Chapter 6
Databases and Other Tools

In this chapter, we show how to connect to a relational database from R and from Python. Unlike the rest of the book, this chapter assumes that the reader is already familiar with R and/or Python.

Communication between R and Python and a database is very formulaic; once the basic pattern is understood, most of the work is to figure out what we want to extract from the database. In the following examples, we stick with very simplistic interactions, in order to make the basic pattern clear; we use mostly examples from the documentation (in `cran.r-project.org` or the package documentation). The important point to remember is that all the interactions with the relational database are carried out in SQL; therefore, everything shown in the rest of the book can be applied here.

6.1 SQL and R

There are many ways to work with databases within R. Among them,

- Using the DBI library.
- Using packages dplyr and dbplyr.
- Using package sqldf.

In this section we describe each one of them. We assume that the reader is familiar with the basics of R; in particular, knowledge of *data frames* and their manipulation is assumed. We show how to translate much of the SQL we have seen in the book into R, but we will not go into full detail.

In our examples, we use the `mtcars`, `warpbreaks`, and `iris` datasets, which come with the standard R distribution,[1] as well as the New York flights dataset we

[1] Use `data()` to list the data frames available (built-in) in R.

A. Badia, *SQL for Data Science*, Data-Centric Systems and Applications,
https://doi.org/10.1007/978-3-030-57592-2_6

have been using all along, so that the similarities between SQL and R are made clear. We show R commands as they should be typed into the system; comments are in lines started with '#'. Occasionally we show the answer of the systems; such lines are prefixed with '>.'

6.1.1 DBI

DBI is a package that provides basic interfacing with databases. As such, it is used by some of the other interfaces, as will become apparent. Here we show only some basic DBI functionality.

DBI connects with a relational database and interacts with it using SQL. It uses packages called `drivers` to handle interaction with a particular database (Postgres, MySQL, etc.). This allows DBI to hide a large amount of low-level details and make the interaction with the database much simpler than it would otherwise be.

The package provides a collection of functions to support each step in the interaction with the database. These steps are:

1. connection: to establish a connection with the database, the function `dbConnect` takes as arguments a `driver` (depending on the type of database one is connecting to), a `host` (addresses where to find the database), `dbname` (database name), `user` (username), and `password` (ditto). This function returns an object of type 'connection,' which is used as a parameter by other functions that make use of this connection.
2. access: to access the database, the user has at her disposal a series of functions that take, as one of the parameters, the connection object (this makes sure the right database is accessed) and as another one a string representing an SQL command, which tells the database what we want to do. Depending on the required access, the most common functions are:
   - for metadata, `dbListTables` takes a connection object as parameter and lists all the tables available in the database accessed; `dbListFields` takes a connection object and a table name as parameters and returns the schema of the table. There is also metadata associated with a result returned by a query (see below); function `dbColumnInfo` takes as argument a result set and gives information about its schema.
   - to send data from R to the database, function `dbWriteTables(connection, table-name, data-frame)` will create a table in the connected database with the name given by the second argument and will copy to it the data in the third argument. The schema of the table is deduced from the data frame.
   - to get data from the database into R, function `dbReadTables(connection, table-name)` returns a data frame containing the data in the table named by the second parameter.
   - to make changes in the database, functions `dbCreateTable`, `dbRemoveTable` do exactly what the name suggests, creating and dropping tables. The first

one takes as arguments a connection, a name for the table, and (optionally) a named vector describing the attributes of the table (the name is the name of the attribute, the value is its data type), or a data frame. When using a data frame, a schema is inferred from the data in the data frame. Also, function `dbExecute` takes as arguments a connection and a string representing a "modify" SQL statement, that is, INSERT, DELETE, or UPDATE.

- to retrieve some data selectively from the database, we use `dbSendQuery`, which takes as parameters a connection and a string with an SQL SELECT statement. This function returns an object called a *result set*, which can then be used to access whatever data was returned by the query. The data in the result set is typically accessed by *iterating* (AKA *looping*) through it, that is, accessing each record one by one. The iteration is accomplished by combining several functions:
  - function `dbFetch`, which takes as arguments a result set and (optionally) a number n. The function returns records from the result set, which can be assigned to a data frame in R. If $n > 0$, it returns n records from the result set (assuming there are at least n records; if there are fewer than n, it just returns whatever records are available). If $n < 0$, all remaining records are returned. If n is not used, all records in the result set are returned.
  - function `dbHasCompleted` returns True if fetching has exhausted all the rows in the return set; False if there are any still left.

