Let’s go over some practice queries using functions. Now remember functions are kind of a premade, pre-coded bit of code that we don't have to really worry about. Someone has already created it and named it; given a name. What we do is we’ll give our data as input so that, that function can do something, process it, do something with it, format it whatever that might be without us having to really worry about it. We just have to know the name of it and give it the input. Then we can get the output that we desire. We have to choose the right function for what output we need. And we have to input to give it. That input is called parameters. Let’s practice with those and we can see how those work. I'm going to be using the bike database.

```
USE bike;
```

I'm going to run that and make sure my bike is my current database. Let’s just take a look at everything that’s in the store table.

```
SELECT * FROM store;
```

We have three stores, their name, number, email, street, city, state, ZIP and they each have the primary key 1, 2, 3. Let's start off with the CONCAT function so in this instance we want the output of that function to show up as a column in our result set table. So I'm going to just put it inside the select statement for this one. What the CONCAT function does is, it takes all the parameters and just put them together. It stands for concatenation which is just putting them together. If I wanted to make a sentence, well let’s just start with to columns for now. We're going to type in two column names and email. So those are two parameters. We have parens that always follow a function name. The function name is CONCAT. The parens always follow a function name. Sometimes the parens hold parameters. So this is the input as going into our function, the store name and the email we know that the CONCAT function is pre-made code that someone put together. What it’s going to do is take these two inputs, the two parameters that are separated by a comma and it's going to put them together.

```
SELECT CONCAT(store_name, email)
FROM store;
```

So we’ll put what table those columns are from. Now and if I run this now you'll see it threw together and it was very literally together by the name and the email with absolutely no space in between; shoved them together. If you did want to add a space with a CONCAT, you can have number of different parameters. You could put empty quotes with a space in between and that would give you a little bit of a space between those two. You could even throw in other strings that you choose. ‘The’ with a space after it and I have to put commas in between all these, The store name and then could go something like ‘email is’. You could even add a period at the end and whatever you wanted to do and it will make some output that looks like that.

```
SELECT CONCAT('The ', store_name, ' email is ', email)
```

FROM store;

Notice it's got kind of crazy heading name how because it’s just going to use the whole entire function and all the parameters as the heading name. Showing everything that's there. This would be a good place for an alias--’store’s email’ or something like that.

SELECT CONCAT('The ', store_name, ' email is ', email) **AS ‘Store’s Email’**
FROM store;

Notice how it thinks I’ve ended this right here because I had a possessive quote in there. So we do have to either use the backslash to ignore that out. Put it right before the one that you want to ignore or I could have used double quotes on the outside and single quotes on the inside and that would have worked as well.

SELECT CONCAT('The ', store_name, ' email is ', email) **AS “Store’s Email”**
FROM store;

Remember this is a very literal system that takes exactly what you do and it doesn't understand that you wanted that as an apostrophe so you have to be very careful. Then you would get nicer meaning heading there. That was the CONCAT function.

If we were to add something like the stores or the email is or maybe we want to put ‘the store number is’ and we put store_id, which is not the email anymore for our Alias. If we would run that it now becomes the output of this function now becomes a string even though store ID was an integer to begin with, it's going to be concatenated in with all the rest of the strings and it's going to become a part of that string. So not only does it put different things together it will change the datatype sometimes. So when you send parameters in as a certain data type in some functions, it may output as a different datatype. So keep that in mind as you're running some of these functions.

All right, let's try the left function. And what the left function is going to do is it's going to return a number of characters from the left of a string. So we had all those bike product names, all the names of the bikes, those big long names. Let's just look at those product names really fast so we can see what they look like from the product table. They were these long names with the years at the end, the name of the brand at the beginning. So if we wanted to just grab a certain number of characters from the left of that string, we can say I want to get the left side of the string product name and I want to return 15 characters.

