Table of Contents:

Week 1 Notes:

Databases are used in multiple places, such as:

Enterprise Information:

- Sales
- Accounting
- Human Resources
- Manufacturing
- Online Retailers

Banking and Finance:

- Banking
- Credit Card Transactions
- Finance

Other Applications:

- Universities
- Airlines
- Hospitals

History of Databases:

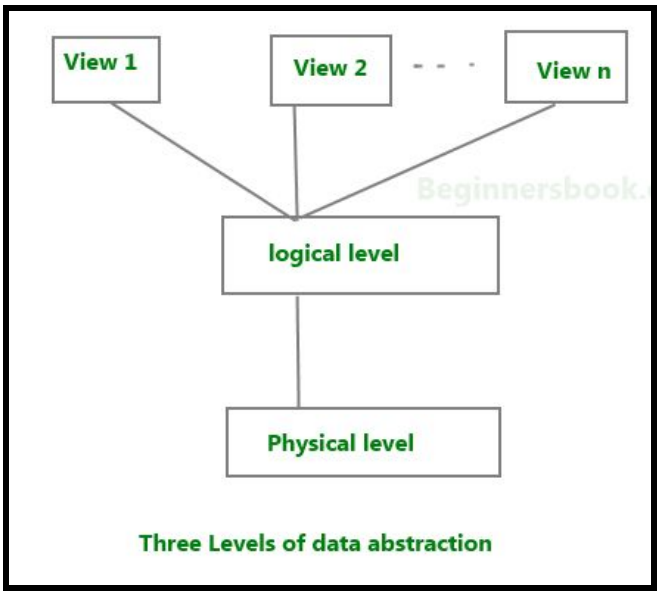
- In the 1960s data storage changed from tape to **direct access**. This allowed shared interactive data use.
- A tape drive provides **sequential access** storage, unlike a hard disk drive, which provides direct access storage. A disk drive can move to any position on the disk in a few milliseconds, but a tape drive must physically wind tape between reels to read any one particular piece of data. As a result, tape drives have very large average access times.
- Early databases were **navigational** which was very inefficient for searching. Edgar Codd created a new system in the 1970s based on the **relational model**.
- In a relational database, data is stored in tables and users access data by doing queries. In a navigational database, data is accessed by defining the path to find the desired data.
- In the late 1970s and early 1980s, SQL was developed based on the relational model and is the foundation of current databases.
- In the 2000s, with increasingly large datasets, new XML databases and NoSQL databases are becoming more prevalent.

Why use databases:

- Commercialized management of large amounts of data
- Ability to update and maintain data
- Keep track of relationships between subsets of the data
- Efficient access and searching capabilities
- Multiple users can access and share data
- Ability to limit access to a portion of the data according to user type and enables security of data
- Minimizes redundancy of multiple data sets
- Enables consistency constraints
- Allows users an abstract view of the data which hides the details of how the data are stored and maintained.

Data Abstraction:

- **Physical Level:**
  - The lowest level.
  - Describes how the data are actually stored.
  - You can get the complex data structure details at this level.
  - These details are often hidden from the programmers.
- **Logical Level:**
  - The middle level.
  - Describes what data are stored in the database and what relationships exist between the data.
  - Implementing the structure of the logical level may require complex physical low level structures. However, users of the logical level don't need to know about this. We refer to this as the **physical data independence**.
- **View Level:**
  - The highest level of abstraction.
  - Describes only a small portion of the database.
  - Allows user to simplify their interaction with the database system.
  - The user just interact with system with the help of GUI and enter the details at the screen, they are not aware of how the data is stored and what data is stored.



Relational Model:

- A **relational database** is a collection of tables each having a unique name.
- Each **table** is also known as a **relation**.
- **Rows** are referred to as **tuples**.
- **Columns** are referred to as **attributes**.
- **Database Schema:** The logical design of the database. It will give you all the tables in that database. It is the consolidation of all the relational schemas. It is the overall design of the database.

