Assignment 2: Files, Dictionaries and Objects

Due: Electronically, by 11:55 PM on December 1, 2016

Introduction

We've covered many important Python concepts by this point, including lists, files, dictionaries and (basic) OOP. In this assignment, we apply most of these concepts to write a program that can combine and transform tables of data in order to answer questions about that data. The program you will write is called SQuEaL. It stands for **S**implified **Qu**ery **E**ngine **a**nd **L**anguage, but more importantly it's the noise that Dan Zingaro (at UTM) made when he came up with what he thinks is a super-cool idea for this assignment. We'll see!

Querying Data

The quantity and variety of data generated today requires that we have ways to organize that data and extract pieces of data that are currently of interest. For example, you might have thousands of emails in your inbox right now, but you have ways of searching and filtering that data to find messages from certain people, with specific keywords, or sent on specific dates.

Nowhere is the importance of data more evident in our daily lives than when we interact with online retailers. Consider the typical interaction with a website like Amazon. Perhaps you're looking for a new CS book to read over the holidays. You might search for *Computer Science* in the search box. Among the millions of products available, Amazon returns only those products of relevance in the search results.

When you purchase a book, you are required to create a customer account. Amazon stores your account information along with this order and all orders you make in the future. Later, you can check on your order history to see exactly what you've purchased. Of course, millions of other customers have purchased stuff too, so Amazon has to maintain all of this data and make sure that you see only your data and another person sees only theirs. Amazon will also want to use this data for internal purposes to see how much they've sold, what products are most popular, what kinds of ads to serve to you, etc. Data is used to answer a wide variety of questions such as these:

- What are the Computer Science books available right now?
- Does the current user have a customer account?
- What is the order history for the current customer?
- How many products did Amazon sell this year?
- What are the top ten most popular books this year?

Websites like Amazon use databases to store these vast amounts of data. They then make *queries* on that data in order to retrieve data for a variety of purposes. One sample query might be: "give me all of the Computer Science books"; another might be: "give me this customer's order history, from oldest order to most recent order". There are languages used to make such queries on databases. The most popular of these languages is SQL (Structured Query Language), and if you take CSCC43 you'll learn a lot more SQL.

In this assignment, you'll write a program that understands a SQL-like language for making queries of data. Don't worry: we won't mention SQL again in this handout, and we expect no outside knowledge of databases or SQL (OOPS!) or Amazon or anything like that. Everything is in this handout. **Warning: if you know some SQL, you should be extra careful to do what we are asking and not make assumptions based on what you might have learned elsewhere.**

Tables and Databases

The first two important terms for us are *table* and *database*. From now on, when we use these terms, they mean what is described in this section.

Below is a sample table that we will call `movies`. It contains information on movies; try to understand what the table is showing before continuing to the text below.

movies

m.year	m.title	m.studio	m.gross
1997	Titanic	Par.	2186.8
2003	The Lord of the Rings: The Return of the King	NL	1119.9
2010	Toy Story 3	BV	1063.2

The table starts with a header giving the names of each column. The column names are `m.year`, `m.title` `m.studio`, and `m.gross`. (Ignore the `m.` prefix for now; you'll see why it's there soon.) Note that the header isn't one of the data rows. The data rows are the rows below the header. The table has four columns, and each row gives a value in each column. For example, we see that the first movie is Titanic, the year it was made was 1997, the studio that produced it was Par. (short for Paramount), and the gross revenue (amount the movies made) for the movie was 2186.8. (Yes, that's given in millions of dollars... anyone who hasn't dropped CS to enroll in film school, please continue reading.)

So that's a *table*: a grid where each column has a column name and each row gives a value for each column. Each row corresponds to a single thing/object/entity; in this example, each row corresponds to a movie.

A *database* is a collection of one or more tables. A database might consist of only one table, such as `movies`. But we can expand our database to include more tables. For example, we can add a second table, `oscars`, so that we have two tables in our database:

oscars

o.year	o.category	o.title
2010	Animated Feature Film	Toy Story 3
2003	Directing	The Lord of the Rings: The Return of the King
1997	Directing	Titanic
1997	Best Picture	Titanic

This table gives names of movies that won Academy Awards (also knows as "The Oscars", the year they

won, and the category in which they won. Notice that the names are the same ones as in the `movies` table above. This is no coincidence. Stay tuned.

