SQL

SQL is a standard language for accessing and manipulating databases.

1. **SQL stands for Structured Query Language**
2. SQL lets you access and manipulates databases
3. SQL can execute queries against a database
4. **SQL can retrieve data from a database**
5. **SQL insert, update and delete records in a database**
6. SQL can **create new databases**
7. SQL can **create new tables in a database**
8. SQL can create views in a database
9. SQL can create stored procedures in a database
10. SQL can set permissions on tables, procedures, and views

SQL is an ANSI/ISO standard, with different versions of the SQL language.

(MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.)

They all support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, WHERE)

To build a website that shows data from a database, you will need:

An RDBMS database program (i.e. MS Access, SQL Server, MySQL)

The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

Note:

SQL keywords are NOT case sensitive: select is the same as SELECT

Some database systems require a semicolon at the end of each SQL statement.

A semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

SQL Statements -

Select

1. **SELECT *column1*, *column2, ...***

   FROM *table_name*;

2. **SELECT * FROM *table_name*;**

Distinct - drops duplicates

1.SELECT DISTINCT *column1*, *column2, ...*

FROM *table_name*;

2.SELECT COUNT(DISTINCT Country) FROM Customers;

Where

1.SELECT *column1*, *column2, ...*

FROM *table_name*

WHERE *condition*;

2. WHERE Country='Mexico';

Operators

The WHERE clause can be combined with AND, OR, and NOT operators.

WHERE *condition1* AND *condition2* AND *condition3 ...*;

WHERE *condition1* OR *condition2* OR *condition3 ...*;

WHERE NOT *condition*;

WHERE Country='Germany' OR Country='Spain';

WHERE Country='Germany' AND (City='Berlin' OR City='München')

WHERE NOT Country='Germany' AND NOT Country='USA’;

ORDER BY: Sort ascending or descending (default ascending)

SELECT *column1*, *column2, ...*

FROM *table_name*

ORDER BY *column1, column2, ...* ASC|DESC;

ORDER BY Country;

ORDER BY Country DESC;

ORDER BY Country ASC, CustomerName DESC;

INSERT INTO: used to insert new records in a table

1.INSERT INTO *table_name* (*column1*, *column2*, *column3*, ...)

VALUES (*value1*, *value2*, *value3*, ...);

2. INSERT INTO *table_name*

VALUES (*value1*, *value2*, *value3*, ...);

IS NULL: check for na

WHERE *column_name* IS NULL;

WHERE *column_name* IS NOT NULL;

UPDATE: modify the existing records in a table.

UPDATE *table_name*

SET *column1* = *value1*, *column2* = *value2*, ...

WHERE *condition*;

UPDATE Customers

SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'

WHERE CustomerID = 1;

DELETE: delete existing records in a table.

DELETE FROM *table_name* WHERE *condition*;

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

SELECT TOP: Returning a large number of records

SELECT TOP 3 * FROM Customers;

MIN, MAX, AVG, SUM, COUNT

SELECT MIN(Price) AS SmallestPrice

FROM Products;

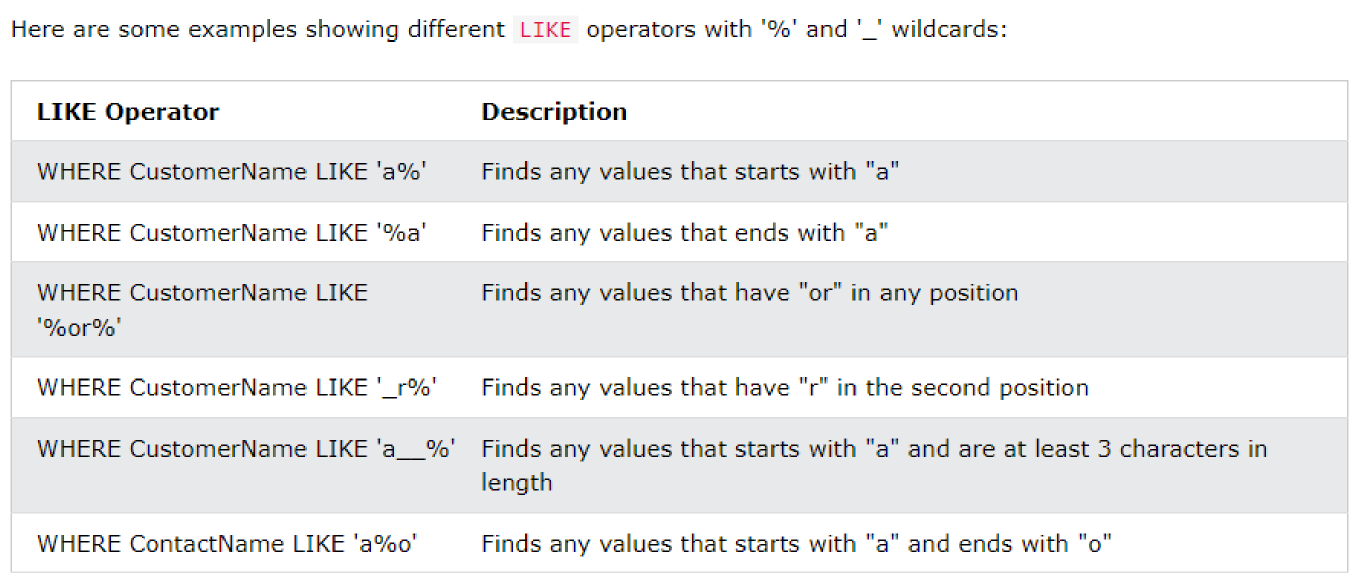
SELECT MIN(*column_name*)

