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:

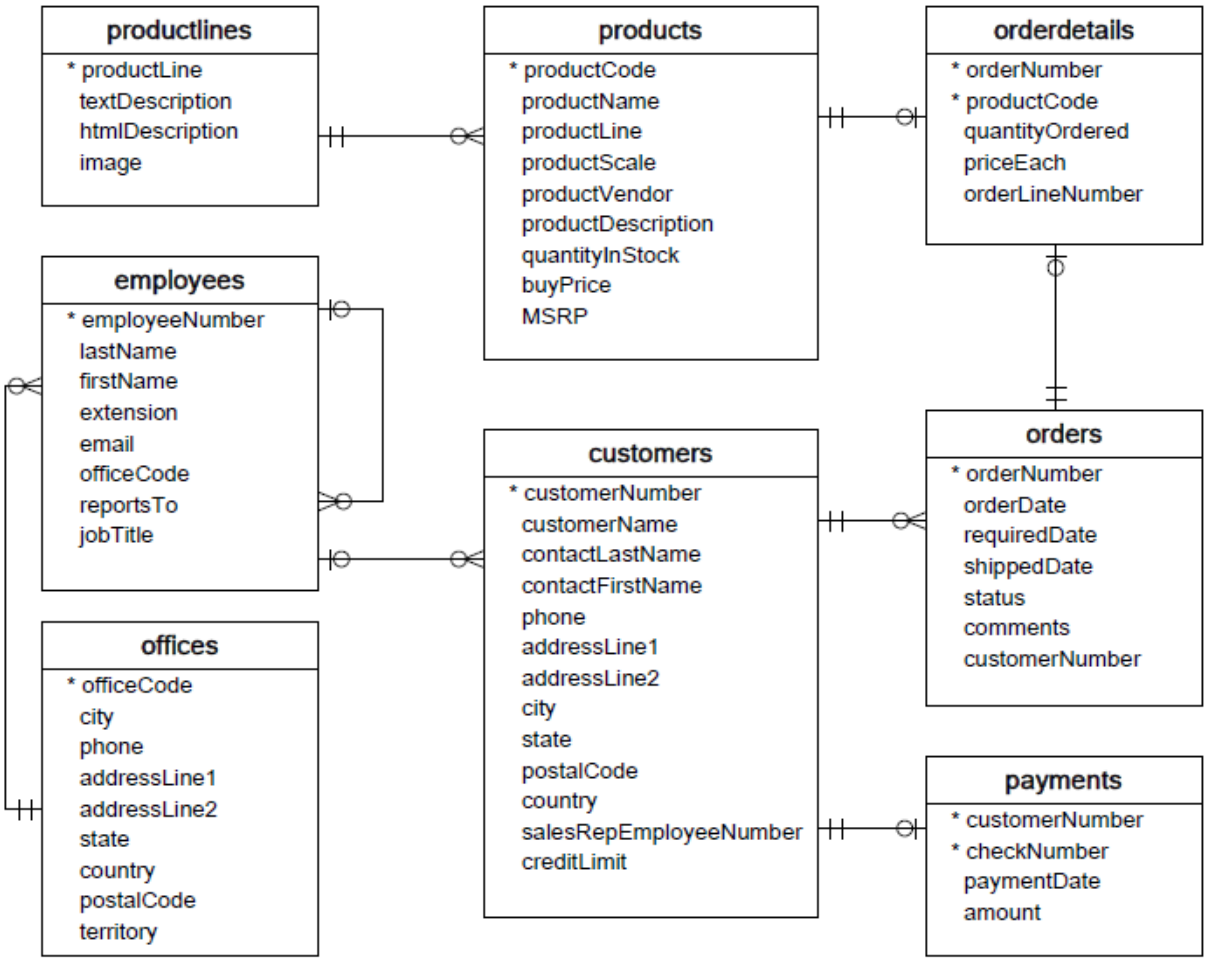
- Retrieve a subset of columns from a table
- Create an alias in a SQL query
- Use SQL CASE statements to transform selected columns
- Use built-in SQL functions to transform selected columns

The Data

Below, we connect to a SQLite database using the Python `sqlite3` library (documentation here (https://docs.python.org/3/library/sqlite3.html)):

In [2]:
```
import sqlite3
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):

In [3]:
```
import pandas as pd
pd.read_sql("""SELECT * FROM employees;""", conn)
```

Out[3]:

	employeeNumber	lastName	firstName	extension	email	officeCo
0	1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	
1	1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	
2	1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	
3	1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	
4	1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	
5	1143	Bow	Anthony	x5428	abow@classicmodelcars.com	
6	1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	
7	1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	
8	1188	Firrelli	Julie	x2173	jfirrelli@classicmodelcars.com	
9	1216	Patterson	Steve	x4334	spatterson@classicmodelcars.com	
10	1286	Tseng	Foon Yue	x2248	ftseng@classicmodelcars.com	
11	1323	Vanauf	George	x4102	gvanauf@classicmodelcars.com	
12	1337	Bondur	Loui	x6493	lbondur@classicmodelcars.com	
13	1370	Hernandez	Gerard	x2028	ghernande@classicmodelcars.com	
14	1401	Castillo	Pamela	x2759	pcastillo@classicmodelcars.com	
15	1501	Bott	Larry	x2311	lbott@classicmodelcars.com	
16	1504	Jones	Barry	x102	bjones@classicmodelcars.com	
17	1611	Fixter	Andy	x101	afixter@classicmodelcars.com	
18	1612	Marsh	Peter	x102	pmarsh@classicmodelcars.com	
19	1619	King	Tom	x103	tking@classicmodelcars.com	
20	1621	Nishi	Mami	x101	mnishi@classicmodelcars.com	

	employeeNumber	lastName	firstName	extension	email	officeCo
21	1625	Kato	Yoshimi	x102	ykato@classicmodelcars.com	
22	1702	Gerard	Martin	x2312	mgerard@classicmodelcars.com	

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:

In [4]:
```
pd.read_sql("""
SELECT *
  FROM employees;
""", conn)
```

Out[4]:

	employeeNumber	lastName	firstName	extension	email	officeCode
0	1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1
1	1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	1
2	1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	1
3	1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	6
4	1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4
5	1143	Bow	Anthony	x5428	abow@classicmodelcars.com	1
6	1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	1
7	1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	1
8	1188	Firrelli	Julie	x2173	jfirrelli@classicmodelcars.com	2
9	1216	Patterson	Steve	x4334	spatterson@classicmodelcars.com	2
10	1286	Tseng	Foon Yue	x2248	ftseng@classicmodelcars.com	3