The two tables `movies` and `oscars` is another example of a database; this database has two tables.

SQuEaL: Operations on Tables

In this section, you'll learn the operations that can be performed on tables and how these operations are written in SQuEaL. Throughout, keep one thing in mind: each operation on a table creates a new table.

Selecting Columns

One thing we can do on a table is select some or all of its columns. For example, consider the `movies` table (the first table given in this handout). We might be interested in only the names of the movies, so we might want a table with only the `m.title` column. We could express this in SQuEaL as:

```
select m.title from movies
```

and this would give us a table with just one column:

one column from movies

m.title
Titanic
The Lord of the Rings: The Return of the King
Toy Story 3

Or maybe you want only the movie names and year (and not the studio or gross revenue). You'd express this as:

```
select m.title,m.year from movies
```

and you'd get a table with two columns, `m.title` and `m.year`:

two columns from movies

m.year	m.title
1997	Titanic
2003	The Lord of the Rings: The Return of the King
2010	Toy Story 3

You may have noticed that the columns in this table are in the "wrong" order. However, that doesn't matter. The order of the columns does not change the data in the table. You will see in the next section that we will use dictionaries to store columns, so the order of columns is not preserved. This is expected behaviour.

Moving on: there are two ways to get all of the columns. You could list them all separated by commas:

```
select m.title,m.studio,m.gross,m.year from movies
```

or you can use a shorthand; the `*` means "all columns":

```
select * from movies
```

In both cases, you get a new table that contains exactly the same information as `movies`.

Multiple Tables

Selecting columns from a table is nice, but it doesn't let us use the full capacity of our data stored in the database. Recall that a database can have multiple tables, and that these tables might be meaningfully related. For example, our `movies` table contains the name of the movie, but the `oscars` table contains additional information on awards that movie won (we'll give you the data to make more related tables, such as ratings and actors who starred in the various movies).

Look back at those two tables. It would be nice to be able to create a single table that joins those two tables into one, like this:

movies join oscars

m.title	m.studio	m.gross	o.category
Titanic	Par.	2186.8	Directing
Titanic	Par.	2186.8	Best Picture
The Lord of the Rings: The Return of the King	NL	1119.9	Directing
Toy Story 3	BV	1063.2	Animated Feature Film

Notice that, for each row, we have some of the data from the `movies` and some of the data from oscars (`o.category`).

You should be able to perform these *joins* by hand. The column on which we are joining the two tables is the column that contains the common information. In this case, both tables have a `title` column, so we join on that. Consider the first row of `movies`. The title of the film is Titanic, so we're going to join with all rows of the `oscars` table where the `o.title` is Titanic. In this case, there are two rows where `o.title` is Titanic, the 3rd and 4th (best picture and directing for 1997). Joining the first row from `movies` and that third row from `oscars` gives us the first row for our new table:

movies join oscars

m.title	m.studio	m.gross	o.category
Titanic	Par.	2186.8	Directing

The first row of `movies` would also join with the 4th row of `oscars`, to produce the 2nd row of our new table. Likewise, the 2nd and 3rd rows of `movies` would join with the 2nd and 1st rows of `oscars` respectively, to produce our final table.

Here is the SQuEaL syntax for performing the join described here:
`select m.title,m.studio,m.gross,o.category from movies,oscars where m.title=o.title`
You should notice two things. First, after `from`, we now have the names of *two* tables. In addition, we have added the keyword `where` followed by a condition that must be true in order for a row to appear in the table. `m.title=o.title` means that a row from the movie table can be joined to a row of the oscar table exactly when the title columns are equal. (This is why we have the `m.` and `o.` prefixes. We can't have two columns

with the same name or we wouldn't have a way to refer uniquely to them!)

