IS NOT NULL
SELECT *
    FROM accounts
    WHERE primary_poc IS NOT NULL
    LIMIT 5;

-- NULLs - Expert Tip

-- There are two common ways in which you are likely to encounter NULLs:
/*
    NULLs frequently occur when performing a LEFT or RIGHT JOIN.
    You saw in the last lesson - when some rows in the left table
    of a left join are not matched with rows in the right table,
    those rows will contain some NULL values in the result set.
*/

/*
    NULLs can also occur from simply missing data in our database.
*/

-- COUNT the Number of Rows in a Table
-- finding the number of rows in each table. Here is an example of finding all the rows.
-- ignore NULL VALUES.

SELECT COUNT(*)
    FROM accounts;

SELECT COUNT(accounts.id)
    FROM accounts;

SELECT *
```

```sql
    FROM orders
    WHERE occurred_at >= '2016-12-01'
        AND occurred_at < '2017-01-01';

SELECT COUNT(*)
    FROM orders
    WHERE occurred_at >= '2016-12-01'
        AND occurred_at < '2017-01-01';

SELECT COUNT(*) AS order_count
    FROM orders
    WHERE occurred_at >= '2016-12-01'
        AND occurred_at < '2017-01-01';

-- Notice that COUNT does not consider rows that have NULL values.
-- Therefore, this can be useful for quickly identifying which rows have missing data.

SELECT COUNT (*) AS account_count
    FROM accounts;

SELECT COUNT (id) AS account_id_count
    FROM accounts;

SELECT COUNT(primary_poc) AS account_primary_poc_count
    FROM accounts;

SELECT *
    FROM accounts
    WHERE primary_poc IS NULL;

SELECT count(*)
    FROM accounts
    WHERE primary_poc IS NULL;

-- Unlike COUNT, you can only use SUM on numeric columns. However, SUM will ignore NULL values

SELECT *
    FROM orders;

--summation of quantity of each type of paper
SELECT SUM(standard_qty) AS standard,
       SUM(gloss_qty) AS gloss,
       SUM(poster_qty) AS poster
    FROM orders;

--summation of income of each type of paper
SELECT SUM(orders.standard_amt_usd) AS standard_income,
       SUM(orders.gloss_amt_usd) AS gloss_income,
       SUM(orders.poster_amt_usd) AS poster_income
    FROM orders;

SELECT SUM(standard_qty) AS standard_quantity,
       SUM(orders.standard_amt_usd) AS standard_income,
       SUM(gloss_qty) AS gloss_quantity,
       SUM(orders.gloss_amt_usd) AS gloss_income,
       SUM(poster_qty) AS poster_quantity,
       SUM(orders.poster_amt_usd) AS poster_income
    FROM orders;

-- Quiz: SUM
/*
Aggregation Questions:
    Use the SQL environment below to find the solution for each
    of the following questions. If you get stuck or want to check
    your answers, you can find the answers at the top of the next concept.
*/

-- Find the total amount of poster_qty paper ordered in the orders table.
--code
```

```sql
SELECT sum(orders.poster_qty) AS amount_poster_qty_paper
    FROM orders;

-- Find the total amount of standard_qty paper ordered in the orders table.
-- code
SELECT sum(orders.standard_qty) AS amount_standard_qty_paper
    FROM orders;

-- Find the total dollar amount of sales using the total_amt_usd in the orders table.
-- code
SELECT sum(orders.total_amt_usd) AS total_amt_usd
    FROM orders;

-- Find the total amount spent on standard_amt_usd and gloss_amt_usd paper for each order in the orders table.
-- This should give a dollar amount for each order in the table.
--code
SELECT (orders.standard_amt_usd + orders.gloss_amt_usd) AS total_gloss_and_standard_amt_usd
    FROM orders;

-- Find the standard_amt_usd per unit of standard_qty paper.
-- Your solution should use both aggregation and a mathematical operator.