11	1323	Vanauf	George	x4102	gvanauf@classicmodelcars.com	3
12	1337	Bondur	Loui	x6493	lbondur@classicmodelcars.com	4
13	1370	Hernandez	Gerard	x2028	ghernande@classicmodelcars.com	4
14	1401	Castillo	Pamela	x2759	pcastillo@classicmodelcars.com	4
15	1501	Bott	Larry	x2311	lbott@classicmodelcars.com	7
16	1504	Jones	Barry	x102	bjones@classicmodelcars.com	7
17	1611	Fixter	Andy	x101	afixter@classicmodelcars.com	6
18	1612	Marsh	Peter	x102	pmarsh@classicmodelcars.com	6
19	1619	King	Tom	x103	tking@classicmodelcars.com	6
20	1621	Nishi	Mami	x101	mnishi@classicmodelcars.com	5

	employeeNumber	lastName	firstName	extension	email	officeCode
21	1625	Kato	Yoshimi	x102	ykato@classicmodelcars.com	5
22	1702	Gerard	Martin	x2312	mgerard@classicmodelcars.com	4

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 (https://www.sqlstyle.guide/)](https://www.sqlstyle.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:

In [7]:
```
pd.read_sql("""
SELECT lastName, firstName
  FROM employees;
""", conn).head()
```

Out[7]:

	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:

In [8]:
```
pd.read_sql("""
SELECT firstName, lastName
  FROM employees;
""", conn).head()
```

Out[8]:

	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:

In [6]:
```
pd.read_sql("""
SELECT firstName AS name
  FROM employees;
""", conn).head()
```

Out[6]:

	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":

In [7]:
```
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)
```

Out[7]:

	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:

In [8]:
```
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)
```

Out[8]:

	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:

In [9]:
```
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)
```

Out[9]:

	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 (https://docs.python.org/3/library/functions.html), 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 (https://www.sqlite.org/lang_corefunc.html) page, mathematical functions (https://www.sqlite.org/lang_mathfunc.html) page, and/or date and time functions (https://www.sqlite.org/lang_datefunc.html) 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 (https://www.sqlite.org/lang_corefunc.html#length)). 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:

In [10]:
```
pd.read_sql("""
SELECT length(firstName) AS name_length
  FROM employees;
""", conn).head()
```

Out[10]:

	name_length
0	5
1	4
2	4
3	7
4	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 (https://www.sqlite.org/lang_corefunc.html#upper)). 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`:

In [11]:
```
pd.read_sql("""
SELECT upper(firstName) AS name_in_all_caps
  FROM employees;
""", conn).head()
```

Out[11]:

	name_in_all_caps
0	DIANE
1	MARY
2	JEFF
3	WILLIAM
4	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 (https://www.sqlite.org/lang_corefunc.html#substr)).