  A typical iteration is shown in the example below.

3. Finally, `dbDisconnect` takes as parameter the connection object and 'finishes' it, closing the connection.

Example: Connecting to a Database with DBI

Typical code used in a database connection with `DBI`:

```
library(DBI)
# Connect to a MariaDB remote database
con <- dbConnect(RMariaDB::MariaDB(), host = "hostname.com",
                 user = "username", password = "password")

# Connect to a MariaDB database local database
con <- dbConnect(RMariaDB::MariaDB(), dbname = "mydb")

# Create and connect to an in-memory RSQLite database
con <- dbConnect(RSQLite::SQLite(), dbname = ":memory:")

#find out what tables exist in the database, if any
dbListTables(con)

#write data frame mtcars to database table "mtcars"
dbWriteTable(con, "mtcars", mtcars)
```

```
#find out attributes of table
dbListFields(con, "mtcars")

# read table "mtcars" into R data frame
df = dbReadTable(con, "mtcars")

# You can fetch all results at once:
res <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbFetch(res)

# Or a chunk at a time
res <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
while(!dbHasCompleted(res)){
  chunk <- dbFetch(res, n = 5)
  print(nrow(chunk))
}

#disposes of the results after iteration
dbClearResult(res)

dbDisconnect(con)
```

The loop shown above is the typical way to examine the result returned by a query, using the parameter *n* to control how many records we retrieve at a time. This is especially useful for very large results, since one of the limitations of R is that it does not do well when it cannot have all data in a data frame (or any other structure) in memory. A similar approach is used in other interfaces, as we will see.

The DBI package contains many other functions that allow the user to 'remotely control' a database, that is, to connect to it and send all sorts of SQL statements, achieving the same effects as if the user were directly connected to the database. In addition, the package makes very easy to transfer data between R and the database. Note, though, that all the interactions with the database are handled using SQL.

DBI (and several other packages) uses (sometimes by default) a database called SQLite. This system, unlike Postgres and MySQL (and most any other database system) does not require any setup and can be used from within other applications without any configuration. This makes it a very popular database for R and other systems to call.

Example: Using SQLite

The approach is similar to using other databases with DBI:

```
install.packages("RSQLite")
library(RSQLite)

#create a new, empty database using the given file
conn <- dbConnect(SQLite(),'mycars.db')
```

```
#create a table with schema deduced from data
dbWriteTable(conn, "cars", mtcars)

#one can also create tables the traditional way
dbGetQuery(conn, 'CREATE TABLE test_table(id int, name text)')
```

SQLite stores all data in a single file and does not enforce data types or foreign keys, so it is possible to store 'wrong' data on it. Its use is mainly for lightweight, exploratory analysis. Most standard SQL querying is available:

```
dbGetQuery(conn, "SELECT * FROM cars WHERE mpg > 20")
```

If one already has a Postgres server up and running, using it instead is very simple:

```
#installing the library
install.packages("RPostgreSQL")
library(RPostgreSQL)
#a driver is needed
drv <- dbDriver("PostgreSQL")
#establish a connection
con <- dbConnect(drv,
                 user = "postgres",
                 dbname = "databaseName",
                 host = "myhost.com")
```

As a rule for all databases, the equivalent of SQL NULL in R is NA. The predicate `is.na` is equivalent to `IS NULL`; `!is.na` is the equivalent of `IS NOT NULL`. To get all rows without nulls in a data frame `df`, R uses `na.omit(df)`.

6.1.2 dbplyr

The `dbplyr` package can be seen as an extension of the popular `dplyr` package. `dbplyr` adds to `dplyr` the ability to deal with data in a database. This is helpful when the data is in the database to start with, or when the data does not fit into memory (more on this later). `dbplyr` connects to the database using DBI connections; for instance, it can connect to MySQL using the `RMySQL` driver and to Postgres using the `RPostgreSQL` driver. After the connection is established, `dbplyr` uses its own commands to move data between R and the database.