SELECT sum(standard_amt_usd)/sum(standard_qty) AS standard
    FROM orders;

-- MIN will return the lowest number.
-- MAX does the opposite—it returns the highest number
-- MIN and MAX number of orders of each paper type. However, you could run each individually.
-- Notice that MIN and MAX are aggregators that again ignore NULL values.

SELECT MIN(standard_qty) AS standard_min,
       MIN(gloss_qty) AS gloss_min,
       MIN(poster_qty) AS poster_min,
       MAX(standard_qty) AS standard_max,
       MAX(gloss_qty) AS gloss_max,
       MAX(poster_qty) AS poster_max
    FROM orders;

-- Similar to other software AVG returns the mean of the data
-- that is the sum of all of the values in the column divided by the number of values in a column.
-- This aggregate function again ignores the NULL values in both the numerator and the denominator.

SELECT AVG(standard_qty) AS standard_avg,
       AVG(gloss_qty) AS gloss_avg,
       AVG(poster_qty) AS poster_avg
    FROM orders;

-- Quiz: MIN, MAX, & AVG
/*
Questions: MIN, MAX, & AVERAGE :
    Use the SQL environment below to assist with answering the following questions.
    Whether you get stuck or you just want to double-check your solutions
    my answers can be found at the top of the next concept.
*/

-- When was the earliest order ever placed? You only need to return the date.
-- code
SELECT min(orders.occurred_at)
    FROM orders;

-- Try performing the same query as in question 1 without using an aggregation function.
-- code
SELECT orders.occurred_at
    FROM orders
    ORDER BY orders.occurred_at
    LIMIT 1;

-- When did the most recent (latest) web_event occur?
```

```sql
-- code
SELECT max(web_events.occurred_at)
    FROM web_events;

-- Try to perform the result of the previous query without using an aggregation function.
-- code
SELECT occurred_at
    FROM web_events
    ORDER BY occurred_at DESC
    LIMIT 1;

/*
    Find the mean (AVERAGE) amount spent per order on each paper type,
    as well as the mean amount of each paper type purchased per order.
    Your final answer should have 6 values - one for each paper type
    for the average number of sales, as well as the average amount.
*/
-- code
SELECT avg(orders.total_amt_usd/orders.standard_amt_usd),
       avg(orders.total_amt_usd/orders.gloss_amt_usd),
       avg(orders.total_amt_usd/orders.poster_amt_usd),
       avg(orders.total/orders.standard_qty),
       avg(orders.total/orders.gloss_qty),
       avg(orders.total/orders.poster_qty)
    FROM orders;


SELECT AVG(standard_qty) mean_standard, AVG(gloss_qty) mean_gloss,
        AVG(poster_qty) mean_poster, AVG(standard_amt_usd) mean_standard_usd,
        AVG(gloss_amt_usd) mean_gloss_usd, AVG(poster_amt_usd) mean_poster_usd
FROM orders;
/*
    Via the video, you might be interested in
    how to calculate the MEDIAN. Though this is more advanced
    than what we have covered so far try finding
    what is the MEDIAN total_usd spent on all orders?
*/
-- code
SELECT *
    FROM (
        SELECT total_amt_usd
            FROM orders
            ORDER BY total_amt_usd
            LIMIT 3457) AS Table1
    ORDER BY total_amt_usd DESC
    LIMIT 2;


-- GROUP BY can be used to aggregate data within subsets of the data.
-- For example, grouping for different accounts, different regions, or different
-- sales representatives.

-- Any column in the SELECT statement that is not within an aggregator must be in the GROUP BY clause.

-- The GROUP BY always goes between WHERE and ORDER BY.

-- ORDER BY works like SORT in spreadsheet software.

SELECT account_id,
       SUM(standard_qty) AS standard,
       SUM(gloss_qty) AS gloss,
       SUM(poster_qty) AS poster
    FROM orders;

SELECT account_id,
       SUM(standard_qty) AS standard,
       SUM(gloss_qty) AS gloss,
       SUM(poster_qty) AS poster
    FROM orders
```

```sql
    GROUP BY account_id
    ORDER BY account_id;

-- Quiz: GROUP BY
/*
GROUP BY Note:
    Now that you have been introduced to JOINs, GROUP BY, and aggregate
    functions, the real power of SQL starts to come to life.
    Try some of the below to put your skills to the test!
*/

/*
Questions: GROUP BY:
    Use the SQL environment below to assist with answering the following questions.
    Whether you get stuck or you just want to double-check your solutions,
    my answers can be found at the top of the next concept.