FROM *table_name*

LIKE: Regex (patterns finding)

SELECT * FROM Customers

WHERE CustomerName LIKE 'a%';



Here are some examples showing different `LIKE` operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a__%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

IN: check inside

SELECT *column_name(s)*

FROM *table_name*

WHERE *column_name* IN (*value1*, *value2*, ...);

BETWEEN : values within a given range.

1.SELECT *column_name(s)*

FROM *table_name*

WHERE *column_name* BETWEEN *value1* AND *value2;*

2.WHERE Price BETWEEN 10 AND 20;

ALIASES: a temporary name.

1.SELECT *column_name* AS *alias_name*

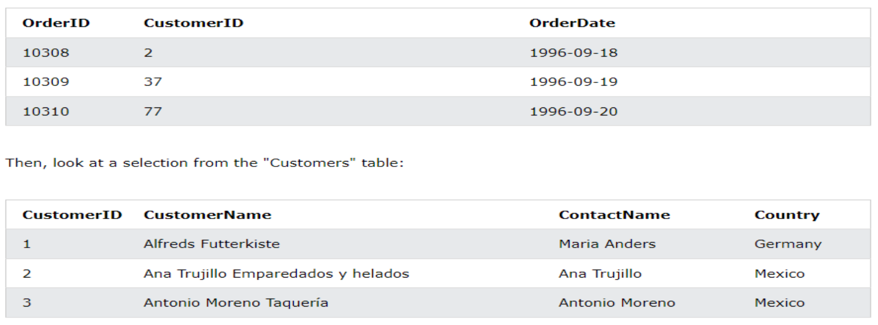
FROM *table_name;*

2.SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address

FROM Customers;

JOIN : combine rows from two or more tables, based on a related column between them.

INNER, LEFT, RIGHT, OUTER, FULL



OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

1.SELECT Orders.OrderID, Orders.CustomerID, Orders.OrderDate

FROM Orders

INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;

2. SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID

ORDER BY Customers.CustomerName;

SELF JOIN: its just a conditional filtering

SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City

FROM Customers A, Customers B

WHERE A.CustomerID <> B.CustomerID

AND A.City = B.City

ORDER BY A.City;

UNION: combine the result set of two or more SELECT statements (only distinct values)

UNION ALL: all values

SELECT City FROM Customers

UNION

SELECT City FROM Suppliers

ORDER BY City;

GROUP BY:

SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country;

HAVING

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions like group by

SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

HAVING COUNT(CustomerID) > 5;

EXISTS: the existence of any record in a subquery

1.SELECT *column_name(s)*

FROM *table_name*

WHERE EXISTS

(SELECT *column_name* FROM *table_name* WHERE *condition*);

2. SELECT SupplierName

FROM Suppliers

WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Suppliers.supplierID AND Price < 20);

SELECT INTO: copies data from one table into a new table.