To start with, the package must be installed like any R package.

```
install.packages(c("dplyr", "dbplyr"))
library(dplyr)
library(dbplyr)
```

Then a connection to a database is established:

```
#create an in-memory SQLite database and copy over a dataset:
>con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")

#For databases not in memory, in a remote server:
con <- DBI::dbConnect(RMySQL::MySQL(), host = "hostname",
                user = "username", password = "myPassword")
```

`dbplyr` can be used with

- data in R: a data frame can be copied to a database table.
- data in files: data from files can be read into R and then copied to the database.
- data already in the database.

```
#copy from data frame to database table
copy_to(con, nycflights13::flights, "flights")

#from files:
df <- read_csv("filename.csv")

#Create a new SQLite database.
#the argument is a file that SQLite uses for its data
#Warning: any data in the argument is overwritten!
my_db_file <- "myfile.sqlite"
my_db <- src_sqlite(my_db_file, create = TRUE)

copy_to(my_db, df)
```

Data in the database is accessed using the `tbl()` command. The data can be retrieved in two ways:

- using SQL: `dbplyr` can be used to send SQL directly to the database. When a SELECT statement is used, the result is translated into an R data structure, as before.

  ```
  tbl(con, sql("SELECT mpg, wt FROM mtcars"))
  ```

- using `dplyr` commands: `dbplyr` will translate R code into SQL, send it to the database, get the answer back, and translate the answer into an R data structure (data frame). This option is detailed in the next example.

Example: Using `dbplyr` with R Commands

```
flights_db <- tbl(con, "flights")

flights_db %>% select(year:day, dep_delay, arr_delay)

flights_db %>% filter(dep_delay > 240)
```

```
flights_db %>%
  group_by(dest) %>%
  summarise(delay = mean(dep_time))

tailnum_delay_db <- flights_db %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay),
    n = n()
  ) %>%
  arrange(desc(delay)) %>%
  filter(n > 100)
```

`dbplyr` translates the above code into SQL. The translation can be inspected by using `show_query()`:

```
summary <- mtcars2 %>%
   group_by(cyl) %>%
   summarise(mpg = mean(mpg, na.rm = TRUE)) %>%
   arrange(desc(mpg))

summary %>% show_query()

<SQL>
SELECT `cyl`, avg(`mpg`) AS `mpg`
FROM `mtcars`
GROUP BY `cyl`
ORDER BY `mpg` DESC
```

Roughly speaking, the translation works as follows:

- `head` generates a LIMIT clause.
- `filter` results in conditions in a WHERE clause.
- `select` gives attributes in a SELECT clause (if no `select` is used, all attributes available are retrieved with '*').
- `groupby` gives attributes for a GROUP BY clause. This can used with `summarise` to compute aggregates. `aggregates` can be used for aggregates on their own (for instance, `aggregates (n_distinct)` generates COUNT (DISTINCT)) in SQL).
- `arrange` generates an ORDER BY clause.
- For dealing with multiple tables, `dplyr` has an "inner_join" function, as well as a "left_join" and `righ_join` functions that simulate a left (right) outer join (see Sect. 5.1). They take the attribute tables used for the join condition as arguments.[2]
- `intersect`, `union`, and `setdiff` generate INTERSECT, UNION, and EXCEPT in SQL.

[2]Note that only *equi-joins* are supported, but these are by far the most common case.

As shown above, to write the query in `dplyr`, you connect multiple commands using the %>% pipe to build a pipeline.

The system can also deal with functions:

```
mf <- memdb_frame(x = 1, y = 2)

mf %>%
  mutate(
    a = y * x,
    b = a ^ 2,
  ) %>%
  show_query()
#> <SQL>
#> SELECT `x`, `y`, `a`, POWER(`a`, 2.0) AS `b`
#> FROM (SELECT `x`, `y`, `y` * `x` AS `a`
#> FROM `dbplyr_002`)
```

It is highly instructive to build pipelines in `dplyr` and then show their SQL counterpart.