    One part that can be difficult to recognize is when it might be
    easiest to use an aggregate or one of the other SQL functionalities.
    Try some of the below to see if you can differentiate to find
    the easiest solution.
*/

-- Which account (by name) placed the earliest order?
-- Your solution should have the account name and the date of the order.
-- code
SELECT accounts.name , orders.occurred_at
    FROM orders
    JOIN accounts
        ON accounts.id = orders.account_id
    ORDER BY orders.occurred_at
    LIMIT 1;


-- Find the total sales in usd for each account.
-- You should include two columns - the total sales for each company's orders in usd and the company name.
-- code
SELECT accounts.name , orders.total_amt_usd
    FROM orders
    JOIN accounts
        ON accounts.id = orders.account_id
    ORDER BY accounts.name;


-- Via what channel did the most recent (latest) web_event occur,
-- which account was associated with this web_event?
-- Your query should return only three values - the date, channel, and account name.
-- code
SELECT accounts.name , web_events.occurred_at , web_events.channel
    FROM web_events
    JOIN accounts
        ON accounts.id = web_events.account_id
    GROUP BY accounts.name
    ORDER BY web_events.occurred_at DESC
    LIMIT 1;

-- Find the total number of times each type of channel from the web_events was used.
-- Your final table should have two columns ,
-- the channel and the number of times the channel was used.
-- code
SELECT w.channel, COUNT(*)
    FROM web_events w
    GROUP BY w.channel

-- Who was the primary contact associated with the earliest web_event?
-- code
SELECT w.channel, COUNT(*)
    FROM web_events w
    ORDER BY w.channel
```

```sql

-- What was the smallest order placed by each account in terms of total usd.
-- Provide only two columns - the account name and the total usd.
-- Order from smallest dollar amounts to largest.
-- code
SELECT a.primary_poc
FROM web_events w
JOIN accounts a
ON a.id = w.account_id
ORDER BY w.occurred_at
LIMIT 1;

-- Find the number of sales reps in each region.
-- Your final table should have two columns ,the region and the number of sales_reps.
-- Order from the fewest reps to most reps.
-- code
SELECT a.name, MIN(total_amt_usd) smallest_order
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.name
ORDER BY smallest_order;

/*
    You can GROUP BY multiple columns at once, as we showed here.
    This is often useful to aggregate across a number of different segments.
*/

/*
    The order of columns listed in the ORDER BY clause does make a difference.
    You are ordering the columns from left to right.
*/
/*
    GROUP BY - Expert Tips:

        The order of column names in your GROUP BY clause doesn't matter
        the results will be the same regardless. If we run the same query and reverse
        the order in the GROUP BY clause, you can see we get the same results.

        As with ORDER BY, you can substitute numbers for column names
        in the GROUP BY clause. It's generally recommended to do this only
        when you're grouping many columns, or if something else is causing
        the text in the GROUP BY clause to be excessively long.

        A reminder here that any column that is not within an aggregation must show up in your GROUP BY statement.
        If you forget, you will likely get an error. However, in the off chance that your query does work,
        you might not like the results!
*/
SELECT account_id,
       channel,
       COUNT(id) as events
    FROM web_events
    GROUP BY account_id, channel
    ORDER BY account_id, channel;

SELECT account_id,
       channel,
       COUNT(id) as events
    FROM web_events
    GROUP BY account_id, channel
    ORDER BY account_id, channel DESC;


/*
    Quiz: GROUP BY Part II
    Questions: GROUP BY Part II
    Use the SQL environment below to assist with answering the following questions.
    Whether you get stuck or you just want to double-check your solutions,
```

