SQL FUNCTIONS

/*
CREATING TABLES & INSERTING LINES
/*

Int is a numerical value. Varchar is a variable character, meaning a text string; the number indicating how many spaces are allowed in the cell.

```
CREATE TABLE EmployeeDemographics
(EmployeeID int,
FirstName varchar(50),
LastName varchar(50),
Age int,
Gender varchar(50)
)
```

Next, populate the table with rows of data.

```
INSERT INTO EmployeeDemographics VALUES
(1001, 'Laura', 'Palmer', 17, 'Female'),
(1002, 'Dale', 'Cooper', 31, 'Male'),
(1003, 'Harry', 'Truman', 44, 'Male'),
(1004, 'Shelly', 'Johnson', 20, 'Female'),
(1005, 'Bobby', 'Briggs', 23, 'Male'),
(1006, 'Benjamin', 'Horne', 51, 'Male'),
(1007, 'Donna', 'Hayward', 20, 'Female'),
(1008, 'Audrey', 'Horne', 25, 'Female'),
(1009, 'Gordon', 'Cole', 45, 'Male')
```

Next, we do similarly for a salary table for different positions in the company.

```
Create Table EmployeeSalary
(EmployeeID int,
JobTitle varchar(50),
Salary int
)

Insert Into EmployeeSalary VALUES
(1001, 'Student', 45000),
(1002, 'FBI Special Agent', 36000),
(1003, 'Sheriff', 63000),
(1004, 'Waitress', 47000),
(1005, 'Jock', 50000),
(1006, 'Business Owner', 65000),
(1007, 'Student', 41000),
(1008, 'Student', 48000),
(1009, 'FBI Director', 42000)

SELECT *
FROM EmployeeSalary
```

/*

SELECT & FROM STATEMENTS

/*

Retrieve the first selected rows if the dataset is too large. In this case, five.

```
SELECT TOP 5 *
FROM EmployeeDemographics
```

Retrieves each unique value in a column.

```
SELECT DISTINCT(EmployeeID)
FROM EmployeeDemographics
```

Retrieve the total amount of all non-null values in a column.

```
SELECT COUNT(LastName) AS LastNameCount
FROM EmployeeDemographics
```

Determine the max, min, and average for a given salary.

```
SELECT MAX(Salary)
FROM EmployeeSalary

SELECT MIN(Salary)
FROM EmployeeSalary
SELECT AVG(Salary)
FROM EmployeeSalary
```

/*

/WHERE STATEMENTS

/*

The where statement helps limit the amount of data and specify what data you want returned. Let's find all the employees named "Laura".

```
SELECT *
FROM EmployeeDemographics
WHERE FirstName = 'Laura'
```

Excluding all "Lauras".

```
SELECT *
FROM EmployeeDemographics
WHERE FirstName <> 'Laura'
```

Find all employees older than thirty, inclusive.

```
SELECT *
FROM EmployeeDemographics
WHERE Age >= 30
```

And function: find all male employees younger than thirty-two, inclusive.

```
SELECT *
```

```
FROM EmployeeDemographics
WHERE Age <= 32 AND Gender = 'Male'
```

Or function: only one criterion needs to be correct to produce a result.

```
SELECT *
FROM EmployeeDemographics
WHERE Age <= 32 OR Gender = 'Male'
```

Like function: here, we want to find all employees whose last name starts with the letter S. % is a wildcard.

```
SELECT *
FROM EmployeeDemographics
WHERE LastName LIKE 'H%'
```

A wildcard at the beginning finds anyone with an S in their surname, not just starting with.

```
SELECT *
FROM EmployeeDemographics
WHERE LastName LIKE '%H%'
```

Locate null values and non-null values in a column.

```
SELECT *
FROM EmployeeDemographics
WHERE FirstName IS NULL

SELECT *
FROM EmployeeDemographics
WHERE FirstName IS NOT NULL
```

In function: a condensed way to say = for multiple things.

```
SELECT *
FROM EmployeeDemographics
WHERE FirstName IN ('Laura', 'Dale')
```

/*
<u>GROUP BY/ORDER BY</u>
/*

Returns the total amount of men and women.

```
SELECT Gender, COUNT(Gender)
FROM EmployeeDemographics
GROUP BY Gender
```

```
Returns the number of men and women of a specified age.
```

```
SELECT Gender, Age, COUNT(Gender)
FROM EmployeeDemographics
GROUP BY Gender, Age
```

List of employees, ordered by age. The default in SQL is ascending, so for descending we add the DESC modifier.

```
SELECT *
FROM EmployeeDemographics
ORDER BY Age DESC
```

List of employees, ordered by both age and gender.

```
SELECT *
FROM EmployeeDemographics
ORDER BY Age, Gender

/*
JOINS
/*
```