One of the advantages of using `dbplyr` is its ability to handle large dataset using a technique called *lazy evaluation*. Basically, when the user sends a query to the database, the system does not execute the query right away; hence, no results are generated (yet). When the user wants the result, she must ask for them, using `collect()`:

```
mtcars2 %>% select(mpg, wt) %>% collect()

#Or, if a previous query was saved in a variable:
summary %>% collect()
```

Also, the system does not retrieve all results at once. This is why, when looking at some results, one may receive an answer with some data and a last line like

```
# ... with more rows
```

The system does not retrieve all results at once. This is also the reason that, when asking about the size of an answer, the system claims it does not know it:

```
nrow(mtcars2)

#> [1] NA
```

The user can ask for more results using `collect()`. While this may seem like a strange technique, it allows R to deal with results that exceed the size of the available memory.

6.1.3 sqldf

`sqldf` is an R package that also allows an R user to use SQL over R data or database data. `sqldf` can work with a data frame in R's memory; with data loaded from a

file, and with data that resides in a database. After getting installed as usual in R,

```
install.packages("sqldf")
library(sqldf)
```

the package can be used with data frames simply by using the data frame name as a table name in the FROM clause of a query, so that the data frame can be manipulated using SQL:

```
data(mtcars)

sqldf("SELECT * FROM mtcars WHERE mpg > 20")
```

`sqldf` can also be used with a data file by using the `file` command; this creates a connection, which can then be used as a data source:

```
iris = file("iris.dat")
sqldf("SELECT count(*) FROM iris")
```

Another way to read from a file is to use the function `read.csv.sql`. This function, like `read.csv`, will take a file name to read as argument, but it will also take an additional argument called `sql`, where the user can put an SQL SELECT statement. Using this statement, it is possible to control exactly which data/how much data is read, something very useful when the file is very large.

We can perform insert, update, and delete statement through `sqldf`. However, it is important to understand that `sqldf` automatically copies the data from the data frame to the SQLite database, and all changes are made to the copy in the database, not to the data in R. To distinguish between the two, the copy in the database uses the prefix `main` in its name:

```
sqldf("INSERT INTO main.iris
              values(4.1, 3.2, 4.5, 1.3, 'setosa')")
data frame with 0 columns and 0 rows
sqldf("SELECT count(*) FROM main.iris")
  count(*)
1      151
```

The result returned by a `sqldf` query is a data frame, which can be saved and then used for further work. As a result, more complex analyses can be broken down into steps, just like using subqueries or views in SQL:

```
df <- sqldf("SELECT * FROM mtcars WHERE mpg > 20",
            row.names=TRUE)

sqldf("SELECT avg(mpg) avg, min(mpg) min, max(mpg) max
        FROM df WHERE make_model like 'Merc%'")
```

Example: Using `sqldf`

The following examples show how many data frame operations in R have counterparts in SQL and vice versa. Each pair of statements (the first one operating directly on the data frame, the second one using `sqldf`) produces the same result. Note: when comparing the results from R and SQL in this example, the user must take into account that the row names with the name of each car are not standard columns and will not be returned by sqldf unless the user asks for it, using `row.names=TRUE`:

```
sqldf("SELECT * FROM mtcars WHERE mpg > 20", row.names=TRUE)
```

It is also possible to make the row names into a regular column in the data frame by using

```
mtcars$make_model<-row.names(mtcars)
```