STOP! Make sure you understand this join stuff before continuing. Be sure you can work through questions like the following, or you'll have conceptual trouble later:

- What if there were an oscar winner from the `oscar` table that didn't have a row in the `movies` table? Would that winner show up in the new table?
- What if there were a movie in the `movies` table that didn't win any awards in the `oscars` table? Would that movie show up in the new table?

s
p
o
i
l
e
r

The answer to both questions is *no*. For example, in the second case, we'd be looking at a movie in the `movies` table and trying to connect that row with a row in the `oscars` table. But since that movie's title isn't in `oscars`, we find no match, and hence can't add a row to the join table. (believe it or not, making money doesn't mean that a movie is any good -insert your own Twilight/Michael Bay joke here-).

Your Tasks

Your task for this assignment is in three parts. In the first part, you will create objects and methods to allow you to work with tables and dictionaries, then you will write functions to read a database (one or more tables), finally you will write a program that accepts SQuEaL queries from the keyboard, runs them on a database, and outputs the results.

Starter Code

Please download the Assignment 2 Starter Files and extract the zip archive. The archive contains starter Python code files and some sample tables that constitute databases (see below).

Part 1: Building Table and Database Objects

First, you will write the code to complete the `Table` and `Database` classes in `database.py`. You will have to decide how to implement these classes, including what instance variables and methods they should have. The two methods that we will insist upon both classes having is `get_dict`, and `set_dict`, whose DocStrings have already been provided, and deal with dictionary representation of tables. These methods will allow us to more easily test your code. You should NOT be using these in your own code.

A table, is represented as a dictionary where each key is the name of a column and each value is the list of items in that column from the top row to the bottom. For example, the movies dictionary would contain four key/value pairs. Here is one of them: the key is `'m.year'` and the value is `['1997','2003','2010']`, another would have the key `'m.title'` mapped to the value `['Titanic','The Lord of the Rings: The Return`

`of the King','Toy Story 3'].`

A database can be represented as a dictionary where the keys are table names and the values are table objects.

Before you spend time building the methods for `Table` and `Database`, we suggest that you read the rest of the handout so you better understand what you'll have to do with these classes. Building sensible and well thought-out methods will make your life much easier when completing the next part of the assignment.

Part 2: Reading Tables and Databases

Next, you will write some reading-related functions in `reading.py`.

Write the following functions in `reading.py`:

- `read_table: (str) -> Table`

  The parameter is the name of a table file (note: this time, it's the name of the file, and not the file handle). A table file is a comma-separated file such as `movies.csv`. (Do not open `csv` files in Excel or some other spreadsheet program. They are text files. Open them in a text editor -OTHER THAN NOTEPAD- to see them.) It is your task to understand the structure of this type of file in order to read it properly. The function returns a Table object.

- `read_database: () -> Database`

  The database on which this function operates consists of **all** of the table files in the current directory. These files all have an extension of `.csv`. This function reads each file and returns a Database object representing the data from all csv files in the current directory. For a file named `x.csv`, the name of the table is `x`; that is, the `.csv` is not part of the table name. Note that each column name in the database is guaranteed to be unique. That is, if there is a column called `x` in one table, then there is guaranteed not to be column `x` in another table. (This is why we use the `m.`-style prefixes on column names: so that our column names are not shared by multiple tables in the database.)

  How are you to obtain a list of all of the `.csv` files in the directory? Use `glob.glob('*.csv')` to obtain a list of these filenames. We have included this bit of code to demonstrate this for you, so you are just provided with a list of the names of the files thats you need to process.

  Note that there should be no `input` and no `print` in `reading.py`. Two functions and that's all. If you run `reading.py` and it prints something or requests user input, something is wrong.

Part 3: Running Queries

In `squeal.py`, you will write a program that:

1. Reads the database in the current directory.
2. Reads SQuEaL queries from the keyboard until a blank line is entered. Each query is guaranteed to be of the proper syntax. You **do not** need to try to verify that the query has the correct syntax; it is guaranteed to be a well-formed query, and it is fine if your program crashes on malformed queries. After running the query, continue with the next input prompt.