SELECT * INTO CustomersBackup2017

FROM Customers;

INSERT INTO SELECT: copies data from one table and inserts it into another table

INSERT INTO Customers (CustomerName, City, Country)

SELECT SupplierName, City, Country FROM Suppliers;

CASE: Similar to if else condition

Instead of if use case and at last put end, when & then are used

```
1. SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'
    WHEN Quantity = 30 THEN 'The quantity is 30'
    ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

IF NULL: return an alternative value if an expression is NULL:

IFNULL(UnitsOnOrder, 0)

SQL Store Procedure: similar to functions

SYNTAX:

Stored Procedure Syntax

CREATE PROCEDURE ***procedure_name***

AS

sql_statement

GO;

Execute a Stored Procedure

```
EXEC procedure_name;
```

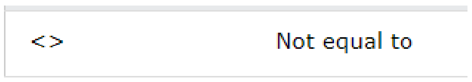
Example:

```
CREATE PROCEDURE SelectAllCustomers

AS

SELECT * FROM Customers

GO;

EXEC SelectAllCustomers;
```

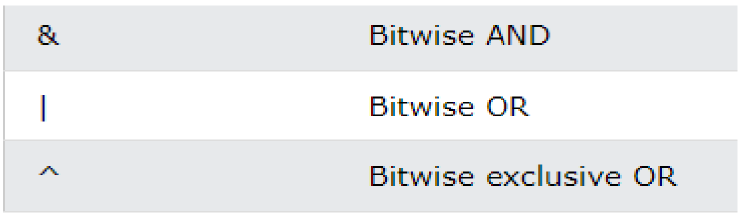
Comments : use double -

Example: `SELECT * FROM Customers -- WHERE City='Berlin';`

Operators:



<>	Not equal to



&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR

# Create & Manipulate database and Create new tables:

```
CREATE DATABASE databasename;

DROP DATABASE databasename;

BACKUP DATABASE databasename
TO DISK = 'filepath';

CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

DROP TABLE *table_name*;

ALTER TABLE *table_name*
ADD *column_name datatype*;

SQL constraints are used to specify rules for data in a table.

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

The following constraints are commonly used in SQL:

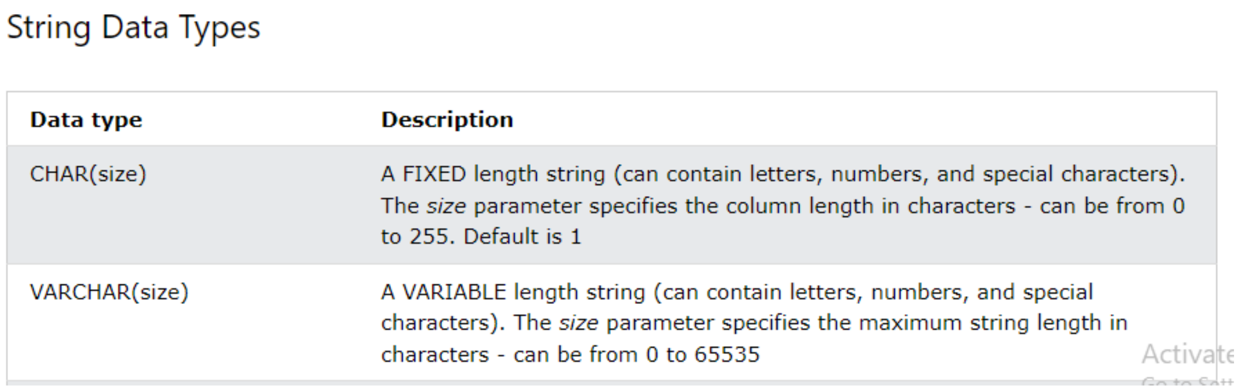
- `NOT NULL` - Ensures that a column cannot have a NULL value
- `UNIQUE` - Ensures that all values in a column are different
- `PRIMARY KEY` - A combination of a `NOT NULL` and `UNIQUE`. Uniquely identifies each row in a table
- `FOREIGN KEY` - Prevents actions that would destroy links between tables
- `CHECK` - Ensures that the values in a column satisfies a specific condition
- `DEFAULT` - Sets a default value for a column if no value is specified
- `CREATE INDEX` - Used to create and retrieve data from the database very quickly

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
   PRIMARY KEY (ID)
);

CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
   PRIMARY KEY (OrderID),
   FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```
(both containing same type of id no. but are different)

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
   CHECK (Age>=18));


CREATE INDEX idx_lastname
ON Persons (LastName);