This will make comparing SQL and data frame manipulations easier.

```
#picking attributes
mycar = mtcars[,c("mpg","cyl","make")]
sqldf("SELECT  mpg, cyl, make FROM mtcars")

# head
head(mtcars)
sqldf("SELECT * FROM iris limit 6", row.names=TRUE)

head(mtcars, n=10)
sqldf("SELECT * FROM mtcars limit 10", row.names= TRUE)

# order
head(mtcars[order(mtcars$mpg, decreasing=TRUE),], 5)
sqldf("SELECT * FROM mtcars ORDER BY mpg desc limit 5",
       row.names = TRUE)

# subset
subset(mtcars, mpg > 25)
sqldf("SELECT * FROM mtcars WHERE mpg > 25", row.names=TRUE)

subset(mtcars, mpg > 25 & cyl < 5)
sqldf("SELECT * FROM mtcars WHERE mpg > 25 and cyl < 5",
      row.names=TRUE)

subset(mtcars, grepl("Mazda", mtcars$name))
sqldf('SELECT * FROM mtcars WHERE name LIKE "Mazda%"',
      row.names=TRUE)

# aggregate and GROUP BY
aggregate(mtcars$mpg, list(mtcars$cyl), mean)
r2= sqldf("SELECT cyl, avg(mpg) as avgmpg FROM mtcars
           GROUP BY cyl")
```

```
#subqueries, aggregated.
subset(mtcars, mpg > ave(mpg, cyl, FUN = mean))
#reusing previous result
sqldf("SELECT * FROM mtcars, r2 WHERE mtcars.cyl = r2.cyl
       and mpg > avgmpg")
#same query in single statement
sqldf("SELECT *
       FROM mtcars,
            (SELECT cyl, avg(mpg) as avgmpg FROM mtcars
             GROUP BY cyl) as t
       WHERE mtcars.cyl =t.cyl and mpg > avgmpg")

# table for pivoting
table(mtcars$mpg, mtcars$cyl)
sqldf("SELECT sum(cyl=4), sum(cyl=6), sum(cyl=8)
       FROM mtcars
       GROUP BY mpg")

# reshape
mycar = mtcars[,c("mpg","cyl","make")]
reshape(mycar, timevar ="cyl", idvar="make", v.names="mpg",
        direction="wide")
#this would take us back to the original dataset
reshape(wd, direction="long")

sqldf("SELECT make,
          sum(case when(cyl==4) then mpg else 0 end) as 'c4',
          sum(case when (cyl==6) then mpg else 0 end) as 'c6',
          sum(case when (cyl==8) then mpg else 0 end) as 'c8'
      FROM mycar GROUP BY make")
```

Other operations also have natural counterparts: assume `df1` and `df2` are two data frames with the same schema; then

```
# rbind
rbind(df1, df2)
sqldf("SELECT * FROM df1 union all SELECT * FROM df2")
```

produce equivalent result. Also, if `df3` and `df4` are two data frames (with arbitrary schemas),

```
# merge.
merge(A, B)
sqldf("SELECT * FROM A, B")
```

also produce equivalent results.

6.1.4 *Packages: Advanced Data Analysis*

One of the reasons for the popularity of R is that it has a large number of commands that are specifically designed for data exploration, cleaning, pre-processing, and analysis tasks. Thus, actions that require a whole query in SQL (sometimes a complex one) can be done in one line in R. Also, R is adept at generating simple graphical displays for data, which can be very effective in EDA.

The command `summary`, with a dataset as argument, will make the system generate basic statistics about the input. It can also be applied to the result of any analysis.

Histograms are generated by the function `hist`. The partition of the beans can be specified by passing a vector of values as a second, optional argument: a vector with values $(a_0, \ldots, a_k)$ will create bins (a_i, a_{i+1}) for $i = 0, .., k-1$. A boxplot can be generated by the function `boxplot`, which takes 1 or several attributes as parameters or even a whole dataset (categorical attributes are turned into numerical by enumerating them, which does not always make sense).

Plots are generated by the `plot` command: with two arguments, each an attribute, it will generate the scatterplot of both attributes. With a complex dataset as an argument, it will generate all scatterplots, one for each pair of attributes in the dataset. If an attribute is categorical, the function `as.numeric(attr)` will transform it into a numerical attribute for the plot. Jitter can be added to the scatterplot by calling `plot(jitter(attr))`.

A distance matrix for a dataset can be calculated with `dist(dataset)`. Correlation coefficients can be calculated as follows: assume `attr1` and `attr2` are two attributes. Then `cor.test(attr1,attr2,method="spearman")` will calculate the Spearman coefficient (this can also be used for "kendall" and "pearson").

Finally, complex analysis that was not doable in R without some serious programming can be accomplished using *packages*. A package is a file containing functions defined in R for some specific purpose. For instance, there is a library `outliers` that has some test of outliers. The available packages are too many to mention; the interested reader is referred to https://cran.r-project.org/web/packages and to https://rstudio.com/products/rpackages/ for more information.