SELECT LEFT(product_name, 15)
FROM product;

You can run that and you can see you get 15 characters back. Even that space in between the two words is counted as one of those characters. Apostrophes, special symbols, anything that taking up a spot would count. So it’s going to get 15 from the left. Now you're wondering why

would you do that? If you had column names that you knew the exact numbers between words that would work but you sometimes will use functions inside of other functions. So this function may become a parameter for a different function, so they're nested inside of each other. This may be one that you might use with another function as well. There's also a similar one that is RIGHT and it will go to the right or the end of the string and return 15 characters coming backward through the strings. So it goes to the end and counts 15 places, the spaces, the dashes, the apostrophe, everything counts in those 15. And it will return that. Some of these functions that I run today are not very logical, like why would you do that, I don't see the purpose of that. But again you may use this with a different function. Or you may have columns that are more uniform in length that this might work with. Today our purpose is just to understand what the function is doing even if it may not be very logical for us.

There's also one called trim, ltrim, and rtrim. Sometimes data goes into the database with spaces before and after whatever they entered. And in order to use it as output for your program or wherever that output is going, you want to make sure all those leading and trailing spaces, beginning and ending spaces, are eliminated. So we're going to try one called trim. And since we don't really have spaces in any of the bike databases I'm going to go ahead and just type in my own little string here and I'm going to purposely put two spaces. And I'm going to say this little string, put a couple more spaces and that's going to be my string. I've got two leading spaces two trailing spaces here on this one. And I don't have to put a table name here because it's not really coming for from any table just a string.

SELECT TRIM(' this is it ');

If I run this it will take off both the two spaces at the beginning and the two spaces at the end. It's kind of hard to see that. You can kind of see it beginning there's no spaces but that is what it's doing. If I said I just want to trim off the right or trailing spaces you'll see you can kind of see those two spaces and the end one would be trimmed off.

SELECT **R**TRIM(' this is it ');

And then **L**TRIM just takes it away from the leading or beginning of the string.

SELECT LTRIM(' this is it ');

It's kind of hard to see that so let's combine this with another function and let's just look at it this new function first before we combine them. If I say length here, I can find the length of everything that's in this string. Again if it was a column name it would give the length of all those in there. So it's counting the two spaces here and then the one two three four five six seven eight nine 10 11 12 spaces onto those two, which give us the length of 14 characters in there.
SELECT **LENGTH**(' this is it ');

Now if I go in and I say now I'm going to use this with the TRIM function nested inside of the LENGTH function. Notice I've got to have a set of parens for each function, which gives me a double set here one for LENGTH and one for TRIM.

SELECT LENGTH(**TRIM**(' this is it '))**;**

And I run this it will give me a length of 10 because it trimmed off the two at the beginning and the end. The inside nested one would run first, trimming up that string, then it would count the length of what was left over after it was trimmed and give me 10. So you can see how you can nest functions inside of other functions and have them work for you. And then again if we do LTRIM.

SELECT LENGTH(**L**TRIM(' this is it '));

You can see it brings back the two, It will still count the two on the right because it was only trimmed on the left. Then with RTRIM it gives us the same because now it's counting the two on the left. Because we trimmed off the ones on the right.

SELECT LENGTH(**R**TRIM(' this is it '))**;**

So there was TRIM, LTRIM, RTRIM, and LENGTH.

There's another set that kind of goes together. That's the lower and the upper. So this time let's use a column name. If we look at all of those store names that we had before from the store table. We see that it's got like initial caps. The first of each word is initial or is capital and the rest are underscore or lowercase. So if we use the function UPPER and put in the input of the store name as the parameter, it will make every single character in that whole string an uppercase character. So again the data may have gone into the database in lower case. Maybe you had a two digit state that was reduced down to VARCHAR(2) or CHAR(2) to keep it to those two characters but you didn't specify that you had to have an upper case. So at this point when you want it to come out of the database maybe you want to make sure they're all very similar and all of them are capitals so you might run something like this as you bring that out of the database.

SELECT UPPER(store_name)
FROM store;

Lower is that the one that's similar but it will bring everything into lowercase even those initial caps are now lowercase.

SELECT **LOWER**(store_name)
FROM store;

So upper and lower will convert your string to be either uppercase or lowercase.

