# Fall 2020 Computer Science 220
# Program Assignment 6

## Learning Objectives:

- Develop a Python program to practice file processing, functions, and Boolean logic.

## Assignment:

In a relational database management system, people write SQL queries to retrieve data from the database, where data are stored in tables. A table consists of rows and columns. Query results are also displayed in the form of a table. For example, the SQL query "`SELECT` *`column1, column2`* `FROM` *`table`*" will display all rows of *column1* and *column2* from a table.

In this assignment, you will write some functions to simulate the functionality of SQL queries. The table can be built from a CSV file. A CSV (comma-separated values) file allows data to be saved in a table format, and the first row usually is the header, which specifies the name for each column. For example, the following is the first 6 lines in iris.csv:

```
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5,3.6,1.4,0.2,Iris-setosa
```

The first row contains the column names, and then each subsequent row contains the values of the 5 fields/columns of an Iris flower sample.

After performing SQL query by calling your query functions, the results will also be in the form of a table. In Python, a table can be represented by a list of lists, also called a matrix or 2-dimensional array. For example, above table can be represented in Python as:

```
[['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
    'species'],
 ['5.1', '3.5', '1.4', '0.2', 'Iris-setosa'],
 ['4.9', '3', '1.4', '0.2', 'Iris-setosa'],
 ['4.7', '3.2', '1.3', '0.2', 'Iris-setosa'],
 ['4.6', '3.1', '1.5', '0.2', 'Iris-setosa'],
 ['5', '3.6', '1.4', '0.2', 'Iris-setosa']]
```

## Part A: Simple SELECT Queries

In this part you need write functions that perform simple SQL SELECT queries like:

| **SQL Query** | **Description** |
|---|---|
| `SELECT * FROM` *`table`* | Retrieve data from all columns in *table* |
| `SELECT` *`column(s)`* `FROM` *`table`* | Retrieve data from the specified columns in *table* |
| `SELECT COUNT(*) FROM` *`table`* | Retrieve the number of rows in *table* |

1. Write a function `select(columns, table, limit=10)` that accepts a list of column names, a table name (the name of the CSV file), and an optional row limit. The function should return the content of the specified columns in the first *limit* number of rows (e.g. the first 10 rows). If *columns* is empty, it should return the data from all columns (as if you had run the query "`SELECT * FROM` *`table`*"). If *limit* is 0, it should return all rows.

   For example, `select([], "iris.csv", 5)` should return a table containing the header row and the first 5 rows of data. An example of this output is given on the first page. Similarly, the call `select(["sepal_width", "species"], "iris.csv")` should return the following table:

   ```
   [['sepal_width', 'species'],
    ['3.5', 'Iris-setosa'],
    ['3', 'Iris-setosa'],
    ['3.2', 'Iris-setosa'],
    ['3.1', 'Iris-setosa'],
    ['3.6', 'Iris-setosa'],
    ['3.9', 'Iris-setosa'],
    ['3.4', 'Iris-setosa'],
    ['3.4', 'Iris-setosa'],
    ['2.9', 'Iris-setosa'],
    ['3.1', 'Iris-setosa']]
   ```

2. Write a function `select_count(table)` that accepts a table name and returns the number of rows (excluding the header row). For example, `select_count("iris.csv")` should return 150.

You may wish to write several helper functions to keep your solution organized and modular. Here are some suggested functions, which may also help you in Parts B and C:

- `get_indices(columns, header)`, which accepts two lists of column names (one specified by the user and one representing the full column list from the file), and returns the indices of the columns in the full list. For example, this should return `[1, 4]`:

  ```
  get_indices(['sepal_width', 'species'], ['sepal_length',
      'sepal_width', 'petal_length', 'petal_width', 'species'])
  ```

- `read_columns(columns, table)`, which accepts a list of column names and a table name (the name of a CSV file), and returns the data in those columns as a list of lists, as described above.

If you are familiar with Python's `csv` library, you may use `csv.reader()` to read the CSV files. However, you cannot use other libraries like pandas, sklearn, or numpy for this assignment.