CREATE TABLE Persons (
   Personid int NOT NULL AUTO_INCREMENT,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
   PRIMARY KEY (Personid)
);
```

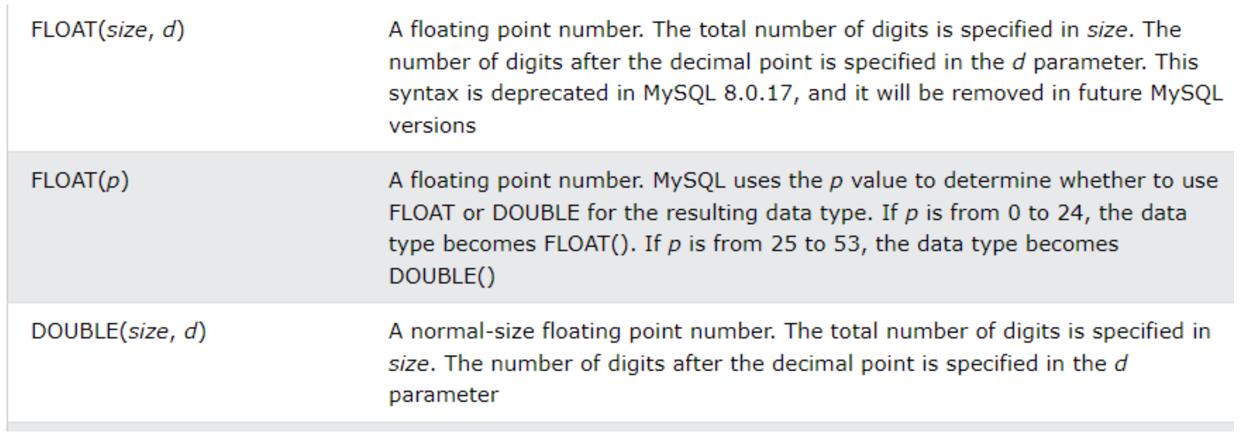
VIEW : view is a virtual table based on the result-set of an SQL statement.

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

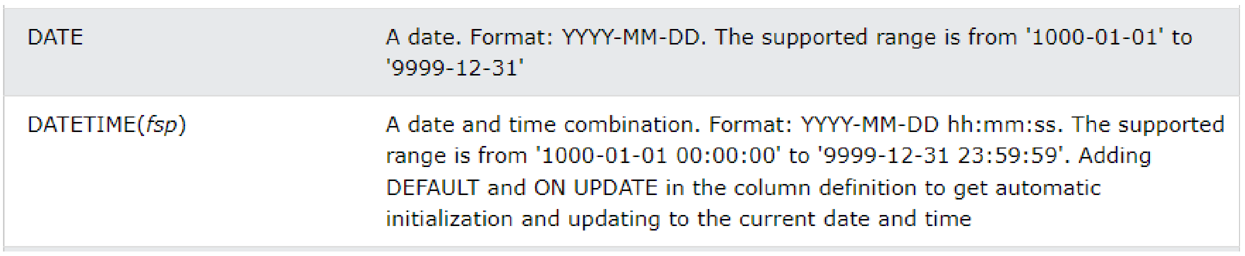


String Data Types

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The *size* parameter specifies the maximum string length in characters - can be from 0 to 65535



FLOAT(*size*, *d*)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions
FLOAT(*p*)	A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(*size*, *d*)	A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter



DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(*fsp*)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time

Python connection and example:
Ex:

1.UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley 345

2. "SELECT * FROM customers LIMIT 5"

Data Base Connections:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword"
)

print(mydb)
```

Create table:

```
mycursor = mydb.cursor()

mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address
VARCHAR(255))")
```

Table call and manipulations:

```
mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

SAS: Statistical Analysis Software

SAS have 2 steps:

1. DATA: Retrieve, Process, Store
2. PROC: Read, Analyze

SAS Shortcuts:
Run or submit a program: F3
Comment the selected code (/): Ctrl + /
Uncomment: ctrl + shift + /
Convert selected text to upper case: ctrl + shift + U
Convert selected text to lower case: ctrl + shift + L

The keywords are as follows:
1. DATA - Creating a new data set
2. INPUT - To define the variables used in the data set.
3. Dollar sign ($) - To identify the variable as a character. Ex: gender $
4. DATALINES or *CARDS* - DATALINES statement
5. PROC PRINT - To print the data set in the output window.
6. RUN - The step ends with a RUN statement.

The default delimiter is blank.
Need to define the delimiter before defining the variables using
INFILE and DLM = options.