```
    my answers can be found at the top of the next concept.
*/


-- For each account, determine the average amount of each type of paper they purchased across their orders.
-- Your result should have four columns - one for the account name and one for the average quantity purchased for each of the paper types for each account.
--code
SELECT  accounts.name,
        AVG(orders.standard_qty) AS avg_stand ,
        AVG(orders.gloss_qty) AS avg_gloss ,
        AVG(orders.poster_qty) AS avg_poster
    FROM accounts
    JOIN orders
        ON accounts.id = orders.account_id
    GROUP BY accounts.name;

-- For each account, determine the average amount spent per order on each paper type.
-- Your result should have four columns - one for the account name and one for the average amount spent on each paper type.
-- code
SELECT  accounts.name,
        AVG(orders.standard_amt_usd) AS avg_stand_price ,
        AVG(orders.gloss_amt_usd) AS avg_gloss_price ,
        AVG(orders.poster_amt_usd) AS avg_poster_price
    FROM accounts
    JOIN orders
        ON accounts.id = orders.account_id
    GROUP BY accounts.name;


-- Determine the number of times a particular channel was used in the web_events table for each sales rep.
-- Your final table should have three columns - the name of the sales rep, the channel, and the number of occurrences.
-- Order your table with the highest number of occurrences first.
--code
SELECT  sales_reps.name , count(web_events.channel) AS no_channels ,
        count(web_events.occurred_at) AS  no_occurrences
    FROM accounts
    JOIN web_events
        ON accounts.id = web_events.account_id
    JOIN sales_reps
        ON accounts.id = sales_reps.id
    ORDER BY web_events.occurred_at;



-- Determine the number of times a particular channel was used in the web_events table for each region.
-- Your final table should have three columns - the region name, the channel, and the number of occurrences.
-- Order your table with the highest number of occurrences first.
--code









-- DISTINCT is always used in SELECT statements
-- it provides the unique rows for all columns written in the SELECT statement.
-- Therefore, you only use DISTINCT once in any particular SELECT statement.

-- which would return the unique (or DISTINCT) rows across all three columns.

```

```sql
-- DISTINCT - Expert Tip
-- It's worth noting that using DISTINCT, particularly in aggregations,
--  can slow your queries down quite a bit.

SELECT account_id , channel , COUNT(id) as events
    FROM web_events
    GROUP BY account_id, channel
    ORDER BY account_id, channel DESC;

SELECT account_id , channel
    FROM web_events
    GROUP BY account_id, channel
    ORDER BY account_id;

SELECT DISTINCT account_id , channel
    FROM web_events
    ORDER BY account_id;

-- Quiz: DISTINCT
/*
Questions: DISTINCT:
    Use the SQL environment below to assist with answering the following questions.
    Whether you get stuck or you just want to double-check your solutions,
    my answers can be found at the top of the next concept.
*/

-- Use DISTINCT to test if there are any accounts associated with more than one region.
-- code
SELECT DISTINCT
    FROM accounts
    JOIN sales_reps
        ON accounts.sales_rep_id = accounts.id
    JOIN region
        ON  sales_reps.region_id = region.id


SELECT DISTINCT id, name
FROM accounts;

-- Have any sales reps worked on more than one account?
-- code
SELECT DISTINCT
    FROM

SELECT DISTINCT id, name
FROM sales_reps;

-- HAVING is the "clean" way to filter a query that has been aggregated
-- Essentially, any time you want to perform a WHERE on an element of your query that was created by an aggregate, you need to use HAVING instead.