Let's do a little more complicated one, but it's really cool. So this one is called LOCATE. And I'm going to put a comment here just so we can see the parameters that are going in here. These are little placeholders here for me to remember what goes into LOCATE and what the parameters are doing. So the parameters here are something you'd have to memorize or look up. You're going to want to locate something inside of a string and know where that piece that you're locating started. So find is the first parameter. What do you want to find? And then where are you searching for this thing you're looking for? And at what position do you want to start looking for that. This one I'm going to put square brackets. You don't need to do square brackets in the actual function but I'm going to put square brackets just because this one is optional, you don't have to have this third parameter. But you would need to have these first two parameters for sure.

```
-- parameters for LOCATE(find, search, [start])
```

That's what we are looking at with the LOCATE function. We're finding where a certain character starts in another larger set of data. Alright so if we say something like SELECT the product name and we'll go ahead and just put that for now. From the product table. We have that big long name of the bike right? So if we wanted to go ahead and use the LOCATE function. It doesn't even have to be in the SELECT statement. You could do something like WHERE LOCATE the string girl. So some of those product names have the word girl in there because they were girl's bikes and it actually said in their name. And we want to search for that word girl. And we want to search for it in the product name and we could even put we want to start it first character. Or we could leave that off and we'd get the same thing because it would, by default, start at the beginning.

```
SELECT product_name
FROM product
WHERE LOCATE('Girl', product_name, 1);
```

If we run this, what's going to happen is we're going to get some product returned here. So it's going to show the whole product name, but the where is filtering down to when this condition is true meaning we got a number back from here. Let's put that same function inside of our select so we can see what that output looks like. Whatever is in the SELECT is what you're going to see in the result set. So what it's saying is, we are seeing the word girl at character number 9 for the first one. If I count over 9 characters 1 2 3 4 5 6 7 8 is the space 9 is where 'G' in girl starts. That is where it started to see that substring that I put in here right. So it says within the product name we found the girl starting at character 9. So that is what's happening here. So we're getting a positive number as a result back from this condition. You could also, if it makes more sense you could say, something like as long as it found something and it was greater than zero, they didn't find anything then show that one.

```
SELECT product_name '
```

```
FROM product
WHERE LOCATE('Girl', product_name, 1) > 0;
```

If I take the WHERE off completely, it's going to show everything where it found zero.

```
SELECT product_name, LOCATE('Girl', product_name, 1)
FROM product;
```

Some of them to not have the word girl in them at all so it returns a zero. And if we scroll down there is where it found that nine. And so that one did find something. So the WHERE was filtering out all those zeros basically. So this LOCATE returns, and I'm going to put it up here in a comment. 'Returns an integer' of where it's located in the number of characters in your data.

```
-- parameters for LOCATE(find, search, start) Returns an integer
```

All right, let's go ahead and do another one that's kind of fun. And that one is called SUBSTRING and I'm going to type in the parameters just little placeholders here, just so we know what's going on. So here again, this is the string that we are searching in. And where do we want to start our search at within that string. And then what length of characters are we going to return from that start point on. So this one's going to return an actual string to us.

```
-- parameters for SUBSTRING(string, start, length) Returns a string
```

So we're getting a piece of a string from another string. And exactly how much we get from that determines where we start and how many characters we are returning from that. We're going to go ahead and look at that product name again because we are pretty familiar with it at this point. And we're going to type in the name of the function and we're going to put as the first parameter what string are we looking at the product name. That whole big string, the whole entire name of the bike. Where do we want to start at? Let's start at character number 9 and return the next 6 characters after that position and this is the product table.

```
SELECT SUBSTRING(product_name, 9, 6)
FROM product;
```

So then it's going to go in and it's going to count 9 spaces. Remember some of these had the word Electra or Trex or something in front of them. So some of them had gone over 9 spaces. In fact let's put the product name in here so we can kind of see the original.

```
SELECT product_name, SUBSTRING(product_name, 9, 6)
FROM product;
```

So we're adding a new column. So it took out the nine before it got to the ninth space. The one two three four five six seven eight nine which is 'A' and then return six more spaces after it. So that's what we're getting from the substring. So again you're like what use it that? Again you

would probably use this is another function. Maybe you'll use the LEFT or the LOCATE or something as a function inside of another function. In fact let's try that. So this time we're going to combine these two and we're not only going to locate the integer where girl starts but we're going to try and return just the word, the 6 characters where the 'G" in girl starts. So we can go ahead and use this position which told us where that word girl started. I'm going to copy that and I'm going to replace the 2nd parameter with that entire function. And remember this returns an integer. So it's going to return zero if it doesn't have girl in the name. It's going to return 9 in some of those that found girl at the ninth space or 33 if it's found at the 33rd space. And then it's just going to return 6 characters after it starts finding that.

```
SELECT SUBSTRING(product_name, LOCATE('Girl', product_name, 1), 6)
FROM product;
```