- **Database Instance:** A snapshot of the data in the database. It is the information stored at a particular moment in time. It is the collection of all the tuples of the database at any moment. The instance of a database frequently changes.
- **Relation Schema:** A list of attributes and their corresponding domains. It will never talk about the data, just the design. In a relation schema, you need to relay the name of the relation, the attributes and the primary keys. Typically, we underline the attributes that are the primary keys.
- The difference between database schema and relational schema is given in this example:
  Let's say we have 2 relations, A and B.
  A(<u>a1</u>, a2, a3)
  B(b1, <u>b2</u>, b3)
  Each individual is a relational schema, however, the consolidation of them is a database schema.
- It is useful to have the same attribute in multiple schema so that you can combine them.
- The **domain** is all the valid values which an attribute may contain. It is the data type of the column (E.g. int, str, etc).
- E.g. Suppose we have the following database:
  Instructor Relation

ID	Name	Department
1	Srinivasan	Comp. Sci.
2	Wu	Finance
3	Mozart	Music

  a. A tuple from the above database is:

2	Wu	Finance

  b. An attribute from the above database is:

Department
Comp. Sci.
Finance
Music

  c. The domain of the attribute department is string.
  d. The domain of the attribute ID is integer.
  e. The instructor relation has the schema: **instructor(ID, Name, Department)**.
  The syntax of a relation schema is: **relation_name(attribute1, attribute2, … attributen)**. Furthermore, you need to underline the primary key.

Keys:

- A **key** is an attribute or set of an attribute which helps you to identify a tuple in a relation. They allow you to find the relationships between two tables.
- We need keys because:
    1. Keys help you to identify any row of data in a table. In a real-world application, a table could contain thousands of records. Moreover, the records could be duplicated. Keys ensure that you can uniquely identify a table record despite these challenges.
    2. Allows you to establish a relationship between and identify the relation between tables
    3. Help you to enforce identity and integrity in the relationship.
- We will look at the following types of keys:
    1. **Super Key:** A set of one or more attributes that taken together uniquely identify a tuple in the relation. If an attribute is already a super key and other attributes get added to the set, then the set is still a super key.
       E.g. Suppose the attribute ID is a super key. If we add other attributes to the set, such as name and phone number, the new set is still a super key.
    2. **Candidate Key:** A super key with no repeated attribute. It is a minimal super key.
       Properties of Candidate key:
        - It must contain unique values
        - Candidate key may have multiple attributes
        - Must not contain null values
        - It should contain minimum fields to ensure uniqueness
        - Uniquely identify each record in a table
    3. **Primary Key:** A candidate key chosen to distinguish between tuples. It is usually denoted in a relation schema by an underline. There is only a single primary key.
    4. **Foreign Key:** A set of attributes in a relation that is a primary key in another relation. The foreign key does not have to be a primary key.
- **E.g.** Suppose we have the relation below:

Instructor Relation

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

a. Some superkeys for this would be:
    1. {ID}
    2. {name, dept_name}
    3. {ID, dept_name}
b. The candidate keys for this would be:
    1. {ID}
    2. {name, dept_name}

- **E.g.** Given the below relation schemas, find the foreign key(s). The primary key(s) are all underlined.
  A(a1, a2, a3)
  B(a1, b1, b2)

  Solution:
  The foreign key here is a1 in B. While it is not a primary key, a1 is a primary key in A.

- **E.g.** Given the below relation schemas, find the foreign key(s). The primary key(s) are all underlined.
  A(b1, b2, b3)
  B(b1, b2, b3)

  Solution:
  The b1 in A is not a foreign key because the primary key in B is
  {b1, b2, b3}. However, the b1 in B is a foreign key as b1 is a primary key in A.