SELECT account_id , SUM(total_amt_usd) AS sum_total_amt_usd
    FROM orders
    GROUP BY account_id
    HAVING SUM(total_amt_usd) >= 250000;
/*
    Questions: HAVING
        Use the SQL environment below to assist with answering the following questions.
        Whether you get stuck or you just want to double-check your solutions,
        my answers can be found at the top of the next concept.
*/
-- How many of the sales reps have more than 5 accounts that they manage?
-- code
SELECT sales_reps.id , sales_reps.name , COUNT(*) AS no_accounts
    FROM accounts
    JOIN sales_reps
        ON accounts.sales_rep_id = sales_reps.id
    GROUP BY sales_reps.id , sales_reps.name
    HAVING no_accounts > 5;
```

```sql

-- How many accounts have more than 20 orders?
-- code
SELECT accounts.id , accounts.name , COUNT(*) AS no_orders
    FROM accounts
    JOIN orders
        ON orders.account_id = accounts.id
    GROUP BY accounts.id , accounts.name
    HAVING no_orders > 20;

-- Which account has the most orders?
-- code
SELECT accounts.id , accounts.name , COUNT(*) AS no_orders
    FROM accounts
    JOIN orders
        ON orders.account_id = accounts.id
    GROUP BY accounts.id , accounts.name
    ORDER by no_orders DESC
    LIMIT 1;

-- Which accounts spent more than 30,000 usd total across all orders?
-- code
SELECT accounts.id , accounts.name , orders.total_amt_usd
    FROM accounts
    JOIN orders
        ON orders.account_id = accounts.id
    GROUP BY accounts.id , accounts.name
    HAVING orders.total_amt_usd > 30000
    ORDER BY orders.total_amt_usd DESC;

-- Which accounts spent less than 1,000 usd total across all orders?
-- code
SELECT accounts.id , accounts.name , orders.total_amt_usd
    FROM accounts
    JOIN orders
        ON orders.account_id = accounts.id
    GROUP BY accounts.id , accounts.name
    HAVING orders.total_amt_usd < 1000
    ORDER BY orders.total_amt_usd;

-- Which account has spent the most with us?
-- code
SELECT accounts.id , accounts.name , SUM(orders.total_amt_usd) AS total_spent
    FROM accounts
    JOIN orders
        ON orders.account_id = accounts.id
    GROUP BY accounts.id , accounts.name
    ORDER BY total_spent DESC
    LIMIT 1;

-- Which account has spent the least with us?
-- code
SELECT accounts.id , accounts.name , SUM(orders.total_amt_usd) AS total_spent
    FROM accounts
    JOIN orders
        ON orders.account_id = accounts.id
    GROUP BY accounts.id , accounts.name
    ORDER BY total_spent
    LIMIT 1;

-- Which accounts used facebook as a channel to contact customers more than 6 times?
-- code
SELECT accounts.id , accounts.name , SUM(orders.total_amt_usd) AS total_spent
    FROM accounts
    JOIN orders
        ON orders.account_id = accounts.id
    GROUP BY accounts.id , accounts.name
    ORDER BY total_spent DESC
    LIMIT 1;
```

```sql

-- Which account used facebook most as a channel?
-- code
SELECT accounts.id , accounts.name , web_events.channel
    FROM accounts
    JOIN web_events
        ON web_events.account_id = accounts.id
    WHERE web_events.channel like 'facebook'
    GROUP BY accounts.id , accounts.name ;

-- Which channel was most frequently used by most accounts?
-- code
SELECT accounts.id , accounts.name , web_events.channel , count(*) AS no_channel
    FROM accounts
    JOIN web_events
        ON web_events.account_id = accounts.id
    WHERE web_events.channel like 'facebook'
    GROUP BY accounts.id , accounts.name
    ORDER BY sum_channel DESC
    LIMIT 10;

-- DATE_TRUNC
--  allows you to truncate your date to a particular part of your date-time column.
--
--
-- DATE_PART
--  can be useful for pulling a specific portion of a date, but notice pulling month or day of the week (dow) means that you are no longer keeping the years in order.
--   Rather you are grouping for certain components regardless of which year they belonged in.