So now if I run this you'll see that for a lot of them it doesn't return anything because we don't have a where in there. But for the ones that do have a substring in it, it returns the six characters the 'girl's' 1 2 3 4 5 6. So now we're getting the exact girl word out of there. Now you can start thinking of more logical ways that you might want to grab things out of other strings or use it in a WHERE clause so that you can just get those because you don't know where it is in the string. So you have to search for it and locate it and use that as a parameter of another function as the start position. Anyway, so you can see how putting these together using functions inside of other functions can start to get you very specific results back from your database, which is super cool.

So we looked a lot of the string functions which have to do with strings. Now let's look at some other types of functions. But before we get into those, I would now practice with yourself and just see if you could just do some of these things on your own. For example you could take the art database, the v_art database and look for the word 'woman' inside of all the artwork titles and see if you can actually return that word 'woman' and just kind of practice with this and see if you can figure some of these things our on your own. Just kind of play with them, that's how you are going to learn.

But let's go on now and look at some of the more numeric functions. So now we're looking at different functions that work with numbers. And I'm going to go back and make sure you're using the bike database

```
USE bike;
```

We're going to take a look at those prices again. Now we can do some math on some numbers. We have a list price from that product table that we can play around with.

```
SELECT list_price
FROM product;
```

We have all these prices of all these bikes. So we see that most of them end in 99 or 00. And we can now look at some of the different numeric functions. One of the numeric functions is the ROUND function. You'll notice that the first bike is 379.99.

SELECT **ROUND(**list_price**)**
FROM product;

Now if you round every single price, that's going to round up or down depending on what the decimal place is. But it since it was .99 it rounded up to 380. So there is this function. You can also place the -- right now the default is zero decimal places -- but you could say I want one decimal place.

SELECT **ROUND(**list_price, 1**)**
FROM product;

We could run that and it would give you one decimal place. And you could also give it two, which is actually going to put it back down where it was because if you round it by two decimal places, that's where it already was.

SELECT ROUND(list_price, **2**)
FROM product;

And then from 3 on up it would just be giving you zeros because we don't really have any more decimal places after that. But you can use a second parameter here that would give you number of decimal places to round to. That's how the ROUND function works. There's some that seem similar but they're a little bit different. We have one that's called CEILING and another that's called FLOOR. Now since these were all .999 doesn't work really well. So I'm just going to throw my own number in here. So if I say 12.2 -- and I don't really need a from since it's not really a table that we're looking at. It's going to make that go to the next highest whole number.

SELECT CEILING(12.2)
FROM product;

So that's what CEILING is going to do. Even though ROUND would have made it 12. Because it would have said it's only .2, it's .5 and below so I would have rounded that down to 12. But CEILING, no matter what the decimal place, even if it was 12.1, it's always going to go to the next highest whole number.
SELECT CEILING(12.**9**)
FROM product;

Now obviously .9 would still go to the next highest whole number. But FLOOR does the opposite. If it was 12.9 and I said FLOOR, it's always going to go down to the next whole number no matter how high this decimal place is, it's going to go down to the next lower number.

SELECT **FLOOR**(12.9)
FROM product;

So that's how FLOOR and CEILING will in case you don't want to round it, you want to go up or down then you use FLOOR and CEILING.

All right you can also format numbers. So what if we wanted these prices of the bikes to have a comma in them. Maybe with a dollar sign or something like. How can we make it look a little different? You can use the FORMAT function and list price would be the first parameter, what do you want to format, list price. You can also put the number of decimal places that you want formatted. Obviously currency is good with two decimal places. And we'll say, from product table. Now we'll see we get to two decimal places. While this is how it already was, but look at those that are over a thousand. We now have a comma at the thousands place.

