**Introduction to relational databases**

You have already mastered the art of importing all types of single files in Python: congratulations! However, to earn your daily bread and butter as a Data Scientist, you'll be required to interact with more complex data structures, such as relational databases.

**What is a relational database?**

What is a relational database? It's a type of database that is based upon the Relational model of data, first described by Ted Codd in the late 1960s. Before getting too theoretical, however, let's check out at an illuminating example,

**Example: Northwind database**

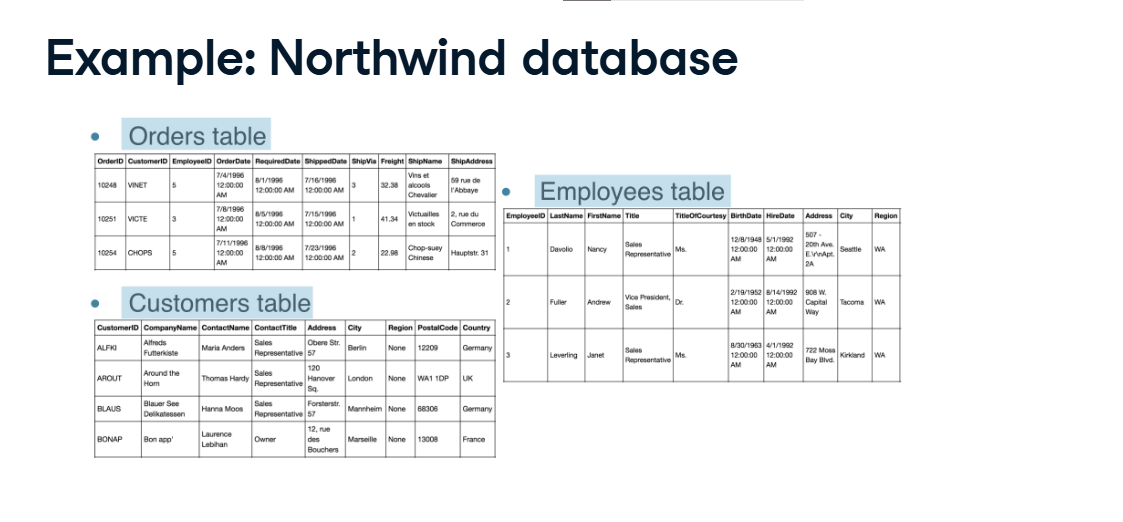
the Northwind Traders database, a synthetic database that contains sales data for a fictitious company. Firstly, a database consists of tables. Here you can see 3 tables from the Northwind database:

**Example: Northwind database**

'Orders',

**Example: Northwind database**

'Customers' and 'Employees'. So what's a table? A table generally represents one entity type,**.**



**The Orders table**

such as 'Order' . Notice that this table looks a great deal like a dataframe. That's the point. In a relational database table,each row or record represents an instance of the entity type: in this case, each row is an Order.

Each column represents an attribute of each instance, such as 'OrderDate' in the case of 'Orders'. In this sense, a table is entirely analogous to a dataframe. It is essential that each row contain a unique identifier, known as a primary key, that we can use to explicitly access the row in question. In our 'Orders' table, you can see that

**The Orders table**

the key is 'OrderID' the first column. But recall that a database consists of many tables! The really cool thing about relational databases is not merely that you have a bunch of tables,

**Tables are linked**

but that the tables are linked. How this linking works is ultra-intuitive: see that the 'Orders' table has both a column called 'CustomerID' and one called 'EmployeeID'. These columns correspond precisely to the primary keys in the

**Tables are linked**

'Customers' and 'Employees' tables, respectively. So, given an Order, you can immediately look up the details of the relevant Customer or Employee. This is cool because it means that you don't need to store all the Customer details, such as first name, last name, company with every order that they place: you merely need to look it up in the 'Customers' table. This saves an incredible amount of space!

**Relational model**

As stated earlier, the relational database model was originally proposed by "Ted" Codd and has been widely adopted. There is a great deal to theory but it is most neatly summarized in Codd's 12 Rules, also known as Codd's 12 Commandments, which he developed in the early 1980s to combat what he viewed as a dilution of his original relational database vision. Codd's 12 Rules actually consist of 13 rules but they are zero-indexed, that is, the first rule is zero-indexed. If that makes you laugh, you're definitely a geek like me! These 13 rules were defined to describe what a Relational Database Management System should adhere to in order to be considered relational.