-- this is code without any trunction
-- Query 1:
    SELECT occurred_at , SUM(standard_qty) AS standard_qty_sum
        FROM orders
        GROUP BY occurred_at
        ORDER BY occurred_at;

SELECT occurred_at , SUM(standard_qty) AS standard_qty_sum ,
        DATE_TRUNC('day',occurred_at) AS day_trunc
--         ,TO_CHAR(day_trunc) AS day_trunc_char
        FROM orders
        GROUP BY occurred_at
        ORDER BY occurred_at;

    SELECT DATE_TRUNC('day',occurred_at) , SUM(standard_qty) AS standard_qty_sum
        FROM orders
        GROUP BY DATE_TRUNC('day',occurred_at)
        ORDER BY DATE_TRUNC('day',occurred_at);

-- DATE_PART can be useful for pulling a specific portion of a date
--
-- but notice pulling month or day of the week (dow) ENUMURATION 0--->>6   sun--->>sat
--
-- means that you are no longer keeping the years in order.
--
-- Rather you are grouping for certain components regardless of which year they belonged in.

-- Query 2:
    SELECT DATE_PART('dow',occurred_at) AS day_of_week , SUM(total) AS total_qty
        FROM orders
        GROUP BY day_of_week
        ORDER BY total_qty;

/*
```

```
    Quiz: DATE Functions
    Questions: Working With DATEs
        Use the SQL environment below to assist with answering the following questions.
        Whether you get stuck or you just want to double-check your solutions,
        my answers can be found at the top of the next concept.
*/
--
-- Find the sales in terms of total dollars for all orders in each year,
-- ordered from greatest to least. Do you notice any trends in the yearly sales totals?
-- code
SELECT DATE_TRUNC('year',occurred_at) AS yearly_sales ,
           SUM(orders.total_amt_usd) AS total_dollars
        FROM orders
        GROUP BY yearly_sales
        ORDER BY total_dollars  DESC;

SELECT DATE_PART('year', occurred_at) ord_year,  SUM(total_amt_usd) total_spent
    FROM orders
    GROUP BY 1
    ORDER BY 2 DESC;


-- Which month did Parch & Posey have the greatest sales in terms of total dollars?
-- Are all months evenly represented by the dataset?
-- code
SELECT DATE_TRUNC('month',occurred_at) AS month_sales ,
           SUM(orders.total_amt_usd) AS total_dollars
        FROM orders
        GROUP BY month_sales
        ORDER BY total_dollars  DESC;

SELECT DATE_PART('month', occurred_at) ord_month, SUM(total_amt_usd) total_spent
    FROM orders
    GROUP BY 1
    ORDER BY 2 DESC;

-- Which year did Parch & Posey have the greatest sales in terms of the total number of orders?
-- Are all years evenly represented by the dataset?
-- code
SELECT DATE_PART('year', occurred_at) ord_year,  COUNT(*) total_sales
    FROM orders
    GROUP BY 1
    ORDER BY 2 DESC;


-- Which month did Parch & Posey have the greatest sales in terms of the total number of orders?
-- Are all months evenly represented by the dataset?

-- In which month of which year did Walmart spend the most on gloss paper in terms of dollars?
-- code
SELECT DATE_TRUNC('month', o.occurred_at) ord_date, SUM(o.gloss_amt_usd) tot_spent
    FROM orders o
    JOIN accounts a
        ON a.id = o.account_id
    WHERE a.name = 'Walmart'
    GROUP BY 1
    ORDER BY 2 DESC
    LIMIT 1;

-- ================================================================================
-- CASE - Expert Tip
--  The CASE statement always goes in the SELECT clause.
--
--

-- CASE must include the following components:
-- WHEN, THEN, and END. ELSE is an optional component to catch cases that didn't meet any of the other previous CASE conditions.
-- --
```

```sql
-- Query 1:
    SELECT id , account_id , occurred_at , channel,
           CASE WHEN channel = 'facebook' THEN 'yes'
                END AS is_facebook
        FROM web_events
        ORDER BY occurred_at;

-- Query 2:
    SELECT id , account_id , occurred_at , channel ,
           CASE WHEN channel = 'facebook' THEN 'yes'
                ELSE 'no'
                END AS is_facebook
        FROM web_events
        ORDER BY occurred_at;