SELECT FORMAT(list_price, 2)
FROM product;

So we know that our format is working. It's also giving us a comma at the thousands place. In different parts of the world commas might be in different areas or numbers might be formatted little different. So there is a third optional locale code that you can throw in there. For example, this one is English US.

SELECT FORMAT(list_price, 2, **'en_US'**)
FROM product;

So we use the comma at the thousand spot in the US. That give us the exact same results. But if there is a country that has a little different way to format numbers you can use that third parameter if you need to. Now if we wanted to dollar sign also here, it would simply just be concatenated on with the CONCAT and just put your dollar string as the string you want to add to the next part of the string. Just this whole thing here. Then you need to end your concatenate paren. Opening and closing concatenate and opening and closing for format. If you run that we now have a dollar sign at the beginning.

SELECT **CONCAT('$ ',** FORMAT(list_price, 2, 'en_US')**)**
FROM product;

You can even put a space before with the dollar sign whatever you wanted to do. So that's how you might make your currency look a little more like currency, if you needed to.

Let's look at some date functions. So say you had a date in your database but you wanted to just grab that part of that date. Like just the year from the date or just the month or just the day.

You can use these different date formats. For example, -- I was saying format, I meant function-- if you use the YEAR function, you can extract out just the date or just the year from the order date. So now we can run that from -- and this is actually in the customer order table.

SELECT order_date, YEAR(order_date)
FROM cust_order;

And it grabs out just the year part of that. So you see order date actually the whole thing with year, month, and day. And we extracted out just the year part of that. If we want to grab just the month part of it we can put MONTH.

SELECT order_date, **MONTH**(order_date)
FROM cust_order;

If we want to extract just the day part of it, we can put DAY and those parts will get extracted out for us.

SELECT order_date, **DAY**(order_date)
FROM cust_order;

So if we don't want to deal with the whole date, we want to just look at different portions of it, that's what you'd use. The year, date, and month functions.

Let's do another date one. This one's cool so you can get your current date as of right now, today by using the NOW function.

SELECT NOW();

So this is just coming from my computer system's clock. And it knows at the time I recorded this it was June 12th at this exact time. So when you run it, it will have your date of today. So noticed here's a function without any parameters but it still has those parens so it still is a function called NOW. If I wanted to now extract out just maybe the hour from now, you can do that with time as well.

SELECT **HOUR(**NOW()**)**;

Put an ending paren for HOUR, I can get just -- it's now 12:00. I could have also gotten the year out of there.

SELECT **YEAR**(NOW());

The year I did this was 2020. So there we go. So a function inside of another function. This one's getting the entire state and time string. This one is extracting the portions out that we wanted to. In this case, the year.

What if we plan to figure out the difference between two dates, like the customers ordered this date and how many days does it take for it to actually ship out? So here's the order date and here's the ship date, how can we find the difference between the two. There is a function called DATEDIFF--all one word no spaces and no underscore. And we're going to put the later date first if you want a positive number. We want to find out the difference between the date it actually shipped and the date that they actually ordered their order. And this is again from the customer order table.

SELECT DATEDIFF(shipped_date, order_date)
FROM cust_order;

And what that's going to give us is the number of days that there are different between those two. So they ordered one day and then two days later, in this case, it was shipped out. So 3, looks like 3 is kind of our longest one, so it takes us about 1 to 3 days to get an order shipped out into the mail. So we're seeing the difference in days so DATEDIFF will return the difference in days. What if this was a much later date like today and we run that.

SELECT DATEDIFF(**NOW()**, order_date)
FROM cust_order;

It's still going to give me the difference in days but you'll see that's like many years different right? So in order to get the difference in year you would have to divide that by the number of days in a year.

SELECT DATEDIFF(NOW(), order_date)/365
FROM cust_order;

And it would say it's been like 4.5 years, 4.4 years, since they ordered--between now and when they ordered. Because this database had some pretty old dates in it. So that is what DATEDIFF is doing. You could even type in a date and see the difference there as well.

SELECT DATEDIFF(**'2020-06-17'**, order_date)/365
FROM cust_order;

I put my today's date there, but you can put any date. So that's DATEDIFF, the difference in days between two dates. If I reversed those and put the later day second, as the second parameter, I will get a negative number.