SQuEaL Syntax

Here is the syntax of valid SQuEaL queries. When we speak of a *token*, we mean a piece of the query that you get by calling `split` on the query. In order, the query contains:

- The token `select`. This token is required.
- A comma-separated list of one or more column names. There are no spaces in this list, there are no duplicate column names in this list, and all columns are guaranteed to exist in the table. The special token `*` (instead of a comma-separated list of column names) means "all columns".
- The token `from`. This token is required.
- A comma-separated list of one or more table names. There are no spaces in this list, there are no duplicate table names in this list, and all tables are guaranteed to exist in the database.
- The token `where`. This token is **optional**. If not present, then the query is finished here and no more tokens will be present. If the token `where` is present, then it is followed by a `where` clause that looks like this:
    - A comma-separated list of one or more constraints. There are no spaces in this list, and each column mentioned in the constraints is guaranteed to exist. Each constraint is of one of the following four forms: `column_name1=column_name2`, `column_name1>column_name2`, `column_name1='value'`, or `column_name1>'value'` (you can also implement < if you wish, but we won't be testing it). Single quotes are used to indicate that the second operand is a hard-coded value rather than the value of another column in the row. Notice that there are no spaces around the operators. Only these two operators are allowed. Also, note that only `column_name1='value'` and `column_name1>'value'` are valid, `'value'=column_name` and `'value'>column_name` are not valid constraints (i.e., the value can only come **after** the operator)

A Required Function: `run_query`

We're not specifying much in terms of required functions in `squeal.py`. It is your responsibility to break the work into meaningful, self-contained functions that perform well-defined and small tasks, and we will be marking your design. That said, we are requiring that you write two particular functions. The first of which is:
`run_query: (Database, str) -> Table`

This function takes a Database object, and a query (in the form of a string) as its parameters. Runs the given query on the given database, and returns a table representing the resulting table.

Another Required Function: `cartesian_product`

The second function you must write is this:
`cartesian_product: (Table, Table) -> Table`

The **Cartesian product** of two tables is a new table where each row in the first table is paired with every row in the second table. For example, the Cartesian product of the `movies` and `oscars` tables is the following.

cartesian product

m.year	m.title	m.studio	m.gross	o.year	o.category	o.title
					Animated	

1997	Titanic	Par.	2186.8	2010	Feature Film	Toy Story 3
1997	Titanic	Par.	2186.8	2003	Directing	The Lord of the Rings: The Return of the King
1997	Titanic	Par.	2186.8	1997	Directing	Titanic
1997	Titanic	Par.	2186.8	1997	Best Picture	Titanic
2003	The Lord of the Rings: The Return of the King	NL	1119.9	2010	Animated Feature Film	Toy Story 3
2003	The Lord of the Rings: The Return of the King	NL	1119.9	2003	Directing	The Lord of the Rings: The Return of the King
2003	The Lord of the Rings: The Return of the King	NL	1119.9	1997	Directing	Titanic
2003	The Lord of the Rings: The Return of the King	NL	1119.9	1997	Best Picture	Titanic
2010	Toy Story 3	BV	1063.2	2010	Animated Feature Film	Toy Story 3
2010	Toy Story 3	BV	1063.2	2003	Directing	The Lord of the Rings: The Return of the King
2010	Toy Story 3	BV	1063.2	1997	Directing	Titanic
2010	Toy Story 3	BV	1063.2	1997	Best Picture	Titanic

Note that there are three rows in `movies` and four rows in `oscars`, so we get 3*4 = 12 rows in the cartesian product table. Of course, most of these 12 rows are probably not what you want, as the titles don't match. The way that these nonsense rows get removed is through a suitable `where` clause that constrains which rows are kept. For example, with a `where` clause of `m.title=o.title`, you end up with only four rows, as expected.

Although it isn't very efficient (and certainly isn't the way things are implemented in a real database system), you **must** process the SQuEaL query by first using your `cartesian_product` function to join all the tables in the query into a huge table and then keep the "good" rows and columns for the query based on the `where` and `select` clauses.

You may have noticed in reading the SQuEaL Syntax that one or more tables are allowed in the `from` part of the query. That is, you must support queries that contain one or two or three or more tables. Don't overcomplicate this. First produce the Cartesian product on the first two tables; that gives you a new table. Then, do the Cartesian product on that new table and the third table in the list of tables; that gives you a new table again, and so on. You can use `imdb.csv` or `oscar-actors.csv` in your starter files if you want to try a query that involves three tables.