## Part B: WHERE Clauses

In this part, you will write a function that performs SQL queries with WHERE clauses:

| **SQL Query** | **Description** |
|---|---|
| `SELECT` *`column(s)`*<br>`FROM` *`table`*<br>`WHERE` *`condition`* | Retrieve only data from the specified columns in *table* that meet the given condition. |

A *condition* takes the form of: *column operator value*, such as `"sepal_width > 3.5"`. Valid operators are =, !=, <, <=, >, and >=. A full SQL query with a WHERE might look like:

```
SELECT sepal_width, species
   FROM iris
   WHERE species = "Iris-virginica"
```

which retrieves sepal_width and species for those rows where species is Iris-virginica.

3. Write a function `select_where(columns, table, column, operator, value, data_type="string")` that works like `select()` except that it only returns rows that meet the specified condition. Assume *operator* is a string holding one of the 6 valid operators. If *data_type* is `"int"` or `"float"`, you will need to convert the data to that type before doing any comparisons. Notice that `'15' < '8'` produces a different result from `15 < 8` because strings are compared lexicographically rather than numerically.

Examples:

- `select_where(["sepal_width", "species"], "iris.csv", "species", "=", "Iris-virginica")`
  **Result:** Same as executing the SQL statement above, and returns 50 rows.

- `select_where(["sepal_width", "sepal_length", "species"], "iris.csv", "sepal_width", "<", 2.3, "float")`
  **Result:** Retrieves the sepal_width, sepal_length and species for all rows whose sepal_width is less than 2.3:

```
[['sepal_width', 'sepal_length', 'species'],
 ['2.0', '5.0', 'Iris-versicolor'],
 ['2.2', '6.0', 'Iris-versicolor'],
 ['2.2', '6.2', 'Iris-versicolor'],
 ['2.2', '6.0', 'Iris-virginica']]
```

Suggested helper function:

- `eval_condition(val1, operator, val2, data_type="string")`, which compares *val1* and *val2* using *operator* and returns a Boolean. For example, `eval_condition("4", ">", "2", "float")` would return true, and `eval_condition("Iris-virginica", "=", "Iris-setosa")` would return false.

## Part C (Extra Credit): Aggregate Functions

In SQL, an aggregate function performs a calculation on a set of values and produces a single value. Examples including averaging, summing, counting, and finding the min or max. In this optional part, you will write a function that performs a full SQL query including a WHERE clause and aggregate function.

| **SQL Aggregate Function** | **Description** |
|---|---|
| `AVG(`*`column`*`)` | Calculates the average of a numeric column |
| `SUM(`*`column`*`)` | Calculates the sum of a numeric column |
| `COUNT(`*`column`*`)` | Calculates the number of rows in a column |
| `MIN(`*`column`*`)` | Calculates the smallest value in a column |
| `MAX(`*`column`*`)` | Calculates the largest value in a column |

For example, a SQL query using `AVG()` might look like:

```
SELECT AVG(sepal_width)
   FROM iris
   WHERE species = "Iris-setosa"
```

4. Write a function `select_aggr(function, column, table, where_column, operator, value, data_type="string")` that works like `select_where()` except that it returns the result of applying *function* to *column* in the selected data.

Examples:

- `select_aggr("MAX", "sepal_width", "iris.csv", "species", "=", "Iris-virginica")`
  **Result:** 3.3

- `select_aggr("COUNT", "sepal_width", "iris.csv", "species", "=", "Iris-versicolor")`
  **Result:** 50

```
select_aggr("AVG", "sepal_width", "iris.csv", "species",
"=", "Iris-setosa")
```
**Result:** 3

Suggested helper function:

- `aggregate(function, values)`, which accepts the name of a function and a list of numeric values, and returns the result of applying the specified aggregate function to the values.

## Assignment Specifics

1. Implement the above functions as described. The inputs and outputs must work exactly as written, but the function internals are up to you. You can modify, omit, or add to the suggested helper functions however you wish.
2. Include docstrings in your functions that document their purpose, inputs, and outputs. Imagine you are creating a library and the only way other developers will know how to use your functions is through their documentation.
3. Write test cases for your functions as you go, and write the results to a file called `query_result.txt`. You can open the file in "append" mode so that subsequent writes do not overwrite your earlier results.

```
with open("query_result.txt", "a") as out_file:
```

**Submission:**
File you need to test your works is: iris.csv.
File to be submitted: `query.py` and `query_result.txt`.

**Policies:**
Please see the homework policy to review all policies for all homework submissions in CSCI220.