-- Query 3:
SELECT id , account_id , occurred_at , channel,
       CASE WHEN channel = 'facebook' OR channel = 'direct' THEN 'yes'
            ELSE 'no'
            END AS is_facebook
    FROM web_events
    ORDER BY occurred_at;

-- Query 4:
SELECT account_id , occurred_at , total,
       CASE WHEN total > 500 THEN 'Over 500'
            WHEN total > 300 THEN '301 - 500'
            WHEN total > 100 THEN '101 - 300'
            ELSE '100 or under' END AS total_group
    FROM orders ;


SELECT
        CASE WHEN total > 500 THEN 'OVer 500'
        ELSE '500 or under'
        END AS total_group, COUNT(*) AS order_count
    FROM orders
    GROUP BY 1;



SELECT COUNT(1) AS orders_over_500_units
    FROM orders
    WHERE total > 500;

/*
    Quiz: CASE
    Questions: CASE
        Use the SQL environment below to assist with answering the following questions.
        Whether you get stuck or you just want to double-check your solutions,
        my answers can be found at the top of the next concept.
*/

-- Write a query to display for each order, the account ID, the total amount of the order, and the level of the order - ‘Large’ or ’Small’ - depending on if the order is $3000 or more, or smaller than $3000.
-- query to display the number of orders in each of three categories, based on the total number of items in each order. The three categories are: 'At Least 2000', 'Between 1000 and 2000' and 'Less than 1000'.
-- code
SELECT
    CASE WHEN total >= 2000 THEN 'At Least 2000'
        WHEN total >= 1000 AND total < 2000 THEN 'Between 1000 and 2000'
        ELSE 'Less than 1000' END AS order_category,
    COUNT(*) AS order_count
FROM orders
GROUP BY 1;

-- We would like to understand 3 different levels of customers based on the amount associated with their purchases.
-- The top-level includes anyone with a Lifetime Value (total sales of all orders) greater than 200,000 usd.
```

```sql
-- The second level is between 200,000 and 100,000 usd. The lowest level is anyone under 100,000 usd.
-- Provide a table that includes the level associated with each account. You should provide the account name, the total sales of all orders for the customer, and the level.
-- Order with the top spending customers listed first.
-- code
SELECT a.name, SUM(total_amt_usd) total_spent,
    CASE WHEN SUM(total_amt_usd) > 200000 THEN 'top'
        WHEN  SUM(total_amt_usd) > 100000 THEN 'middle'
        ELSE 'low' END AS customer_level
    FROM orders o
    JOIN accounts a
    ON o.account_id = a.id
    GROUP BY a.name
    ORDER BY 2 DESC;


-- We would now like to perform a similar calculation to the first,
-- but we want to obtain the total amount spent by customers only in 2016 and 2017.
-- Keep the same levels as in the previous question. Order with the top spending customers listed first.
-- code
SELECT a.name, SUM(total_amt_usd) total_spent,
     CASE WHEN SUM(total_amt_usd) > 200000 THEN 'top'
        WHEN  SUM(total_amt_usd) > 100000 THEN 'middle'
        ELSE 'low' END AS customer_level
    FROM orders o
    JOIN accounts a
    ON o.account_id = a.id
    WHERE occurred_at > '2015-12-31'
    GROUP BY 1
    ORDER BY 2 DESC;
-- We would like to identify top-performing sales reps, which are sales reps associated with more than 200 orders.
-- Create a table with the sales rep name, the total number of orders, and a column with top or not depending on
-- if they have more than 200 orders. Place the top salespeople first in your final table.
-- code



-- The previous didn't account for the middle, nor the dollar amount associated with the sales.
-- Management decides they want to see these characteristics represented as well.
-- We would like to identify top-performing sales reps,
-- which are sales reps associated with more than 200 orders or more than 750000 in total sales.
-- The middle group has any rep with more than 150 orders or 500000 in sales.
-- Create a table with the sales rep name, the total number of orders, total sales across all orders, and a column with top, middle, or low depending on these criteria.
-- Place the top salespeople based on the dollar amount of sales first in your final table. You might see a few upset salespeople by this criteria!
-- code
```