What to do with a Query

So, a user runs `squeal.py` and types a query, like `select * from movies` or whatever. What does your program have to do to process this query?

The first thing you want to do is break the query into tokens by calling `split`. Remember: the query is guaranteed to have proper syntax!

Now, start with the first table after the `from`; call that your base table. If there is a second table in the tables list, compute the Cartesian product of the base table and that second table; that results in another table (call it `t2`). If there is a third table in the tables list, compute the Cartesian product of `t2` and that third table. Keep doing this until you've finished with all of the tables. When you're done, you have a table containing all of the columns from all of the tables that are involved in the query.

Now, apply any constraints in the `where` clause. Each constraint from left to right results in a new table that keeps only the rows that satisfy the constraint. Processing the first constraint removes 0 or more rows; processing the second constraint takes the result of the first constraint and removes a further 0 or more rows, and so on. We see many opportunities for helper functions here: processing one constraint, processing all constraints, finding operators, handling values vs. column names, etc.

Finally, create a new table where you keep only those columns that were listed after the `select`. (Another opportunity for good design with a helper function!) The resulting table is the result of the query. Be careful: even though `select` is listed first in the query (it is the first token), it is the last thing you do in processing the query!

Once you've generated the table for the query, you want to output that table to the screen with commas separating columns. We have provided a helper function in your starter code that outputs a table in the proper format.

When are Tables Equivalent?

If two tables have the same rows and the same column names, but the column order is different, then the tables are equivalent.

Similarly, if two tables have the same rows and the same column names, but the order of the rows is different, then the tables are equivalent.

So, you can take a table, change the order of the columns, change the order of the rows, and it will still be the same table. What does this mean? It means that the order in which you add rows during a Cartesian product operation doesn't matter. As long as you have the correct rows in your table, it is fine.

Testing your Code

It is recommended that you test each function as you write it by creating small files, tables, or queries, depending on the function. This is preferable to doing lots of work, finding a bug, and then having no idea where in the code that bug was introduced.

We are not asking you to submit your testing code as part of your submission.

To help you test, we are also providing a type-checker. It is in your starter code (`typechecker.py`). This module simply tests that the required functions accept the proper number of parameters and return the proper type. Passing the type-checker says nothing of the correctness of your functions, and does not test any helper functions that you have written. (We haven't prescribed how you should write those, so we can't test their type contracts!)

More Examples

Looking for more databases on which to run your code? We've provided a bunch of sample `.csv` files. We've added much more detailed movie info, an olympic database, as well as a database of every type of food that was mentioned in the tv show Seinfeld (some databases are more practical than others). We may add new files to play with as we go, so check back periodically. If you come across some fun .csv files, feel free to let me know.

Marking

These are the aspects of your work that will be marked for Assignment 2:

- **Correctness:** Your functions should perform as specified.
- **Formatting style:** Make sure that you follow the PEP-8 Python style guidelines, and the general programming rules that we have introduced throughout the term.
- **Programming style:** Your variable names should be meaningful and your code as simple and clear as possible.
- **No magic numbers:**: remember this from Assignment 1? You should define constants and use them instead of magic numbers.
- **Commenting**: be sure to include accurate docstrings (that follow the design recipe we have been using) and include internal comments.
- **Design**: We expect a reasonable choice of functions/methods. Functions/methods should generally be short -- no longer than about 20-30 lines of code. Write helper functions/methods to meet this goal.

What to Submit

The very last thing you should do before submitting is to run the type-check module one last time. Otherwise, you could make a small error in your final changes before submitting that causes your code to receive zero for correctness. Make sure that any `print` or `input` statements are hidden behind `if(__name__ == "__main__")`, or else the type-checker (and more importantly the auto-marker) will wait for input when it imports your file, and pause indefinitely.

Submit `database.py`, `reading.py` and `squeal.py`. Your files must be named exactly as given here (i.e. no capital letters, and please don't change the filenames).