- **E.g.** Given the below relation schemas, find the foreign key(s). The primary key(s) are all underlined.
  instructor(ID,name,dept_name,salary)
  department(dept_name,building,budget)
  teaches(ID,course_id,sec_id,semester,year)

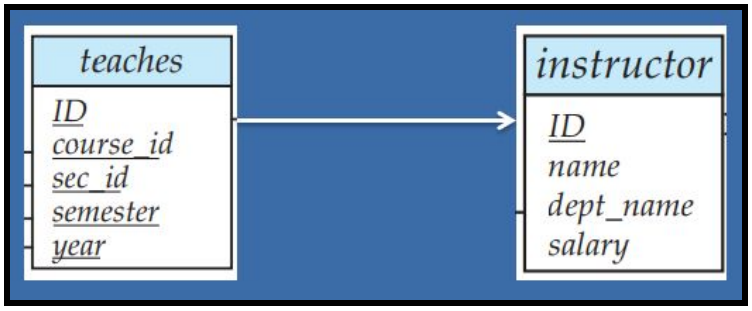
  Solution:
  The ID in teaches is the foreign key.
  We say ID from teaches references instructor.
  teaches is the referencing relation.
  instructor is the referenced relation.

- A **foreign key constraint** is the fact that a foreign key value in one relation must appear in the referenced relation. A foreign key cannot contain data that the primary key in the referenced relation doesn't have. If data in the primary key of the reference relation changes, that change won't be reflected in the foreign key. However, if a user tries to access that information from the foreign key, an error message will appear.

- **E.g.** Given the below relation schemas, find the foreign key constraint. The primary key(s) are all underlined.
  teaches(ID,course_id,sec_id,semester,year)
  section(course_id,sec_id,semester,year,building,room_number)

Solution:
course_id,sec_id,semester,year in teaches has a foreign key constraint on section. teaches is the referencing relation and section is the referenced relation.

- We can depict foreign key constraints and primary keys using a **schema diagram**. In a schema diagram, the relation is in light blue and the primary keys are underlined. Furthermore, there is an arrow from the foreign key attributes in the referencing relation pointing to the primary key of the referenced relation.
  Note: Suppose we have 2 relations, A and B, that both have foreign keys that reference each other.
  I.e. A(a1, a2, a3, b1) & B(b1, b2, b3, a1) → b1 in A is a foreign key and a1 in B is also a foreign key.
  In this case, there would be a double arrowed line connecting A and B.
  E.g.



Terminology:

- **Candidate Key:** A super key with no repeated attribute. It is a minimal super key.
  Properties of Candidate key:
  - It must contain unique values
  - Candidate key may have multiple attributes
  - Must not contain null values
  - It should contain minimum fields to ensure uniqueness
  - Uniquely identify each record in a table
- **Database:** A collection of interrelated data that is relevant to an enterprise.
- **Database Management System (DBMS):** A software package designed to define, manipulate, retrieve and manage data in a database. A DBMS generally manipulates the data itself, the data format, field names, record structure and file structure. It must be convenient and efficient.
- **Database Schema:** The logical design of the database. It will give you all the tables in that database. It is the consolidation of all the relational schemas. It is the overall design of the database.
- **Database Instance:** A snapshot of the data in the database. It is the information stored at a particular moment in time. It is the collection of all the tuples of the database at any moment. The instance of a database frequently changes.
- **Direct Access:** The ability to obtain data from a storage device by going directly to where it is physically located on the device rather than by having to sequentially look for the data at one physical location after another.
- **Domain:** All the valid values which an attribute may contain. It is the data type of the column (E.g. int, str, etc).

