```sql
-- Lesson Overview
-- Basic SQL Lesson Overview
/*
    In this lesson, we will cover and you will be able to:

    Describe why SQL is important
    Explain how SQL data is stored and structured
    Create SQL queries using proper syntax including
        SELECT & FROM
        LIMIT
        ORDER BY
        WHERE
        Basic arithmetic operations
        LIKE
        IN
        NOT
        AND & BETWEEN & OR
There is a lot to cover so let's get started!

*/
/*
Parch & Posey Database:

    In this course, we will mostly be using the Parch & Posey database for our queries.
    Whenever we use a different database, we will let you know.

    Parch & Posey (not a real company) is a paper company and the database
    includes sales data for their paper.

    Using the sales data, you'll be able to put your SQL skills
    to work with data you would find in the real world.
*/
/*
Entity Relationship Diagrams:

An entity-relationship diagram (ERD) is a common way to view data in a database.
Below is the ERD for the database we will use from Parch & Posey.
These diagrams help you visualize the data you are analyzing including:

    The names of the tables.
    The columns in each table.
    The way the tables work together.
*/
/*
There are some major advantages to using traditional relational databases,
which we interact with using SQL. The five most apparent are:

    SQL is easy to understand.
    Traditional databases allow us to access data directly.
    Traditional databases allow us to audit and replicate our data.
    SQL is a great tool for analyzing multiple tables at once.
    SQL allows you to analyze more complex questions than dashboard tools like Google Analytics.
*/
/*
A few key points about data stored in SQL databases:

    Data in databases is stored in tables that can be thought of just like Excel spreadsheets.
        For the most part, you can think of a database as a bunch of Excel spreadsheets.
        Each spreadsheet has rows and columns.
        Where each row holds data on a transaction, a person, a company, etc.,
        while each column holds data pertaining to a particular aspect of one of the rows
        you care about like a name, location, a unique id, etc.

    All the data in the same column must match in terms of data type.
        An entire column is considered quantitative, discrete, or as some sort of string.
        This means if you have one row with a string in a particular column,
        the entire column might change to a text data type. This can be very bad
        if you want to do math with this column!

    Consistent column types are one of the main reasons working with databases is fast.
    Often databases hold a LOT of data. So, knowing that the columns are all
    of the same types of data means that obtaining data from a database can still be fast.
*/
/*
The key to SQL is understanding statements. A few statements include:

    CREATE TABLE is a statement that creates a new table in a database.

    DROP TABLE is a statement that removes a table in a database.

    SELECT allows you to read data and display it. This is called a query.

        The SELECT statement is the common statement used by analysts,
        and you will be learning all about them throughout this course!
*/

/*
    SQL command that will be used in every query:
    SELECT---FROM---
*/

/*
SELECT : indicates which column(s) you want to be given the data for.
FROM   : specifies from which table(s) you want to select the columns.
            Notice the columns need to exist in this table.
*/

--If you want to be provided with the data from all columns in the table,
-- you use "*", like so:

SELECT *
    FROM orders
    LIMIT 10;

-- SELECT does not create a new table with these columns in the database
-- SELECT just provides the data to you as the results, or output, of this command.

/*
    Your Turn
    Try writing your own query to select only the id, account_id, and occurred_at
    columns for all orders in the orders table.
*/
--code
SELECT id, account_id,occurred_at
    FROM orders

/*
    LIMIT to see just the first few rows of a table
    It is much faster for loading than if we load the entire dataset.
    Syntax:
            LIMIT <num.of.row>
*/

SELECT id,account_id,occurred_at
    FROM orders
    LIMIT 10;

-- Avoid Spaces in Table and Variable Names
/*
It is common to use underscores and avoid spaces in column names.
It is a bit annoying to work with spaces in SQL.
In Postgres, if you have spaces in column or table names, you need to refer to these columns/tables with double quotes around them
(Ex: FROM "Table Name" as opposed to FROM table_name).
In other environments, you might see this as square brackets instead (Ex: FROM [Table Name]).
*/

-- Quiz: LIMIT
```

```sql

/*
    Try using LIMIT yourself below by writing a query that displays all the data
    in the occurred_at, account_id, and channel columns
    of the web_events table,
    and limits the output to only the first 15 rows.
*/

SELECT occurred_at, account_id, channel
    FROM web_events
    LIMIT 15;

/*
    ORDER BY statement allows us to sort our results using the data in any column.
    Pro-Tip :
        Remember DESC can be added after the column in your ORDER BY statement
        to sort in descending order, as the default is to sort in ascending order.
*/
--using ORDER BY in a SQL query only has temporary effects,
--for the results of that query, unlike sorting a sheet by column in Excel or Sheets.