**Relational Database Management Systems**

Among the most popular of such systems are PostreSQL (commonly called Postgres),

**Relational Database Management Systems**

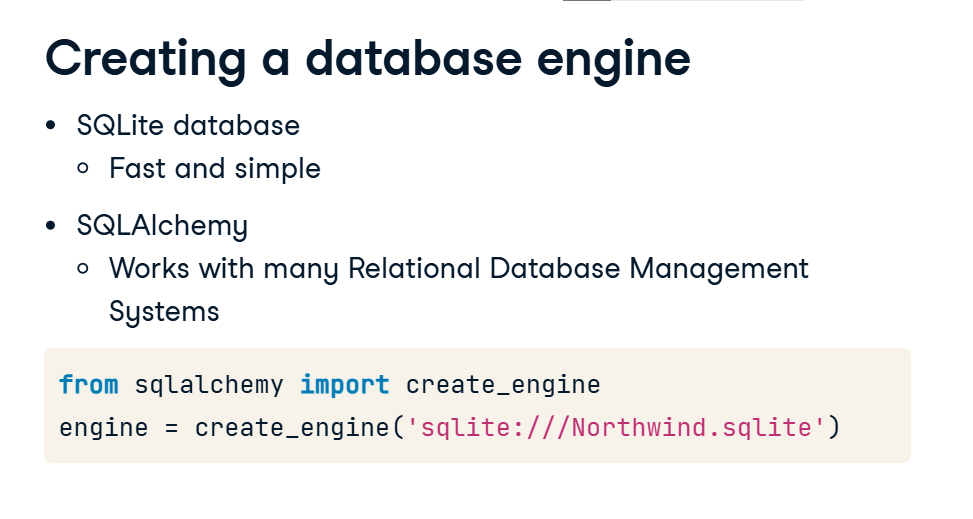
MySQL and SQLite, all of which use the SQL query language. In fact, SQL itself is actually an acronym for Structured Query Language, which describes how you communicate with a database in order to both access and update the information it contains. The term "querying" is really just a fancy way of saying getting data out from the database. Next up, you'll learn how to connect to databases but before that, let's make sure that you have a solid conceptual grip on the relational model.

**Creating a database engine in Python**

All right, we're back! What we really want to do is to get data out of our databases using SQL, or the Structured Query Language. But before we get to that, we're going to need to figure out how to connect to a database.