```
DATA outdata;
   INFILE Datalines dlm =",";
   INPUT age gender $ dept obs1 obs2 obs3;
   Datalines;
1,F,3,17,6,24
```

PROC IMPORT: Import external files into SAS. **excel file, csv, txt etc.**

Import excel file into SAS
1. **OUT** - outdata is the data set saved in work library (temporary library)
2. **DBMS** - To specify the type of data to import.
3. **REPLACE** - To overwrite an existing SAS data set.
4. **SHEET** - To import a specific sheet from an excel workbook
5. **GETNAMES** - To include variable names from the first row of data.
6. **Range** : import data from range B2:D10 from sheet1

```
PROC IMPORT DATAFILE= "c:\deepanshu\sampledata.xls"
OUT= outdata
DBMS=xls
REPLACE;
SHEET="Sheet1";
GETNAMES=YES;
RUN;
```

Reading a CSV File

INFILE statement - To specify path where data file is saved.
DSD - To set the default delimiter from a blank to comma.
FIRSTOBS=2 : To tell SAS that first row contains variable names and data values starts from second row.

```
data outdata;
infile 'c:\users\deepanshu\documents\book1.csv' dsd firstobs=2;
input id age gender $ dept $;
run;
```

Note:
There are two library.
Use semi colon in each statement to close the argument.

1. WORK library *(temporary library)*
2. Input library *(Permanent library)*.

OUT = Age is smiliar to OUT = Work.Age .
OUT = Input.Age.

colon modifier : read variable "Name" until there is a space or other delimiter.
The $30. defines the variable as a character variable having max length 30.

```
data ex2;
input ID Name:$30. Score fee:$10.;
cards;
1 DeepanshuBhalla 22 1,000
2 AttaPat 21 2,000
3 XonxiangnamSamnuelnarayan 33 3,000
;
Run
```

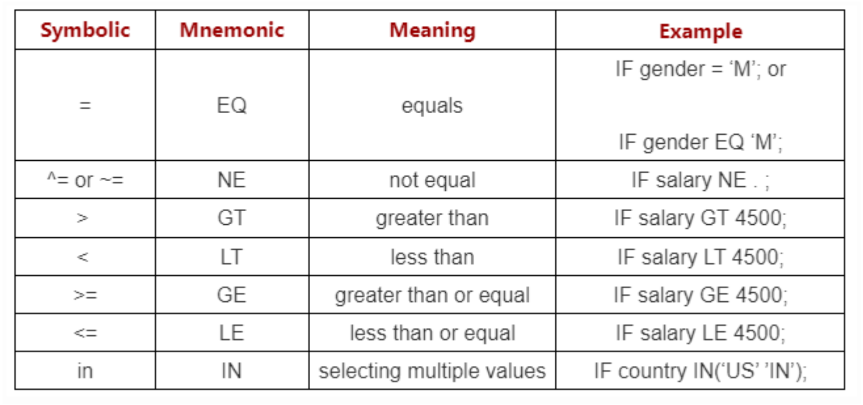
length
use a length statement prior to input statement to adjust varying length of a variable. In this case, the variable **Name** would be read first. **Use only $ instead of $30. after "Name" in INPUT statement.**