- **Foreign Key:** A set of attributes in a relation that is a primary key in another relation. The foreign key does not have to be a primary key.
- **Foreign key constraint:** The fact that a foreign key value in one relation must appear in the referenced relation. A foreign key cannot contain data that the primary key in the referenced relation doesn't have. If data in the primary key of the reference relation changes, that change won't be reflected in the foreign key. However, if a user tries to access that information from the foreign key, an error message will appear.
- **Key:** An attribute or set of an attribute which helps you to identify a tuple in a relation. They allow you to find the relationships between two tables.
- **Navigational Database:** A type of database in which records or objects are found primarily by following references from other objects.
- **Physical data independence:** Allows you to separate conceptual levels from the internal/physical levels. It allows you to provide a logical description of the database without the need to specify physical structures.
- **Primary Key:** A candidate key chosen to distinguish between tuples. It is usually denoted in a relation schema by an underline. There is only a single primary key.
- **Relational database:** A collection of tables each having a unique name.
- **Relational Model:** Represents the database as a collection of table values. A database organized in terms of the relational model is a relational database.
- **Relation Schema:** A list of attributes and their corresponding domains. It will never talk about the data, just the design. In a relation schema, you need to relay the name of the relation, the attributes and the primary keys. Typically, we underline the attributes that are the primary keys.
- **Sequential Access:** A method of retrieving data from a storage device by moving through all the information until the desired data is reached.
- **Super Key:** A set of one or more attributes that taken together uniquely identify a tuple in the relation. If an attribute is already a super key and other attributes get added to the set, then the set is still a super key.

Week 2 Notes:

Relational Algebra:

- We can perform queries on a set of relations to get information from them. A **query** is a request for data or information from a relation. The input is a relation and the output is a new relation.
- An algebra is a mathematical system consisting of the following:
    1. **Operands:** Variables or values from which new values can be constructed.
    2. **Operators:** Symbols denoting procedures that construct new values from given values.
- **Relational algebra** is a widely used procedural query language. It collects instances of relations as input and gives occurrences of relations as output. It uses various operations to perform this action. It is an algebra whose operands are relations or variables that represent relations. Operators are designed to do the most common things that we need to do with relations in a database.
- Relational algebra operations are performed recursively on a relation. The output of these operations is a new relation, which might be formed from one or more input relations. Relational algebra operations do not modify the input relation in any way.

SELECT (σ):

- The SELECT operation is used for selecting a subset of the tuples according to a given selection condition. It is denoted by **$\sigma_p(x)$**. It is used as an expression to choose tuples which meet the selection condition. The select operation selects tuples that satisfy a given predicate.
- Notation: **$\sigma_p(x)$**
  - σ is the selection predicate.
  - x is the name of the relation.
  - p is the propositional logic. It is a boolean formula of terms and connectives.
    These connectives are: **^(and), V(or), ~(not)**.
    The operators are: **<, >, ≤, ≥, =, ≠**
    These terms are: **attribute operator attribute** and **attribute operator constant**.
- E.g. Consider the below relation.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Instructor Relation

If we do $\sigma_{SALARY >= 85000}$(instructor), we get all the tuples with attribute salary at least 85000 from the instructor relation.
I.e. We would get this as the output:

ID	*name*	*dept_name*	*salary*
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
83821	Brandt	Comp. Sci.	92000

PROJECTION (π):

- The projection operation gets the specified attributes from a relation.
- Notation: **$\pi_{A1, A2, An}(r)$**
  - π denotes the project operation.

  - A1, A2, An are the attributes in the relation, r.
  - r is the name of the relation.
- E.g. Consider the relation below:

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Instructor Relation

If we do: $\pi_{ID,salary}$(instructor), it would get the attributes ID and salary from the instructor relation.
I.e. This would be the output:

ID	*salary*
10101	65000
12121	90000
15151	40000
22222	95000
32343	60000
33456	87000
45565	75000
58583	62000
76543	80000
76766	72000
83821	92000
98345	80000

NATURAL JOIN (⋈):

- Combines two relations into a single relation.
- Can only be performed if there is a common attribute between the relations. The name and domain of the attribute must be the same. Note that if there is an entry in only one relation, it will be omitted from the result relation.
- Also called inner join.
- Notation: **r ⋈ s**