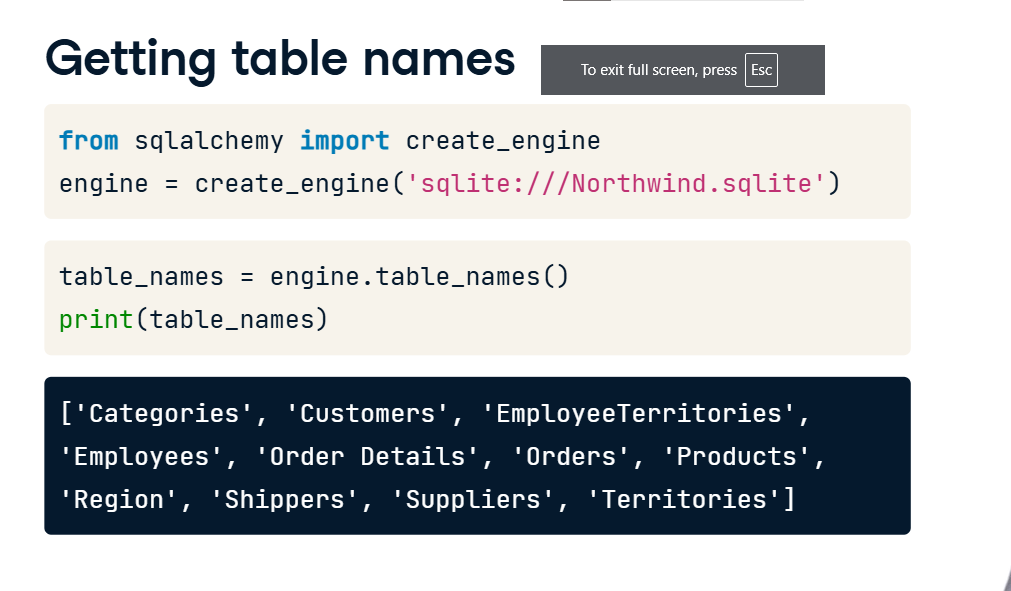
**Creating a database engine**

We'll use an SQLite database as an example because SQLite is fast and simple while still containing enough functionality to introduce you to all the necessary concepts of querying a database. There are times when you would prefer to use PostgreSQL or MySQL, but for our purposes here, an Introduction to Interacting with Relational Databases in Python, a SQLite database is perfect. We'll once again look at the Northwind database. There are many packages we could use to access an SQLite database such as sqlite3 and SQLAlchemy. We'll use SQLAlchemy as it works with many other Relational Database Management Systems, such as Postgres and MySQL. So without further ado, to connect to 'Northwind dot sqlite', we need to import the relevant funtion create\_engine from the package SQLAlchemy. We then use the function create\_engine to fire up an SQL engine that will communicate our queries to the database. The only required argument of create\_engine is a string that indicates the type of database you're connecting to and the name of the database. Next, In order to query the database, we need to connect to the engine to do so.



**Getting table names**

But before we do this, we would like to know the names of the tables it contains. To do this, apply the method table\_names to the object engine. This will return a list of the table names that you can then print the console. Now it's your turn to do the same: fire up the database engine and print the table names!

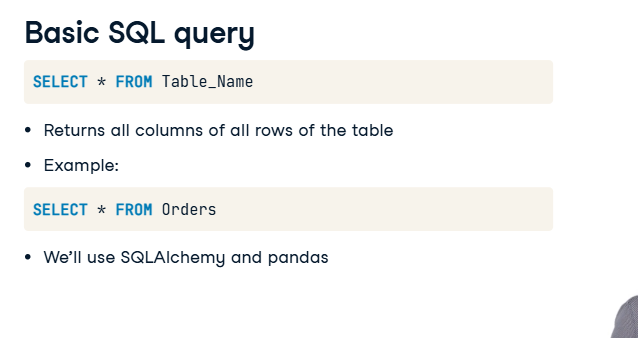


**Querying relational databases in Python**

Now that you have figured out how to create a database engine and to list the tables of the database in question, it's time to connect to the engine and query the database. Once again, the term "querying" is just a fancy way of saying getting data out from the database.

**Basic SQL query**

THe HELLO WORLD of SQL queries is 'SELECT \* FROM Table\_Name', where 'Table\_name' is the name of any of the tables in the database. This query returns all columns of all rows of the Table of interest. For example, I could query the Northwind database with 'SELECT \* FROM Orders' and this would return all columns of all rows of the 'Orders' table. The star after SELECT means 'all columns'. Straightforward, right? Well, nearly! This is an SQL query and we need to figure out how to make such a query using python, SQLAlchemy and, in fact, we'll also use pandas to store the results of our queries.

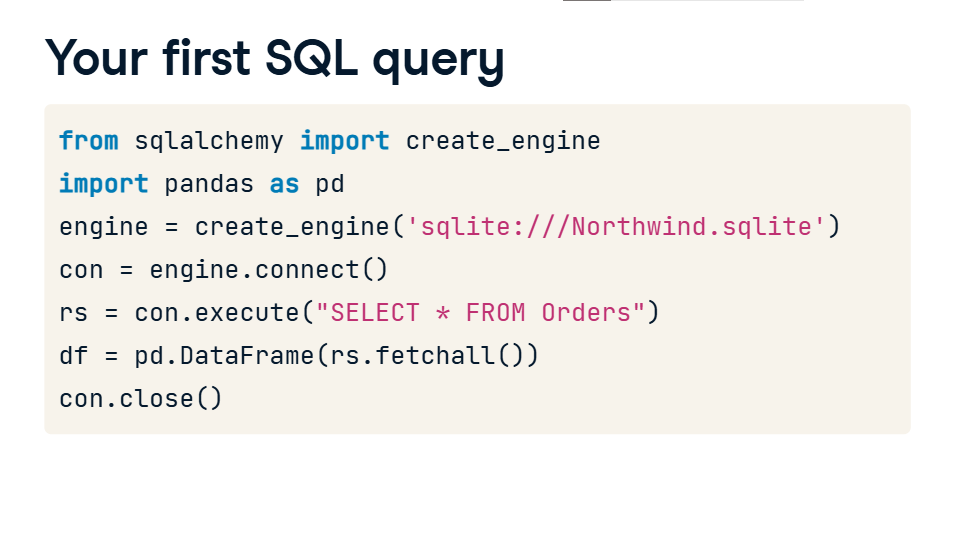


**Workflow of SQL querying**

The workflow will be as follows. You'll import the required packages and functions, create the engine, connect to it, query the database save the results of the query to a dataframe, and close the connection.