SELECT *
    FROM orders
    ORDER BY account_id DESC
    LIMIT 10 ;

/*
    Quiz: ORDER BY :
        Practice :
            Let's get some practice using ORDER BY:

        1.  Write a query to return the 10 earliest orders in the orders table.
            Include the id, occurred_at, and total_amt_usd.

        2.  Write a query to return the top 5 orders in terms of
            the largest total_amt_usd.
            Include the id, account_id, and total_amt_usd.

        3.  Write a query to return the lowest 20 orders in terms of
            the smallest total_amt_usd.
            Include the id, account_id, and total_amt_usd.
*/

SELECT id , occurred_at , total_amt_usd
    FROM orders
    ORDER BY occurred_at
    LIMIT 10;

SELECT id , account_id , total_amt_usd
    FROM orders
    ORDER BY total_amt_usd DESC
    LIMIT 5;

SELECT id , account_id , total_amt_usd
    FROM orders
    ORDER BY total_amt_usd
    LIMIT 20;

/*
    Here, we saw that we can ORDER BY more than one column at a time.
    When you provide a list of columns in an ORDER BY command,
    the sorting occurs using the leftmost column in your list first,
    then the next column from the left, and so on.
    We still have the ability to flip the way we order using DESC.
*/

SELECT  account_id , total_amt_usd
    FROM orders
    ORDER By total_amt_usd DESC, account_id ;
```

```sql


/*
    This query selected account_id and total_amt_usd from the orders table,
    and orders the results first by total_amt_usd in descending order and then
    account_id.
*/

/*
    Quiz: ORDER BY Part II:
    Questions:

    1.  Write a query that displays the order ID, account ID, and
        total dollar amount for all the orders,
        sorted first by the account ID (in ascending order),
        and then by the total dollar amount (in descending order).

    2.  Now write a query that again displays order ID, account ID, and
        total dollar amount for each order,
        but this time sorted first by total dollar amount (in descending order),
        and then by account ID (in ascending order).

    3.  Compare the results of these two queries above.
        How are the results different when you switch the column you sort on first?
*/

SELECT id , account_id , total_amt_usd
    FROM orders
    ORDER BY account_id , total_amt_usd DESC;

SELECT id , account_id , total_amt_usd
    FROM orders
    ORDER BY total_amt_usd DESC , account_id ;

/*
    Compare the results of these two queries above.
    How are the results different when you switch the column you sort on first?

    In query #1, all of the orders for each account ID are grouped together,
    and then within each of those groupings, the orders appear from the greatest
    order amount to the least.

    In query #2, since you sorted by the total dollar amount first,
    the orders appear from greatest to least regardless of which account ID
    they were from. Then they are sorted by account ID next.
    (The secondary sorting by account ID is difficult to see here since only
        if there were two orders with equal total dollar amounts would
        there need to be any sorting by account ID.)
*/

-- WHERE
/*
    WHERE statement, we can display subsets of tables based on conditions
    that must be met.

    WHERE command as filtering the data.

    Common symbols used in WHERE statements include:

        1. > (greater than)

        2. < (less than)

        3. >= (greater than or equal to)

        4. <= (less than or equal to)

        5. = (equal to)

        6. != (not equal to)
*/

SELECT *
    FROM orders
    WHERE account_id = 4251
    ORDER BY occurred_at
    LIMIT 1000;

/*
    Quiz: WHERE
        Questions:
            Write a query that:

            Pulls the first 5 rows and all columns from the orders table that
            have a dollar amount of gloss_amt_usd greater than or equal to 1000.

            Pulls the first 10 rows and all columns from the orders table that
            have a total_amt_usd less than 500.
*/

SELECT *
    FROM orders
    WHERE gloss_amt_usd >= 1000
    LIMIT 5;

SELECT *
    FROM orders
    WHERE total_amt_usd < 500
    LIMIT 10;

/*
    The WHERE statement can also be used with non-numeric data.
    We can use the <=> and <!=> operators here.
    You need to be sure to use single quotes
        (just be careful if you have quotes in the original text)
        with the text data, not double quotes.
*/

-- Query 1

SELECT *
    FROM accounts
    WHERE name = 'United Technologies';
--
-- Query 2
--
SELECT *
    FROM accounts
    WHERE name != 'United Technologies';

/*
    Commonly when we are using WHERE with non-numeric data fields,
    we use the LIKE, NOT, or IN operators.
    We will see those before the end of this lesson!
*/

/*
Quiz: WHERE with Non-Numeric
    Practice Question Using WHERE with Non-Numeric Data:

    Filter the accounts table to include
    the company name, website, and the primary point of contact (primary_poc)
    just for the Exxon Mobil company in the accounts table.