- E.g. Consider the two relations below:

Instructor Relation

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

and

Department Relation

dept_name	building	budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Since they both have the attribute dept_name and since the domain of both dept_name is string, if we do instructor ⋈ department, we get the following output:

ID	name	salary	dept_name	building	budget
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
12121	Wu	90000	Finance	Painter	120000
15151	Mozart	40000	Music	Packard	80000
22222	Einstein	95000	Physics	Watson	70000
32343	El Said	60000	History	Painter	50000
33456	Gold	87000	Physics	Watson	70000
45565	Katz	75000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
76543	Singh	80000	Finance	Painter	120000
76766	Crick	72000	Biology	Watson	90000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000

Theta-Join($\bowtie_C$):

- Theta join combines tuples from different relations provided they satisfy the theta condition.
- Notation: **r $\bowtie_C$ s**
- R3 = R1 $\bowtie_C$ R2
  - Take the product R1 X R2.
  - Then apply σ_C to the result. As for σ, C can be any boolean-valued condition.

- E.g.

Left Outer Join ($\bowtie_L$):

- The left outer join operation allows keeping all tuple in the left relation. However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.
- Notation: **$R \bowtie_L S$**

Right Outer Join ($\bowtie_R$):

- The right outer join operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.
- Notation: **$A \bowtie_R B$**

Full Outer Join ($\bowtie_O$):

- In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.
- Notation: **$A \bowtie_O B$**

CARTESIAN PRODUCT (x):

- The cross product of 2 relations. The cross product produces all possible pairs of rows of the two relations.
- This is used to merge columns from two relations. Generally, a cartesian product is never a meaningful operation when it performs alone. However, it becomes meaningful when it is followed by other operations.
- Notation: **r x s**
- E.g. The cross product of {a, b} and {c, d} is {a,c}, {a,d}, {b,c} and {b,d}.

- E.g. Consider the 2 relations below:

A	B
α	1
β	2

r

and

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

If we do r x s, we get the following output:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

- A problem arises when the 2 relations share a same attribute name. How would we differentiate between the 2 attributes? We can rename the attributes of the relations.

RENAME (ρ):

- Notation: **$\rho_x(E)$**
  - E is the relation name.
  - x is what will be prepended to all the attribute names in relation E.
  - The rename operation renames all attributes in relation E by prepending them with x.
- E.g. Suppose the below table is the relation r.

A	B
a	1
b	2

If I do $P_r(r) \times P_s(r)$, we get the following relation:

r.A	r.B	s.A	s.B
a	1	a	1
a	1	b	2
b	2	a	1
b	2	b	2

UNION (U):

- Union operator when applied on two relations R1 and R2 will give a relation with tuples which are either in R1 or in R2. Furthermore, it eliminates all duplicate tuples.
  I.e. The tuples that are in both R1 and R2 will appear only once in the result relation.
- For a union operation to be valid, the following conditions must hold:
  1. The 2 relations must have the same **arity** (same number of attributes).
  2. The attribute domains must be compatible.
     I.e. The ith column of relation 1 must be of the same domain as the ith column of relation 2.
  3. Duplicate tuples are automatically eliminated.
- Notation: **r U s**
- E.g. Consider the 2 relations below:

A	B
α	1
α	2
β	1

r

and

A	B
α	2
β	3

s

If we do r U s, we will get the output:

A	B
α	1
α	2
β	1
β	3

- E.g. Consider the 2 relations below:

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

If we do A U B, we get the following output:

Table A U B	
column 1	column 2
1	1
1	2
1	3

DIFFERENCE (-):

- Returns a relation consisting of all the tuples which are present in the first relation but are not in the second relation.
- Notation: **r - s**

- E.g. Consider the 2 relations below:

A	B
α	1
α	2
β	1

r and

A	B
α	2
β	3

s

If we do r - s, we will get the following output:

A	B
α	1
β	1

- E.g. Consider the 2 relations below:

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

If we do r - s, we will get the following output:

Table A - B	
column 1	column 2
1	2

INTERSECTION (∩):

- Defines a relation consisting of a set of all tuples that are in both A and B, where A and B are 2 relations.
- Notation: **A ∩ B**

- E.g. Consider the 2 relations below:

A	B
α	1
α	2
β	1

r and

A	B
α	2
β	3

s

If we do r ∩ s, we will get the output:

A	B
α	2

- E.g. Consider the 2 relations below:

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

If we do A ∩ B, we will get the output:

Table A ∩ B	
column 1	column 2
1	1

- **Note:** r ∩ s = r–(r–s)

Summary of Relational Algebra Operations:

Operation	**Purpose**
Select(σ)	The select operation is used for selecting a subset of the tuples according to a given selection condition

Projection(π)	The projection eliminates all attributes of the input relation but those mentioned in the projection list.
Union Operation(∪)	It includes all tuples that are in tables A or in B.
Difference(-)	The result of A - B, is a relation which includes all tuples that are in A but not in B.
Intersection(∩)	Intersection defines a relation consisting of a set of all tuple that are in both A and B.
Cartesian Product(X)	Cartesian product merges columns from two relations.
Theta Join($\bowtie_C$)	The general case of JOIN operation is called a Theta join.
Natural Join(⋈)	Natural join can only be performed if there is a common attribute (column) between the relations. Same as inner join.
Left Outer Join($\bowtie_L$)	In left outer join, the operation allows keeping all tuple in the left relation.
Right Outer join($\bowtie_R$)	In right outer join, the operation allows keeping all tuple in the right relation.
Full Outer Join ($\bowtie_O$)	In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.
Rename(ρ)	Renames the attributes of the relation.

Week 3 & 4 Notes:

Intro to SQL:

- SQL, Structured Query Language, is a computer language for storing, manipulating and retrieving data stored in a relational database.
- SQL keywords are not case sensitive. E.g. select is the same as SELECT.
- 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 requires single quotes around strings.
- Below is a list of SQL commands/clauses listed in alphabetical order.

ALTER TABLE:

- The SQL ALTER TABLE command is used to add, delete or modify columns in an existing table. You should also use the ALTER TABLE command to add and drop various constraints on an existing table.
- There are many variations of the ALTER TABLE command:
    1. If we want to add a column, we use the command **ALTER TABLE table_name ADD column_name datatype;**
    2. If we want to remove a column, we use the command **ALTER TABLE table_name DROP COLUMN column_name;**

3. If we want to change the data type of a column, we use the command **ALTER TABLE table_name MODIFY COLUMN column_name datatype;**
4. If we want to add a not null constraint to a column, we use the command **ALTER TABLE table_name MODIFYcolumn_name datatype NOT NULL;**
5. If we want to add a primary key constraint to a column(s), we use the command **ALTER TABLE table_name ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);**

AS:

- The AS command is used to rename a column or table with an alias. An alias only exists for the duration of the query.
- Syntax: **select col as alias from table;**

AVG:

- The AVG() function returns the average value of a numeric column.
- Syntax: **SELECT AVG(column_name) FROM table_name WHERE condition;**

CASE:

- The CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause. If there is no ELSE part and no conditions are true, it returns NULL.
- Syntax:

```
CASE
        WHEN condition1 THEN result1
        WHEN condition2 THEN result2
        WHEN conditionN THEN resultN
        ELSE result
END;
```

COUNT:

- The COUNT() function returns the number of rows that matches a specified criteria.
- Syntax: **SELECT COUNT(column_name) FROM table_name WHERE condition;**

CREATE DATABASE:

- The CREATE DATABASE statement is used to create a new SQL database.
- Syntax: **CREATE DATABASE databasename;**

CREATE TABLE:

- The CREATE TABLE operator is used to create a new table.
- Syntax: **CREATE TABLE table_name (column1 datatype [arg1], column2 datatype [arg2], …, column(n) datatype [argn], integrity-constraint1, …, integrity-constraint(k));**
- The datatypes are the following:
    1. char(n): A fixed-length character string.
    2. varchar(n): A variable-length character string with max length n.
    3. int: An integer.
    4. numeric(p, d): A fixed point number with p digits of which d of the digits are to the right of the decimal point.
    5. real/double precision: Floating point and double precision floating point.
    6. float(n): A floating point with at least n digits of precision.

- The integrity constraints are the following:
    1. NOT NULL:
        - By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such a constraint on this column specifying that NULL is now not allowed for that column. A NULL is not the same as no data, rather, it represents unknown data.
        - Specifies that this attribute may not have the null value. We list this constraint as an argument when defining the type of the attribute.
    2. PRIMARY KEY:
        - A primary key is a field in a table which uniquely identifies each row/record in a database table.
        - These attributes form the primary keys for the relation. Primary keys must be non-null and unique.
        - A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key.
    3. FOREIGN KEY:
        - A foreign key is a key used to link two tables together. This is sometimes also called a referencing key.
        - A Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.
        - The values of these attributes for any tuple in the relation must correspond to values of the primary key attributes of some other tuple.
- The arguments are optional and are the following:
    1. AUTOINCREMENT:
        - Autoincrement allows a unique number to be generated automatically when a new record is inserted into a table.
    2. NOT NULL:
        - By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such a constraint on this column specifying that NULL is now not allowed for that column. A NULL is not the same as no data, rather, it represents unknown data.
        - Specifies that this attribute may not have the null value. We list this constraint as an argument when defining the type of the attribute.
    3. PRIMARY KEY:
        - You can declare a primary key by putting the words “PRIMARY KEY” beside the attribute name.
- E.g.

```
1. CREATE TABLE CUSTOMERS
    (ID   INT                    NOT NULL,
     NAME VARCHAR (20)     NOT NULL,
     AGE  INT                    NOT NULL,
     ADDRESS  CHAR (25) ,
     SALARY   DECIMAL (18, 2),
     PRIMARY KEY (ID));
2. CREATE TABLE department2
        (dept_name   VARCHAR(20),
```

```
        building        VARCHAR(15),
        budget          NUMERIC(12,2),
        PRIMARY KEY (dept_name));
```

3.
```
CREATE TABLE course
        (course_id      VARCHAR(7),
        title           VARCHAR(50),
        dept_name       VARCHAR(20),
        credit          NUMERIC(2,0),
        PRIMARY KEY (course_id),
        FOREIGN KEY (dept_name) REFERENCES department);
```

4.
```
CREATE TABLE course
        (course_id      VARCHAR(7), AUTOINCREMENT PRIMARY KEY
        title           VARCHAR(50),
        dept_name       VARCHAR(20),
        credit          NUMERIC(2,0),
        FOREIGN KEY (dept_name) REFERENCES department);
```

CROSS JOIN:

- The CROSS JOIN operator joins every row from the first table with every row from the second table. I.e. The cross join returns a Cartesian product of rows from both tables.
- Syntax: **SELECT select_list FROM table1 CROSS JOIN table2;**

DELETE:

- The DELETE operator is used to delete existing records/tuples in a table.
- **Note:** The DELETE operator does not delete the table.
- There are 2 possible syntaxes for the delete operator:
    1. **DELETE FROM table_name;**
    2. **DELETE FROM table_name WHERE condition;**
- The first way deletes all tuples from the table where as the second way deletes the tuples that satisfy the condition.