6.2 SQL and Python

Python is an all-purpose programming language that has become extremely popular within the data analysis community, due to its built-in facilities to deal with datasets and to the large number of libraries developed to do sophisticated data analysis within a Python program. In this section we show how Python programs can be designed to interact with databases.

6.2.1 Python and Databases: DB-API

To read data from a relational database from Python, there is a direct connection using the DB-API, a collection of functions that can be used in any Python program to bring data from the database to the program, manipulate said data, and store it (or other results) back in the database.

Since Python is an object-oriented language (at least in its current version), the DB-API works by creating objects and using the methods attached to those objects. Two types of objects are used: *connection* objects and *cursor* objects. A DB-API access works as follows:

1. first, the program must establish a connection to the database. This is achieved through the `connect` method that takes in a URL, a user name, and a password and returns a connection object. The method establishes a connection with the database server at the given URL, as a user with the credentials provided.
2. the connection object can be used to create a cursor object, with the `cursor()` method. The cursor object is the one that will be used to send and execute SQL commands.
3. The cursor object has an `execute` method that takes in as parameter a string representing an SQL command. This command is sent to the database server and executed.
4. When the command is an SQL SELECT, a result is returned to the cursor object. This result can be retrieved with a variety of `fetch` commands:

   - `fetchall()` will take the whole result and deposit it in a Python variable of type array, with each element of the array representing a record and being itself an array. A typical access with this method will use `fetchall()` to deposit the result of a query in a Python list and then iterate over the list with a `for` loop, doing whatever is needed on each row (see example below). However, this can be quite slow if the result returned from the database is very large.
   - `fetchone()` will pick a row of the result at random. The typical approach is to call `fetchone()` inside a loop. This works for large datasets but makes access row-based. When using this function, the value `None` is returned when no more rows are left; a typical loop (assuming variable `cur` is the cursor object) is:

     ```
     while True:
         record = cur.fetchone()
         if record == None:
             break
         .... action on the retrieved row ...
     ```

   - An intermediate approach is to use `fetchmany(size=n)` where n tuples are returned at once (when there are fewer than n rows left, the method simply returns; however, many rows are left). This method is also used inside a loop but can be more efficient than `fetchone()`.

Calls to all these methods can be combined; whenever any `fetch` method is used, it consumes a certain number of tuples and subsequent calls consume the rest of the tuples. The cursor itself keeps track of which tuples have been consumed.

Example: MySQL Access with Python

```
import MySQLdb
#create a connection with database
conn = MySQLdb.connect('DatabaseName', user='username',
                       password='password')

#create a cursor object to handle interaction with database
cursor = conn.cursor()

#provide an SQL statement to be executed (note the quotes)
cursor.execute('''SELECT * FROM Table
                  WHERE attribute = 'value'''')

#get the result of the previous query
result = cursor.fetchall()

#iterate over the result, one row at a time
for row in results:
      firstAttribute = row[0]
      secondAttribute = row[1]
      thirdAttribute = row[2]
#manipulated retrieved results as needed
      print "first attribute=%s,
             second attribute =%s,
             third attribute = %d" % \
         (firstAttribute, secondAttribute, thirdAttribute)

#close the cursor when finished
cursor.close()

#close the connection when finished
conn.close()
```

The cursor object has an attribute, `description`, that contains metadata—in particular, after a query, this is basically the schema of the returned answer; it is `None` for other commands. The description is a sequence of attribute descriptions; each description contains the name, type, and other information of the attribute.

```
cursor.execute("SELECT * FROM TABLE;")

# metadata from query
columns = cursor.description
```

```
# iterate over attributes
for column in columns
    print "attribute name =%s, attribute type =%d, %
    (column[0], column[1])
```

Another attribute, `rowcount`, tells us how many rows are returned by a query (or how many rows are affected by an INSERT or UPDATE).