**Your first SQL query**

Let's now check out how to do each of these steps! Create the engine using the function create\_engine. To connect to the database after creating the engine, you create a connection object con by applying the method connect to the engine. To query the DB, apply the method execute to the connection con and pass it a single argument, the relevant SQL query; This creates a SQLAlchemy results object which we assign to the variable rs. To turn the results object rs into a dataframe, we apply the method fetchall to rs and save it as a dataframe using the pandas function DataFrame. fetchall fetches all rows, as you would expect. To close the connection, execute 'con dot close'. Don't forget to do this!

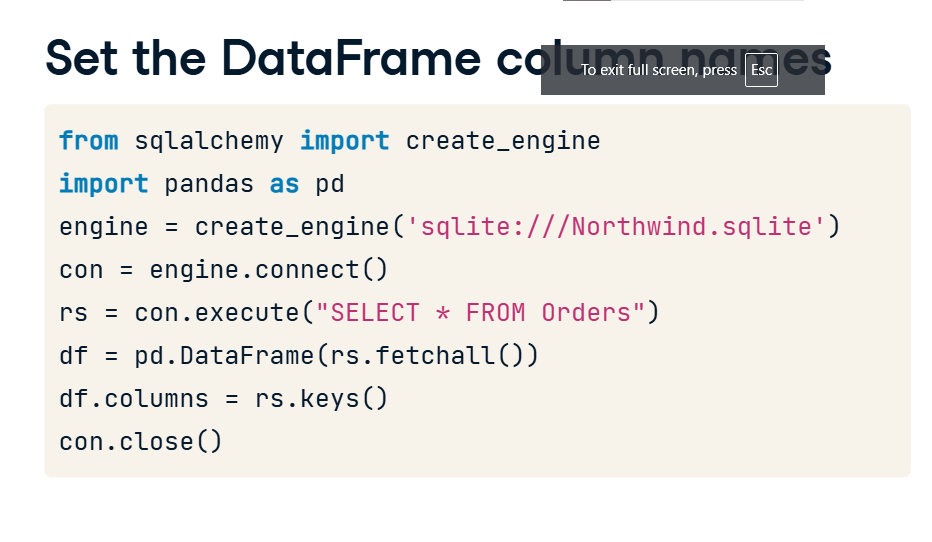


**Printing your query results**

You can then print the head of the dataframe, as we have done before, as a sanity check: all the rows look good but the column names aren't correct.

**Set the DataFrame column names**

To fix this, before closing the connection, you can set the dataframe's column names by executing 'df dot columns equals rs dot keys'.

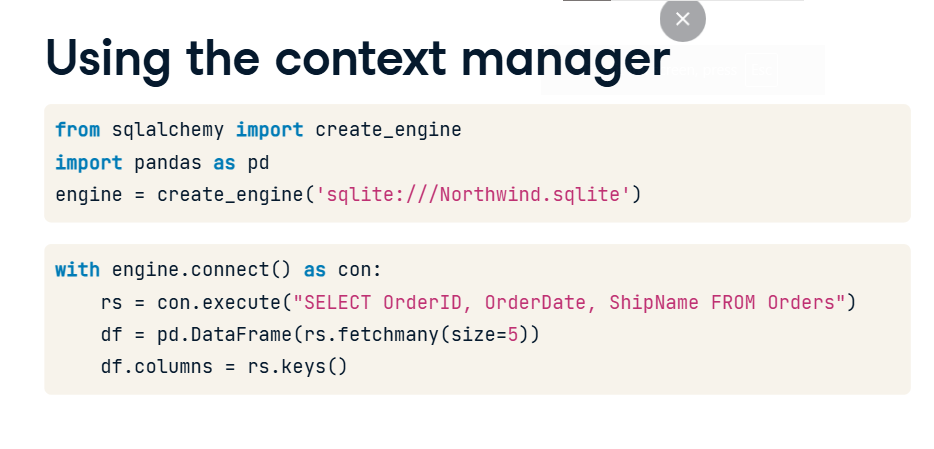


**Set the data frame column names**

One last note: analogous to what you saw in chapter 1 when opening plain text files, you can use

**Using the context manager**

the context manager construct to open a connection, which will save you the trouble of closing the connection later, or save you the trouble of forgetting to close it! There are two other differences that you may have notice between this and the previous code: firstly, I no longer have 'SELECT \*' in the SQL query; I now have column names of the table 'Orders'; all this does is it imports those particular columns and no others whereas 'SELECT \*' imports all columns; secondly, instead of applying the method fetchall to the results rs, I apply the method fetchmany with the argument size equals 5; this imports 5 rows instead of all rows. You'll become better acquainted with these functions and arguments very soon.