*/

SELECT name , website , primary_poc
    FROM accounts
    WHERE name = 'Exxon Mobil';
```

```sql


-- Derived Columns
    -- Creating a new column that is a combination of existing columns

-- you want to give a name, or "alias," to your new column using the AS keyword.

-- This derived column, and its alias, are generally only temporary,
--  existing just for the duration of your query.

-- The next time you run a query and access this table,
-- the new column will not be there.

-- Arithmetic Operators

/*
    If you are deriving the new column from existing columns using
    a mathematical expression, then these familiar
    mathematical operators will be useful:

    * (Multiplication)
    + (Addition)
    - (Subtraction)
    / (Division)
*/

SELECT id, (standard_amt_usd/total_amt_usd)*100 AS std_percent, total_amt_usd
    FROM orders
    LIMIT 10;

SELECT account_id,
       occurred_at,
       standard_qty,
       gloss_qty + poster_qty AS nonstandard_qty
FROM orders;

-- Quiz: Arithmetic Operators

/*
    Q1
    Create a column that divides the standard_amt_usd by
    the standard_qty to find the unit price for standard paper for each order.
    Limit the results to the first 10 orders,
    and include the id and account_id fields.
*/

--code
SELECT id, account_id, standard_amt_usd/standard_qty AS unit_price
    FROM orders
    LIMIT 10;

/*
    Q2
    Write a query that finds the percentage of revenue that comes from
    poster paper for each order. You will need to use only the columns that
    end with _usd. (Try to do this without using the total column.)
    Display the id and account_id fields also.
*/

--code
SELECT id, account_id,
        poster_amt_usd/(standard_amt_usd + gloss_amt_usd + poster_amt_usd)
        AS post_per
    FROM orders
    LIMIT 10;

/*
Introduction to Logical Operators:

    In the next concepts, you will be learning about Logical Operators.
    Logical Operators include:

    1.  LIKE This allows you to perform operations similar to using WHERE and =
        but for cases when you might not know exactly what you are looking for.

    2.  IN This allows you to perform operations similar to using WHERE and =
        but for more than one condition.

    3.  NOT This is used with IN and LIKE to select all of
        the rows NOT LIKE or NOT IN a certain condition.

    4.  AND & BETWEEN These allow you to combine operations
        where all combined conditions must be true.

    5.  OR This allows you to combine operations where
        at least one of the combined conditions must be true.
*/

-- The LIKE operator is extremely useful for working with text.
-- You will use LIKE within a WHERE clause.

-- The LIKE operator is frequently used with %.
-- The % tells us that we might want any number of characters leading up to a particular set of characters

SELECT *
    FROM accounts
    WHERE website LIKE '%google%';

--  Quiz: LIKE
-- Questions using the LIKE operator

-- Use the accounts table to find
-- All the companies whose names start with 'C'.

--code
SELECT *
    FROM accounts
    WHERE accounts.name LIKE '%C%';


-- Use the accounts table to find
-- All companies whose names contain the string 'one' somewhere in the name.

--code
SELECT *
    FROM accounts
    WHERE accounts.name LIKE '%one%';


-- Use the accounts table to find
-- All companies whose names end with 's'.

--code
SELECT *
    FROM accounts
    WHERE accounts.name LIKE '%s';

--IN
-- The IN operator is useful for working with both numeric and text columns.
-- This operator allows you to use an =, but for more than one item of that particular column.

-- We can check one, two, or many column values for which we want to pull data,
-- but all within the same query.

SELECT *
    FROM orders
    WHERE account_id IN (1001,1021);
```

```sql
SELECT *
    FROM accounts
    WHERE accounts.name IN ('Apple','Walmart');

-- Quiz: IN :
-- Questions using IN operator :

-- Use the accounts table to find
--  the account name, primary_poc, and sales_rep_id for Walmart, Target, and Nordstrom.

--CODE
SELECT accounts.name , accounts.primary_poc , accounts.sales_rep_id
    FROM accounts
    WHERE accounts.name IN ('Walmart','Target','Nordstrom');


-- Use the web_events table to find all information regarding individuals
-- who were contacted via the channel of organic or adwords.

--CODE
SELECT *
    FROM web_events
    WHERE web_events.channel IN ('organic','adwords');

/*
    The NOT operator is an extremely useful operator for working with the previous two operators
    we introduced: IN and LIKE. By specifying NOT LIKE or NOT IN,
    we can grab all of the rows that do not meet particular criteria.
*/

SELECT sales_rep_id , name
    FROM accounts
    WHERE sales_rep_id NOT IN (321500,321570)
    ORDER BY sales_rep_id

-- Code from the video has been modified to match our database schema in the workspaces.

SELECT *
    FROM accounts
    WHERE website NOT LIKE '%com%';

-- Quiz: NOT
-- Questions using the NOT operator
--
-- We can pull all of the rows that were excluded from the queries
-- in the previous two concepts with our new operator.