Other accesses, besides queries, are also possible:

```
#create a database
cursor.execute('Create database Example')

#use that database as the current one
cursor.execute('use Example')

#create a table in database
cursor.execute('create table TName(
                        first-att char(30) primary key,
                        second-att int))')

#put some data into table
cursor.execute('insert TName values (%s %d)', ('foo', 30))

#make sure the insertion is in the database
conn.commit()
```

One nice characteristic of DB-API is that accesses to any relational system is pretty much the same, as can be seen comparing the previous example (which used a MySQL database) to the next one (which uses a Postgres database).

Example: Postgres Access with Python

```
import psycopg2 as dbapi2

#establish a connection
db = dbapi2.connect (database="dbname", user="username",
                     password="password")

# get a cursor from the connection
cur = db.cursor()

#use the cursor to send a query
cur.execute ("SELECT * FROM TableName");

# get the result from the query
rows = cur.fetchall()

#iterate over the result
for i, row in enumerate(rows):
    print "Row ", i, "value = ", row
```

```
#close the cursor once done
cur.close()

#close the connection once done
db.close()
```

Besides a result, an `execute` command returns a status variable telling the program if there has been an error. Hence, it is good programming style to wrap a call `execute` within a `try` block:

```
try:
   # Execute the SQL command
   cursor.execute(sql)
   # only if the sql command included changes,
   db.commit()
except:
   # only if the SQL command included changes,
   db.rollback()
```

With SELECT statements, no commit is needed (and if there is an error, no rollback is needed) as no changes are made to the database while querying. However, it is still a good idea to check the status variable before looping over the result since, if something went wrong, there is no result to loop over.

There is also a module to connect to a SQLite database.

Example: Using SQLite

```
import sqlite3
conn = sqlite3.connect('example.db')

c = conn.cursor()
# Create table
c.execute('''CREATE TABLE mytable (id int, name text)''')
# Insert data
c.execute("INSERT INTO mytable VALUES (1, 'Jones')")
c.execute("INSERT INTO mytable VALUES (2, 'Smith')")
c.execute("INSERT INTO mytable VALUES (3, 'Lewis')")
# Save  the changes
conn.commit()

for row in c.execute('SELECT * FROM mytable WHERE id > 1'):
    print(row)
```

It is also possible to create an array of rows, say `records` and call a single statement to do multiple insertions at once, using method `executemany`:

```
c.executemany('INSERT INTO mytable VALUES (?,?)', records)
```

Besides using DB-API, there are several other approaches, like `alchemy` and `django`. For more information, consult
https://docs.python-guide.org/scenarios/db/.

6.2.2 Libraries and Further Analysis

A common use of Python is to get some data from the database and carry out some tasks for which the database is ill-suited, in particular visualizations and complex analysis. This is achieved in Python through the use of *libraries*, collections of functions written by a programmer with a specific purpose in mind.

Some of the most common and useful libraries include:

- *NumPy* (Numerical Python), which provides operations on n-dimensional arrays and matrices, including many numerical routines on which other libraries and modules build;
- *SciPy* (Scientific Python), which builds upon NumPy and contains modules for linear algebra, optimization, integration, and statistics.
- *Pandas*, a package designed to define series (one-dimensional tables: each row is indexed and contains a single value) and data frames (two-dimensional tables: each row is indexed but contains several values) and provides tools for manipulation, wrangling, aggregating, and analyzing both of them.
- *SciKit-Learn*, built on top of SciPy and offering an array of Machine Learning capabilities.
- *NLTK* (Natural Language Toolkit), a library that provides tools for Natural Language analysis, including complex processing like tagging, text classification, named entity recognition, and parsing.
- *Gensim* is another natural language analysis library, but it focuses on sophisticated Information Retrieval methods like Latent Dirichlet Allocation (LDA) and distributed semantic representations (word2vec).
- *Matplotlib*, which provides multiple tools for data visualization.
- *Bokeh*, another data visualization library, specialized in interactive graphics.

There are many more libraries available.

Note that many of these assume that their data is in a certain format (tabular, most of the time) and that it has been cleaned and corrected. Thus, before using any of the functions on any of these libraries, it is a good idea to examine the data and make sure that it is clean and in the expected format, either using SQL or Python code to carry out data pre-processing.