RSQLAlchemy

The objective was to create a package that abstracts sqlite databases in an object oriented programming (OOP) framework. Users of RSQLAlchemy will be able to invoke methods associated with the OOP representations of databases, tables and records to process data in lieu of sending SQL commands to the database. My contribution to this package was to abstract database connections, tables, records along with methods for querying and writing to databases.

Overview: Database connections were encapsulated as classes wrapped around RSQLite connections. The table and record abstractions were achieved using data frames. The choice to use data frames was influenced by the simplicity of the data frame structure and the variety of functions available for processing data frame content.

Details: A function **mapTable** and four reference classes: **control**, **engine**, **session** and **tableRecords** were created to facilitate the read/write functionalities of RSQLAlchemy.

```
control:=> a class to track internal variables
engine:=> a class to abstract database connections
mapTable:=> a function to abstract tables and records as OOP objects
tableRecords:=> a class to abstract table contents
session:=> a class that provides a workspace for RSQLAlchemy applications
```

RSQLAlchemy::control:=> a class to track internal variables

The *control* class provides a neat way of hiding internal variables from the end user. Associated with each OOP abstraction, is a set of internal variables tracking the linkage between abstraction and corresponding sqlite object. As new database objects are introduced during an RSQLAlchemy application, this list of internal variables grows. If left in the global environment these internal variables could clobber the user's workspace, this is why the *control* class was created to keep the internal variables hidden.

The *control* class has a field called *variables* (implemented as a list) that tracks all internal variables. It also has a method *addVariable* which saves new internal variables.

```
# initialize a control
myControl = control$new()

# set a variable
myControl$addVariable("projectName",  "RSQLAlchemy")

# fetch a variable
myControl$variables[["projectName"]]  #returns 'RSQLAlchemy'
```

At the start of an RSQLAlchemy session, a *control* is initialized and saved as *RSQLAlchemyControl*. All internal variables created in the application are saved to this control *RSQLAlchemyControl* . Likewise, all variables are also sourced from this control, *RSQLAlchemy*.

RSQLAlchemy::engine:=> a class to abstract database connections

The *engine* connects the application to a sqlite database. The engine class has one field *databaseName* which holds the connection string to the sqlite database. It has two methods: *query(sqlStatement)* and *insert(tableName, dFrame)*. *query()* runs the sql statement *sqlStatement* against a sqlite database while *insert()* writes the values held in data frame *dFrame* to a sqlite table named *tableName*. These two functions are the low level database read and write functions of RSQLAlchemy.

```
# initialize an engine
testDB<-paste(system.file("exec",package="RSQLAlchemy"),"/test.db",sep="",collapse="")
myEngine=engine(databaseName=testDB)

# run a sql statement to read from database
myEngine$query('select * from snp')

# write to a database
twoTableRecords<-
    as.data.frame(cbind(snp_id=c("SNP01","SNP02"),chr = c("1","2"),all_A=c("A","T"),all_B=c("C","C")))
myEngine$insert(tableName='snp',dFrame=twoTableRecords)
```

RSQLAlchemy::mapTable:=> a function to abstract tables and records as OOP objects

mapTable maps the structure of a database table to an OOP object (reference class). The function is equivalent to the mapper process of the python package, SQLAlchemy. Consider a simple database table *snp* that has columns *snp_id*, *chr*, *all_A* and *all_B* and the *snp_id* column as its primary key.

snp_id	chr	all_A	all_B
SNP01	1	T	C
SNP02	1	A	C

mapTable can be called to abstract the *snp* table as a reference class. See illustration below. The class created using *mapTable* captures the column names, column types and primary keys of the *snp* table by virtue of the *columns* and *primaryKey* arguments. The *tableName* argument draws the link between the **snp reference class** and the **snp sqlite table** .

```
#map snp table structure to snp class
snp<-mapTable(
      tableName="snp",
      columns=' snp_id="character", chr="character", all_A="character", all_B="character"',
      primaryKey="snp_id");

#create instances of snp (records for snp table)
SNPA <- snp(snp_id='SNPA', chr="1", all_A="T", all_B="C")
```

RSQLAlchemy::tableRecords:=> a class to abstract table contents with methods for data processing