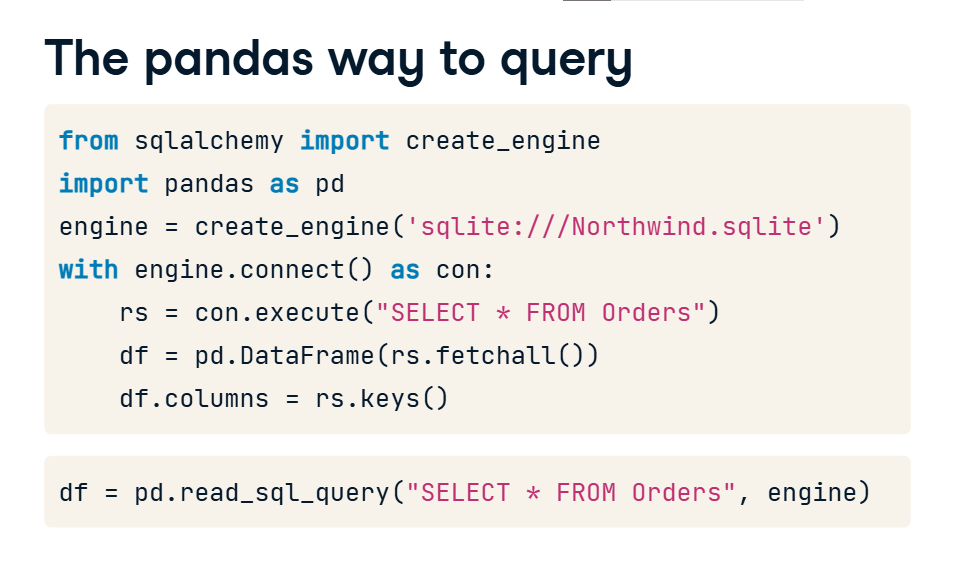


**Querying relational databases directly with pandas**

You have seen that,

**The pandas way to query**

after creating a database engine, you can get the results of any particular line using 4 lines of code: connecting, executing a query, passing the results to a dataframe and naming the columns: 4 lines of code is pretty good but you can do better! You can actually do it in 1 line, utilizing the pandas function read\_sql\_query and passing it 2 arguments. The first argument will be the query you wish to make, the 2nd argument the engine you want to connect to. And thus you can achieve the same as this code by executing this single line. The power of pandas! In the following exercises, you'll gain more expertise in writing SQL queries and using pandas to execute them.



**Advanced querying: exploiting table relationships**

By now, you have become familiar with querying individual tables of databases, such as the 'Orders' table of the Northwind database. You will remember, however, that much of the power of relational databases stems from the fact that they can capture relationships between tables: the tables are linked!

**Tables are linked**

For example, as we saw earlier, the 'Orders' table of the the Northwind Traders database has both a column called

**Tables are linked**

'CustomerID' and one called 'EmployeeID', columns which correspond precisely to the primary keys in the 'Customers' and 'Employees' tables, respectively. This means that, given an Order, you can immediately look up the details of the relevant Customer or Employee in the appropriate table.

**JOINing tables**

Now what if you want to incorporate such information into your query? For example, if you want to query the 'Orders' table and include, for each Order, information about the corresponding Customer from the 'Customers' table? A specific illustrative example will go a long way here: let's say that we wanted, for each Order, to get the

**JOINing tables**

OrderID and the

CompanyName of the Customer. The OrderID lives in the 'Orders' table while the CompanyName lives in the 'Customers' table. SQL has a really clever way of doing this: it's called a JOIN because what you're really doing is joining two tables together,

in this case, the 'Orders' and

'Customers' tables. Specifically

**INNER JOIN in Python (pandas)**

it's an INNER JOIN. There are other types of JOINs which we won't cover here. As it's the CustomerID columns of the 'Orders' and 'Customers' tables that correspond to each, you'll want to JOIN the tables ON these columns and that is precisely what I have done in this code. The notation of dot followed by Column name is merely selecting a column of a table. The table that we are selecting from is "Orders INNER JOIN Customers on Orders dot CustomerID equals Customers dot CustomerID" and I am selecting the OrderID column and the CompanyName column of this new table. I know that that's a bit to take in, but after you get your hands dirty performing some INNER JOINs, you'll be far more comfortable with this advanced querying technique.