In joins, the left table is the one first specified (demographics) and the right table is the second (salary).

```
SELECT *
FROM EmployeeDemographics
INNER JOIN EmployeeSalary
ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID

SELECT *
FROM EmployeeDemographics
FULL OUTER JOIN EmployeeSalary
ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID

SELECT *
FROM EmployeeDemographics
LEFT OUTER JOIN EmployeeSalary
ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID

SELECT *
FROM EmployeeDemographics
RIGHT OUTER JOIN EmployeeSalary
ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID

SELECT EmployeeDemographics.EmployeeID, FirstName, LastName, JobTitle, Salary
FROM EmployeeDemographics
INNER JOIN EmployeeSalary
ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID

/*
USE CASES
/*
```

In order to meet his quota, Gordon needs to deduct pay from his highest earning employee, besides himself. Let’s pull up a “full outer join” and select everything. From here we can determine which columns we need to look at.

```
SELECT *
FROM EmployeeDemographics
FULL OUTER JOIN EmployeeSalary
```

```
ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID

SELECT EmployeeDemographics.EmployeeID, FirstName, LastName, Salary
FROM EmployeeDemographics
INNER JOIN EmployeeSalary
ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID
WHERE FirstName <> 'Gordon'
ORDER BY Salary DESC
```

The accountant has made a mistake with the salary. We want to calculate the average salary for the student. All we will need is the job title and salary.

```
SELECT JobTitle, Salary
FROM EmployeeDemographics
INNER JOIN EmployeeSalary
ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID
WHERE JobTitle = 'Student'
SELECT JobTitle, AVG(Salary)
FROM EmployeeDemographics
INNER JOIN EmployeeSalary
ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID
WHERE JobTitle = 'Student'
GROUP BY JobTitle

/*
UNION FUNCTIONS
/*


First, update existing tables and create a new one.

Insert into EmployeeDemographics VALUES
(1011, 'Josie', 'Packard', 29, 'Female'),
(NULL, 'Lucy', 'Moran', NULL, NULL),
(1013, 'Pete', 'Martell', NULL, 'Male')
Create Table WareHouseEmployeeDemographics
(EmployeeID int,
FirstName varchar(50),
LastName varchar(50),
Age int,
Gender varchar(50)
)
Insert into WareHouseEmployeeDemographics VALUES
(1013, 'Pete', 'Martell', NULL, 'Male'),
(1050, 'Hawk', 'Hill', 39, 'Male'),
(1051, 'Phillip', 'Jeffries', 44, 'Male'),
(1052, 'Margaret', 'Lanterman', 48, 'Female')

First, we want to join these tables together.

SELECT *
FROM EmployeeDemographics
FULL OUTER JOIN WarehouseEmployeeDemographics
ON EmployeeDemographics.EmployeeID = WarehouseEmployeeDemographics.EmployeeID

This will show you both tables with the columns lined up.
```

```
SELECT *
FROM EmployeeDemographics

SELECT *
FROM WarehouseEmployeeDemographics
```

This combines them. (“Union All” combines everything, including duplicates.)

```
SELECT *
FROM EmployeeDemographics
UNION
SELECT *
FROM WarehouseEmployeeDemographics
SELECT *
FROM EmployeeDemographics
UNION ALL
SELECT *
FROM WarehouseEmployeeDemographics
ORDER BY EmployeeID
```

The following two tables have very different information, so the columns don’t align.

```
SELECT *
FROM EmployeeDemographics

SELECT *
FROM EmployeeSalary
ORDER BY EmployeeID
```

We can still use the union function, which will yield similar text string columns, however most of the results will be funky.

```
SELECT EmployeeID, FirstName, Age
FROM EmployeeDemographics
UNION
SELECT EmployeeID, JobTitle, Salary
FROM EmployeeSalary
ORDER BY EmployeeID

/*
CASE STATEMENTS
/*

SELECT FirstName, LastName, Age,
CASE
       WHEN Age > 30 THEN 'Old'
       WHEN Age BETWEEN 27 AND 30 THEN 'Young'
       ELSE 'Baby'
END
FROM EmployeeDemographics
WHERE Age is NOT NULL
ORDER BY Age

SELECT *
FROM EmployeeDemographics
JOIN EmployeeSalary
       ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID
```

```sql
SELECT FirstName, LastName, JobTitle, Salary
FROM EmployeeDemographics
JOIN EmployeeSalary
	ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID
```