It is with this class where the bulk of data processing occurs. While *mapTable* abstracts the structure of tables, *tableRecords* abstracts the content of tables. *tableRecords* has methods for filtering, ordering, sampling (limit, offset, all, first, one, scalar) and counting records. The table contents are saved as data frames in the *dataframe* field and the methods associated with *tableRecords* exploit existing data frame functions for filtering, ordering, sampling and counting.

```
# create a tableRecords using quakes data
quakesTableRecords <- tableRecords(dataFrame = quakes)

## list and scalars (sampling records)
quakesTableRecords  #prints entire table
quakesTableRecords$count()
quakesTableRecords$all()  #returns every observation
quakesTableRecords$first()  #returns first observation
quakesTableRecords$limit()[1:5, ]
quakesTableRecords$offset()[5:10, ]

# order records
quakesTableRecords$orderBy("stations")
quakesTableRecords$orderBy("stations", "mag")  #multiple columns

# filter observations returned from query
quakesTableRecords$filterBy(stations == 15)  #equals
quakesTableRecords$filterBy(stations != 15)  #not equals
quakesTableRecords$filterBy(stations %like% "1")  #like
quakesTableRecords$filterBy(stations %! like% "1")  #not like
quakesTableRecords$filterBy(stations %in% c(10, 15))  #in
quakesTableRecords$filterBy(stations %! in% c(10, 15))  #not in
quakesTableRecords$filterBy(is.null(stations))  #is null
quakesTableRecords$filterBy(!is.null(stations))  #is not null
quakesTableRecords$filterBy(stations == 15 | mag > 4)  #or
quakesTableRecords$filterBy(stations == 15 & mag > 4)  #and
```

RSQLAlchemy::session:=> a class that provides a workspace for RSQLAlchemy application

The *session* provides a workspace for connecting other RSQLAlchemy building blocks, (*control*, *engine*, *mapTable*, *tableRecords*). When the RSQLAlchemy blocks are connected together in a *session*, database communications, data processing and the management of RSQLAlchemy classes and instances are seamless in an application.

The makeup of a *session* consists of an *engine* (the database connection) and its *objects* (temporary collection of objects created during session). The *session* provides methods to save and remove objects: *add()*, *add_all()*, *delete()*, *delete_all()*. Other methods include *bind()*, a

call to link the *session* to an *engine*. The *session* also provides database read methods e.g. *query()*, *join()*, *outerjoin()*, *innerjoin()*, *leftjoin()* and *rightjoin()*.These read methods operate off *engine$query(sqlStatment)*. *commit()* writes all objects temporarily saved in *session* to the database. *commit()* employs *engine$insert()*.

```
# start a session
mySession <- session()

# save or delete objects from session
mySession$add(SNPA)  # add_all() saves multiple objects at once
mySession$delete(SNPA)  # delete_all() removes multiple objects
```

Bind the session to an engine to connect to a database.

```
# bind an engine to session
mySession$bind(engine(databaseName = testDB))
```

The read methods such as *query()* return results as *tableRecords* to facilitate further operations such as counts, filtering, etc.

```
# query database
mySession$query("snp")  #prints entire table
mySession$query("snp.snp_id")  #prints snp_id column in table
mySession$query("snp", "snp.snp_id")

# process query results
mySession$query("snp")$orderBy("all_A")  #tableRecords$orderBy() method applied to results of session$query()
mySession$query("snp")$filterBy(all_A == "A")
mySession$query("snp")$count()
```

Joins are implemented using the *base::merge* function.

```
# joins
mySession$join("snp", "genotype")  # outerjoin
mySession$join("snp", "genotype")$filterBy(snp.snp_id == genotype.snp_id)
mySession$innerjoin("snp", "genotype", joinOn = "snp.snp_id==genotype.snp_id")
mySession$leftjoin("snp", "genotype", joinOn = "snp.snp_id==genotype.snp_id")
mySession$rightjoin("snp", "genotype", joinOn = "snp.snp_id==genotype.snp_id")
```

commit() writes all objects saved in the session to the database using *engine$insert()*.

```
# commit
SNPINSERT <- snp(snp_id = "SNPINSERT", chr = "1", all_A = "T", all_B = "C")
mySession$add(SNPINSERT)
mySession$commit()  #inserts into snp table in database a row with values associated wtih SNPINSERT
```

Conclusion

RSQLALchemy package provides an OOP framework for basic database operations including reading, writing and processing contents of databases.