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

In [12]:
```
pd.read_sql("""
SELECT substr(firstName, 1, 1) AS first_initial
  FROM employees;
""", conn).head()
```

Out[12]:

	first_initial
0	D
1	M
2	J
3	W
4	G

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:

In [13]:
```
pd.read_sql("""
SELECT substr(firstName, 1, 1) || "." AS first_initial
  FROM employees;
""", conn).head()
```

Out[13]:

	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:

In [14]:
```
pd.read_sql("""
SELECT firstName || lastName AS full_name
  FROM employees;
""", conn).head()
```

Out[14]:

	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:

In [15]:
```
pd.read_sql("""
SELECT firstName || " " || lastName AS full_name
  FROM employees;
""", conn).head()
```

Out[15]:

	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:

In [16]:
```
pd.read_sql("""SELECT * FROM orderDetails;""", conn)
```

Out[16]:

	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
...	...	...	...	...	...
2991	10425	S24_2300	49	127.79	9
2992	10425	S24_2840	31	31.82	5
2993	10425	S32_1268	41	83.79	11
2994	10425	S32_2509	11	50.32	6
2995	10425	S50_1392	18	94.92	2

2996 rows × 5 columns

`round`

Let's say we wanted to round the price to the nearest dollar. We could use the SQL `round` function (documentation here (https://www.sqlite.org/lang_corefunc.html#round)), which is very similar to the the Python `round`:

In [17]:
```
pd.read_sql("""
SELECT round(priceEach) AS rounded_price
  FROM orderDetails;
""", conn)
```

Out[17]:

	rounded_price
0	136.0
1	55.0
2	75.0
3	35.0
4	108.0
...	...
2991	128.0
2992	32.0
2993	84.0
2994	50.0
2995	95.0

2996 rows × 1 columns

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 (https://www.sqlite.org/lang_expr.html#castexpr)):

In [18]:
```
pd.read_sql("""
SELECT CAST(round(priceEach) AS INTEGER) AS rounded_price_int
  FROM orderDetails;
""", conn)
```

Out[18]:

	rounded_price_int
0	136
1	55
2	75
3	35
4	108
...	...
2991	128
2992	32
2993	84
2994	50
2995	95

2996 rows × 1 columns

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:

In [19]:
```
pd.read_sql("""
SELECT priceEach * quantityOrdered AS total_price
  FROM orderDetails;
""", conn)
```

Out[19]:

	total_price
0	4080.00
1	2754.50
2	1660.12
3	1729.21
4	2701.50
...	...
2991	6261.71
2992	986.42
2993	3435.39
2994	553.52
2995	1708.56

2996 rows × 1 columns

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:

In [20]:
```
pd.read_sql("""SELECT * FROM orders;""", conn)
```

Out[20]:

	orderNumber	orderDate	requiredDate	shippedDate	status	comments	customerNumber
0	10100	2003-01-06	2003-01-13	2003-01-10	Shipped		363
1	10101	2003-01-09	2003-01-18	2003-01-11	Shipped	Check on availability.	128
2	10102	2003-01-10	2003-01-18	2003-01-14	Shipped		181
3	10103	2003-01-29	2003-02-07	2003-02-02	Shipped		121
4	10104	2003-01-31	2003-02-09	2003-02-01	Shipped		141
...	...	...	...	...	...	...	...
321	10421	2005-05-29	2005-06-06		In Process	Custom shipping instructions were sent to ware...	124
322	10422	2005-05-30	2005-06-11		In Process		157
323	10423	2005-05-30	2005-06-05		In Process		314
324	10424	2005-05-31	2005-06-08		In Process		141
325	10425	2005-05-31	2005-06-07		In Process		119

326 rows × 7 columns

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:

In [21]:
```
pd.read_sql("""
SELECT requiredDate - orderDate
  FROM orders;
""", conn)
```

Out[21]:

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

326 rows × 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 (https://www.sqlite.org/lang_datefunc.html)):

In [22]:
```
pd.read_sql("""
SELECT julianday(requiredDate) - julianday(orderDate) AS days_from_order_to_requi
  FROM orders;
""", conn)
```

Out[22]:

	days_from_order_to_required
0	7.0
1	9.0
2	8.0
3	9.0
4	9.0
...	...
321	8.0
322	12.0
323	6.0
324	8.0
325	7.0

326 rows × 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:

In [23]:
```
pd.read_sql("""
SELECT orderDate, date(orderDate, "+7 days") AS one_week_later
  FROM orders;
""", conn)
```

Out[23]:

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

326 rows × 2 columns

You can also use the `strftime` function, which is very similar to the Python version (https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior). 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:

In [24]:
```
pd.read_sql("""
SELECT orderDate,
       strftime("%m", orderDate) AS month,
       strftime("%Y", orderDate) AS year,
       strftime("%d", orderDate) AS day
  FROM orders;
""", conn)
```

Out[24]:

	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
...	...	...	...	...
321	2005-05-29	05	2005	29
322	2005-05-30	05	2005	30
323	2005-05-30	05	2005	30
324	2005-05-31	05	2005	31
325	2005-05-31	05	2005	31

326 rows × 4 columns

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

In [25]:
```
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.