Management has determined to give every employee a raise, but not every employee will receive the same amount. Create a case statement to calculate their salary after they get their promotion.

```sql
SELECT FirstName, LastName, JobTitle, Salary,
CASE
	WHEN JobTitle = 'Student' THEN Salary + (Salary * .10)
	WHEN JobTitle = 'FBI Special Agent' THEN Salary + (Salary * .05)
	WHEN JobTitle = 'Business Owner' THEN Salary + (Salary * .000001)
	ELSE Salary + (Salary * .03)
END AS SalaryAfterRaise
FROM EmployeeDemographics
JOIN EmployeeSalary
	ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID
```

```sql
/*
HAVING CLAUSE
/*
```

Retrieve job titles and how many people in the company have those titles.

```sql
SELECT JobTitle, COUNT(JobTitle)
FROM EmployeeDemographics
JOIN EmployeeSalary
	ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID
GROUP BY JobTitle
```

Retrieve job titles that have more than one employee.

```sql
SELECT JobTitle, COUNT(JobTitle)
FROM EmployeeDemographics
JOIN EmployeeSalary
	ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID
GROUP BY JobTitle
HAVING COUNT(JobTitle) > 1
```

Find job titles that have an average salary of $45K.

```sql
SELECT JobTitle, AVG(Salary)
FROM EmployeeDemographics
JOIN EmployeeSalary
	ON EmployeeDemographics.EmployeeID = EmployeeSalary.EmployeeID
GROUP BY JobTitle
HAVING AVG(Salary) > 45000
ORDER BY AVG(Salary)
```

```sql
/*
UPDATING/DELETING DATA
/*
```

We want to update an existing row of data from the Employee Demographics table. Since Lucy Moran has null values in her row, we want to complete her statistics.

```
UPDATE EmployeeDemographics
SET EmployeeID = 1012
WHERE FirstName = 'Lucy' AND LastName = 'Moran'
```

This has updated the ID number. However, there remains a null value for age and gender. To correct this, we run the following query.

```
UPDATE EmployeeDemographics
SET Age = 36, Gender = 'Female'
WHERE FirstName = 'Lucy' AND LastName = 'Moran'
```

Moving on to the DELETE function, this will delete an entire row from our table. Keep in mind, there is no way to reverse a delete function, so be very careful in using it. There is no recovery for accidents here. A good safeguard is to use the select function in place of deleting to verify you aren't making a mistake.

That said, if we wanted to delete a given employee row, this is how we would do it.

```
DELETE FROM EmployeeDemographics
WHERE EmployeeID = 1005
```

```
/*
ALIASING
/*
```

Useful for making queries legible for third parties. Let's say we want to combine the two name columns into one column designated FullName.

```
SELECT FirstName + ' ' + LastName AS FullName
FROM EmployeeDemographics
```

Often when using aliasing with the select function, it is when we are using aggregate functions. In the following example, the output will give us an average age, but no column title. This isn't as clear as it can be for end-users.

```
SELECT Avg(Age)
FROM EmployeeDemographics
```

The fix? Simple. The addition of an "AS" qualifier will tell the audience what this information is.

```
SELECT Avg(Age) AS AvgAge
FROM EmployeeDemographics
```

```
/*
PARTITION BY
/*
```

We start by bringing up the two tables we want to extract data from.

```
SELECT *
FROM EmployeeDemographics

SELECT *
FROM EmployeeSalary
```

Then, join them by the ID as demonstrated.

```
SELECT *
FROM EmployeeDemographics dem
JOIN EmployeeSalary sal
       ON dem.EmployeeID = sal.EmployeeID
```

In using the partition, we can isolate the column we want to perform the aggregate function on. In this case, gender. This can be more useful than the group by function, because here we are able to include other columns like names and salary that provide additional context to what we want to produce.

```
SELECT FirstName, LastName, Gender, Salary
, COUNT(Gender) OVER (PARTITION BY Gender) as TotalGender
FROM EmployeeDemographics dem
JOIN EmployeeSalary sal
       ON dem.EmployeeID = sal.EmployeeID
/*
CTE's
/*
```

Common Table Expressions are useful for keeping code legible while performing multiple aggregations on data. Unlike temp tables, its life is limited to the current query and isn't stored anywhere. Defined by using the with clause.

```
WITH CTE_Employee as (
SELECT FirstName, LastName, Gender, Salary
, COUNT(Gender) OVER (PARTITION BY Gender) as TotalGender
, AVG(Salary) OVER (PARTITION BY Gender) as TotalSalary
FROM EmployeeDemographics emp
JOIN EmployeeSalary sal
       ON emp.EmployeeID = sal.EmployeeID
WHERE Salary > '45000'
)
SELECT *
FROM CTE_Employee

/*
TEMP TABLES
/*

CREATE TABLE #temp_Employee (
EmployeeID int,
JobTitle varchar(100),
Salary int
)

SELECT *
FROM #temp_Employee

INSERT INTO #temp_Employee VALUES (
'1001', 'Student', '45000'
)
```