```
data example2;
length Name $30.;
input ID Name $ Score;
cards;
```

ampersand (&) : tell SAS to read the variable until there are two or more spaces
*input ID Name **&** $30. Score;*

USING SQL

PROC SQL;
*select * from outdata;*



Symbolic	Mnemonic	Meaning	Example

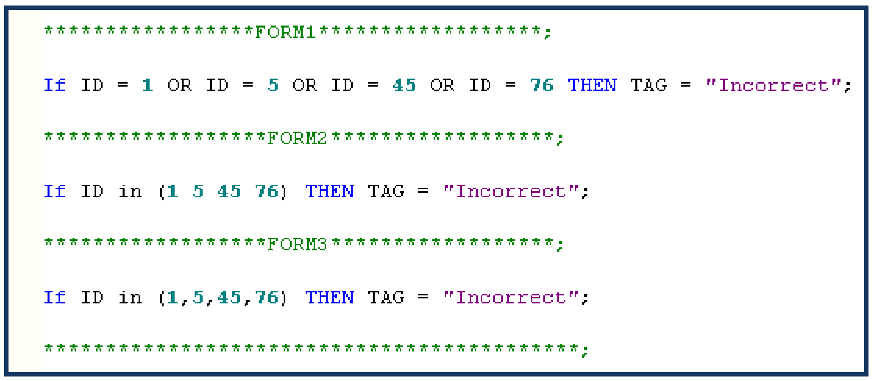
=	EQ	equals	IF gender = ‘M’; or
			
			IF gender EQ ‘M’;

^= or ~=	NE	not equal	IF salary NE . ;
>	GT	greater than	IF salary GT 4500;
<	LT	less than	IF salary LT 4500;
>=	GE	greater than or equal	IF salary GE 4500;
<=	LE	less than or equal	IF salary LE 4500;
in	IN	selecting multiple values	IF country IN(‘US’ ’IN’);

IF ELSE STATEMENTS
Data readin1;
Set readin;
IF ID LE 100 THEN TAG ="Old";
ELSE TAG ="New";
Run;

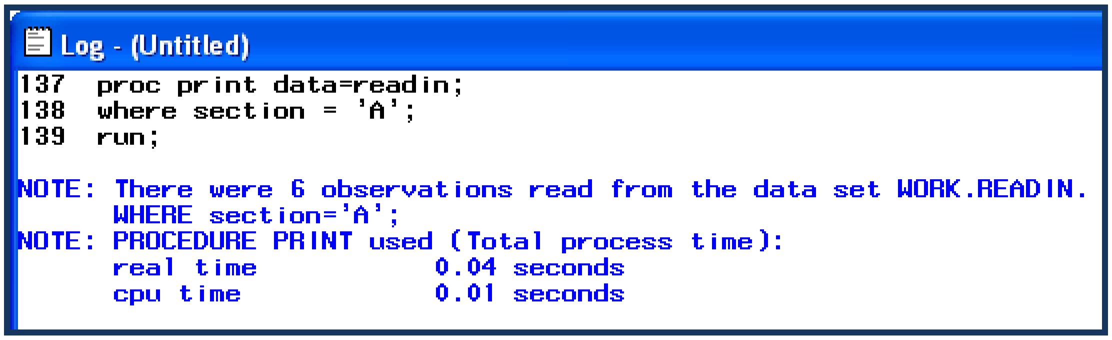
IF ELSE IF
IF ID < 75 THEN TAG ="Old";
ELSE IF 75 <= ID < 100 THEN TAG = "New";
ELSE IF ID >= 100 THEN TAG ="Unchecked";

MUTI CASE CHECK



```
*****************FORM1******************;

If ID = 1 OR ID = 5 OR ID = 45 OR ID = 76 THEN TAG = "Incorrect";

*****************FORM2******************;

If ID in (1 5 45 76) THEN TAG = "Incorrect";

*****************FORM3******************;

If ID in (1,5,45,76) THEN TAG = "Incorrect";

*******************************************;
```

The **WHERE statement** can be used in procedures to subset data while **IF statement** cannot be used in procedures.


Log - (Untitled)

```
137  proc print data=readin;
138  where section = 'A';
139  run;

NOTE: There were 6 observations read from the data set WORK.READIN.
      WHERE section='A';
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.04 seconds
      cpu time            0.01 seconds
```

KEEP: Keep all the variables start with 'X'

DATA READIN2;
SET READIN (KEEP = X:);
RUN;

COLON (:) all the variables starting with the character 'X'.

DATA READIN2;
SET READIN;
IF X_T =: '01';
RUN;

Use of WildCard in IN Operator
DATA READIN2;
SET READIN;
IF X_T IN: ('01', '02');
RUN;

SORT AND TRANSPORT
proc sort data = sashelp.class out=class;
by name sex;
run;

proc transpose data = sashelp.class out=temp;
by name sex;
var height weight;
Run;

MISSING VALUE
SAS stores 28 missing values in a numeric variable. They are as follows :

1. *dot-underscore* **._**
2. *dot* **.**
3. *.A through .Z (Not case sensitive)*

PROC FREQ
"/ MISSING" option on the tables statement, the percentages are based on the total number of observations (non-missing and missing) and the percentage of missing values are reported in the table.

```
PROC FREQ DATA= TEST;
TABLES X / MISSING;
RUN;
```

PROC MEANS

```
Proc Means Data = test N NMISS;
Var q1 - q5 ;
Run;
```

Similarly
PROC CORR
PROC REG
PROC LOGISTIC
PROC FACTOR