/*
    Use the accounts table to find:

        All the companies whose names do not start with 'C'.
        All companies whose names do not contain the string 'one' somewhere in the name.
        All companies whose names do not end with 's'.
*/
SELECT name
    FROM accounts
    WHERE name NOT LIKE 'C%';

SELECT name
    FROM accounts
    WHERE name NOT LIKE '%one%';

SELECT name
    FROM accounts
    WHERE name NOT LIKE '%s';

-- Use the accounts table to find
-- the account name, primary poc, and sales rep id
--  for all stores except Walmart, Target, and Nordstrom.
```

```sql

--CODE
SELECT accounts.name , accounts.primary_poc , accounts.sales_rep_id
    FROM accounts
    WHERE name NOT IN ('Walmart', 'Target', 'Nordstrom');

-- Use the web_events table to find all information regarding individuals
-- who were contacted via any method except using organic or adwords methods.

--CODE
SELECT *
    FROM web_events
    WHERE channel NOT IN ('organic', 'adwords');

-- The AND operator
--  is used within a WHERE statement to consider more than one logical clause at a time.

SELECT *
    FROM orders
    WHERE occurred_at >= '2016-04-01' AND occurred_at <= '2016-10-01'
    ORDER BY occurred_at;

-- BETWEEN Operator
--  Sometimes we can make a cleaner statement using BETWEEN than we can use AND.
--  Particularly this is true when we are using the same column for different parts of our AND statement.

SELECT *
    FROM orders
    WHERE occurred_at BETWEEN '2016-04-01' AND '2016-10-01'
    ORDER BY occurred_at;

-- Quiz: AND and BETWEEN
-- Questions using AND and BETWEEN operators

/*
    Write a query that returns all the orders where the standard_qty is over 1000,
    the poster_qty is 0, and the gloss_qty is 0.
*/
--code
SELECT *
    FROM orders
    WHERE orders.standard_qty > 1000 AND orders.gloss_qty = 0 AND orders.poster_qty = 0;

/*
    Using the accounts table, find all the companies whose names do not start with 'C' and end with 's'.
*/
SELECT *
    FROM accounts
    WHERE accounts.name NOT LIKE 'C%s';

SELECT *
    FROM accounts
    WHERE accounts.name LIKE 'C%s';

/*
    When you use the BETWEEN operator in SQL,
    do the results include the values of your endpoints, or not?
    Figure out the answer to this important question by writing a query
    that displays the order date and gloss_qty data for all orders
    where gloss_qty is between 24 and 29.
    Then look at your output to see if the BETWEEN operator included the begin and end values or not.
*/
SELECT orders.occurred_at , orders.gloss_qty
    FROM orders
    WHERE orders.gloss_qty NOT BETWEEN 24 AND 29 ;

/*
    Use the web_events table to find all information regarding
    individuals who were contacted via the organic or adwords channels,
    and started their account at any point in 2016, sorted from newest to oldest.
*/
SELECT *
    FROM web_events
    WHERE web_events.channel IN ('adwords','organic')
     AND web_events.occurred_at BETWEEN '2016-01-01' AND '2017-01-01'
    ORDER BY web_events.occurred_at DESC;

--OR
--  it can be combine with other operators

SELECT account_id , occurred_at , standard_qty , gloss_qty , poster_qty
    FROM orders
    WHERE standard_qty = 0 OR gloss_qty = 0 OR poster_qty = 0;


SELECT account_id , occurred_at , standard_qty , gloss_qty , poster_qty
    FROM orders
    WHERE (standard_qty = 0 OR gloss_qty = 0 OR poster_qty = 0)
        AND occurred_at = '2016-10-01';


-- Quiz: OR
-- Questions using the OR operator

/*
    Find list of orders ids where either gloss_qty or poster_qty is greater than 4000.
    Only include the id field in the resulting table.
*/
--code
SELECT id
    FROM orders
    WHERE gloss_qty > 4000 OR poster_qty > 4000;


/*
    Write a query that returns a list of orders where the standard_qty is zero
    and either the gloss_qty or poster_qty is over 1000.
*/
--code
SELECT *
    FROM orders
    WHERE standard_qty = 0 AND (gloss_qty > 1000 OR poster_qty > 1000);


/*
    Find all the company names that start with a 'C' or 'W',
    and the primary contact contains 'ana' or 'Ana', but it doesn't contain 'eana'.
*/
--code
SELECT accounts.name
    FROM accounts
    WHERE (accounts.name LIKE 'C%' OR accounts.name LIKE 'W%')
        AND (accounts.primary_poc LIKE '%ana%' OR accounts.primary_poc LIKE '%Ana%')
        AND accounts.primary_poc NOT LIKE '%eana%';
```