The temp table lives somewhere, like a stored procedure. A useful trick in ensuring you can run the queries multiple times without erroring out is by dropping the table if it exists.

```
DROP TABLE IF EXISTS #temp_Employee2
CREATE TABLE #temp_Employee2 (
JobTitle varchar(50),
EmployeesPerJob int,
AvgAge int,
AvgSalary int
)

INSERT INTO #temp_Employee2
SELECT JobTitle, COUNT(JobTitle), AVG(Age), AVG(Salary)
FROM EmployeeDemographics emp
JOIN EmployeeSalary sal
       ON emp.EmployeeID = sal.EmployeeID
GROUP BY JobTitle

SELECT *
FROM #temp_Employee2

/*
STRING FUNCTIONS
/*
```

Let’s say we have a table with a bunch of errors in them: extra spaces, improper punctuation, unnecessary attributions, etc. We can use some of the more common string functions to correct the errors. We start by creating a practice table.

```
CREATE TABLE EmployeeErrors (
EmployeeID varchar(50)
,FirstName varchar(50)
,LastName varchar(50)
)

Insert into EmployeeErrors Values
('1001  ', 'Laurie', 'Plamer')
,('  1002', 'Dale', 'Coooper')
,('1005', 'Bubby', 'BRiggs - Fired')

Select *
From EmployeeErrors
```

To create a new column without the blank spaces on either side of the text (or specifically left or right, depending on the error), we can run the following queries as needed.

```
Select EmployeeID, TRIM(employeeID) AS IDTRIM
FROM EmployeeErrors

Select EmployeeID, RTRIM(employeeID) as IDRTRIM
FROM EmployeeErrors

Select EmployeeID, LTRIM(employeeID) as IDLTRIM
FROM EmployeeErrors
```

To remove unwanted text from a cell, we can use the replace function.

```sql
Select LastName, REPLACE(LastName, '- Fired', '') as LastNameFixed
FROM EmployeeErrors
```

Using substrings. Useful when using “fuzzy matching”. We want like things to match across tables, even if there are minor errors in one or the other.

```sql
Select Substring(err.FirstName,1,3), Substring(dem.FirstName,1,3),
Substring(err.LastName,1,3), Substring(dem.LastName,1,3)
FROM EmployeeErrors err
JOIN EmployeeDemographics dem
       on Substring(err.FirstName,1,3) = Substring(dem.FirstName,1,3)
       and Substring(err.LastName,1,3) = Substring(dem.LastName,1,3)
```

By using upper and lower functions, we can make the text uniform in either format.

```sql
Select firstname, LOWER(firstname)
from EmployeeErrors

Select Firstname, UPPER(FirstName)
from EmployeeErrors
```

```sql
/*
STORED PROCEDURES
/*
```

Stored procedures can be accessed by multiple users across the network and can have various input data. This reduces network traffic and in turn increases performance. If updates are made, everyone in the network will receive them. This is reminiscent of group projects on a platform like Google Docs.

```sql
CREATE PROCEDURE Temp_Employee
AS
DROP TABLE IF EXISTS #temp_employee
CREATE TABLE #temp_employee (
JobTitle varchar(100),
EmployeesPerJob int ,
AvgAge int,
AvgSalary int
)

INSERT INTO #temp_employee
SELECT JobTitle, Count(JobTitle), Avg(Age), AVG(salary)
FROM master.dbo.EmployeeDemographics emp
JOIN master.dbo.EmployeeSalary sal
       ON emp.EmployeeID = sal.EmployeeID
group by JobTitle

SELECT *
FROM #temp_employee

EXEC Temp_Employee
```

```sql
/*
SUBQUERIES (VARIOUS USES)
/*
```

Subqueries can be described as a query within a query. Here are a few examples in the select, from, and where statements.

```
SELECT EmployeeID, JobTitle, Salary
FROM master.dbo.EmployeeSalary

SELECT EmployeeID, Salary, (SELECT AVG(Salary) FROM master.dbo.EmployeeSalary) AS
AllAvgSalary
FROM master.dbo.EmployeeSalary

SELECT EmployeeID, Salary, AVG(Salary) over () as AllAvgSalary
FROM master.dbo.EmployeeSalary

SELECT EmployeeID, Salary, AVG(Salary) AS AllAvgSalary
FROM master.dbo.EmployeeSalary
GROUP BY EmployeeID, Salary
ORDER BY EmployeeID

SELECT EmployeeID, JobTitle, Salary
FROM master.dbo.EmployeeSalary
WHERE EmployeeID in (
	SELECT EmployeeID
	FROM master.dbo.EmployeeDemographics
	WHERE Age > 30)
```