SELECT DATEDIFF(**order_date, '2020-06-17'**)/365
FROM cust_order;

Which is fine. It's the exact same difference but it's just like subtracting a larger number from a smaller number, you're going to a negative. I usually try to think the later date is going to go first if I want a positive number. So that's how DATEDIFF works.

Let's look at DATE_ADD. So what if we wanted to add a particular interval of time to a date? For example once their invoice hits 30 days we want to charge them extra interest. Or within a week we want to make sure that they are warned that they haven't paid yet, or whatever it might be. So we're going to go ahead and look at the difference in interval times there. So we're going to say SELECT the order date and we're going to actually just put that there so we can see it--so we can see the difference. Now here's where our function starts. It's DATE_ADD and this time you put an underscore in it. So remember, it's not the same as DATEDIFF in syntax wise. You're going to actually have an underscore between there. The first parameter is what date are you going to be adding something to, the order date. And we want to see what the date is in nine days after the order date because we're going to do something nine days after they order. In case it hasn't shipped maybe we'll do something or whatever. So we say keyword interval which you always need with DATE_ADD as the second parameter. Then how many days do we want to add to it? Nine days and remember no 's' just 9 DAY from the customer order table.

SELECT order_date, DATE_ADD(order_date, INTERVAL 9 DAY)
FROM cust_order;

If you look at the days we have for January 1st changes to January 10th. It adds nine days on to it. You could also add 9 years ago. So it's going to go from 2016 to 2025 because we added 9 years on to it. You could also do month--you could add 9 months to it. So it goes from January to October. So you can add an interval of time onto a date so you can see when that would be.

SELECT order_date, DATE_ADD(order_date, INTERVAL 9 **MONTH**)
FROM cust_order;

Alright almost done here, just a few more. We're going to look at the date format next. For a date format, this just means that--so far we've seen the date in year, year, year, year dash month, month dash day, day. And that's the only format we've ever seen it in. What if we want to see the date differently. We want to make it look like the word January 1st with an 'st' and 2020. So you want to make it look little different. This is what the DATE_FORMAT is going to do. So DATE_FORMAT and what date are you formatting and we'll throw in order date again and the second parameter is really interesting it's a a collection of little codes and I gave you a little table in your reading that showed some of the different codes. There's lots of different so do you can look them up if you don't have them memorized. So they're format code description patterns and so just by looking them up you would know that they all start with percent sign. Percent sign and capital 'N' happens to be the month day name like January spelled all the way out or June or whatever. Then you might want to say percent sign capital 'D' happens to be the day of the month with suffix like first, second, third. And then the year the percent sign capital 'Y' happens to be a four digit year. You can get two digit years. You can get all sorts of different symbols out there. So that's how your second parameter is going to be and you're going to have

it in quotes because these are like string literals. Let me run this so you can see, oh you need from the customer order table.

SELECT DATE_FORMAT(order_date, '%M %D, %Y')
FROM cust_order;

So there, it looks different. It's the same date but you formatted it different. You give them the name of the date, the one with the suffix. So you could even put string in here. You could say January the first.

SELECT DATE_FORMAT(order_date, '%M **the** %D, %Y')
FROM cust_order;

You can actually put dashes here. And for example the numeric month is 'c' and the day of the month numeric without the little extension on the end is 'e'. If we do a lower case 'y' it's the 2 digit year. We can put slashes in between all these.

SELECT DATE_FORMAT(order_date, '**%c / %e / %y**')
FROM cust_order;

And you get all these different formats. So you can format in lots of different ways. So this is kind of cool, the second parameter that you use with DATE_FORMAT.

There's lots of other functions out there that we haven't covered. But these are some of the more common ones. But go look some up. There's a really cool one with REPLACE. There's all sorts of other cool functions that you can look at but we're just kind of introducing functions at this point and looking at some of them. LOCATE and SUBSTRING are probably the most complicated one that we've seen. But they all have a very good use and like I said you'll use some with other functions. In your homework you're definitely going to be using some functions as parameters of other functions. So it's all about the problem solving sometimes with functions and really digging into exactly what you need from this. So hopefully these common functions have helped you understand what a function is and how they work.