DROP DATABASE:

- The DROP DATABASE statement is used to drop (delete) an existing SQL database.
- Syntax: **DROP DATABASE databasename;**

DROP TABLE:

- The DROP TABLE operator is used to delete an existing table.
- Syntax: **DROP TABLE table;**

EXCEPT:

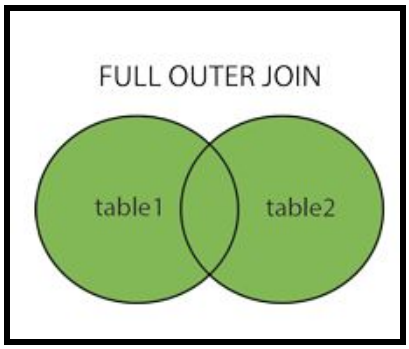
- The EXCEPT operator compares the result sets of two queries and returns the distinct rows from the first query that are not output by the second query. In other words, the EXCEPT subtracts the result set of a query from another.
- Syntax: **query1 EXCEPT query2;**

EXIST:

- The EXISTS operator is used to test for the existence of any record in a subquery.
- The EXISTS operator returns true if the subquery returns one or more records.
- Syntax: **SELECT column_name(s) FROM table_name WHERE EXISTS (SELECT column_name FROM table_name WHERE condition);**

FULL OUTER JOIN:

- The FULL OUTER JOIN keyword returns all records when there is a match in table1's or table2's table records.



- Syntax: **SELECT select_list FROM table1 FULL OUTER JOIN table2 ON join_predicate;**

GROUP BY:

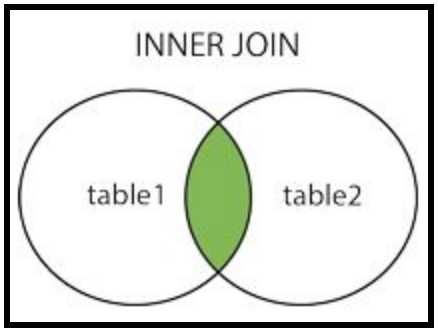
- The GROUP BY statement groups rows that have the same values into summary rows.
- The GROUP BY statement is often used with aggregate functions to group the result-set by one or more columns.
- Syntax: **SELECT column_name(s) FROM table_name WHERE condition GROUP BY column_name(s);**

HAVING:

- A HAVING clause in SQL specifies that an SQL SELECT statement should only return rows where aggregate values meet the specified conditions. It was added to the SQL language because the WHERE keyword could not be used with **aggregate functions**, which is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning. Examples of aggregate functions are avg, count, and sum.
- Syntax: **SELECT column_name(s) FROM table_name WHERE condition GROUP BY column_name(s) HAVING condition**

INNER JOIN:

- The INNER JOIN keyword selects records that have matching values in both tables.



- Syntax: **select select_list from table1 inner join table2 on join_predicate;**

INSERT INTO:

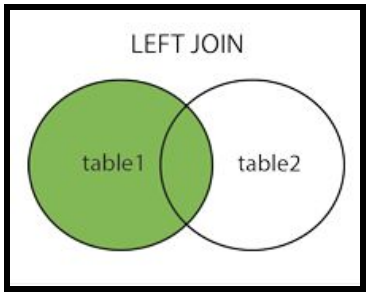
- The INSERT INTO operator is used to insert new records in a table.
- Syntax: **insert into table_name (col1, col2, …, coln) values (value1, value2, …, value(n));**
- **Note:** All primary key values will be updated automatically.

INTERSECT:

- The INTERSECT operator combines result sets of two or more queries and returns distinct rows that are output by both queries.
- Syntax: **query1 INTERSECT query2;**

LEFT JOIN:

- The LEFT JOIN keyword returns all records from table1, and the matched records from table2. The result is NULL from table2, if there is no match.
- Syntax: **SELECT select_list FROM table1 LEFT JOIN table2 ON join_predicate;**



LIMIT:

- The limit clause is used to specify the number of records to return.
- Syntax: **SELECT column_name(s) FROM table_name WHERE condition LIMIT number;**

MIN/MAX:

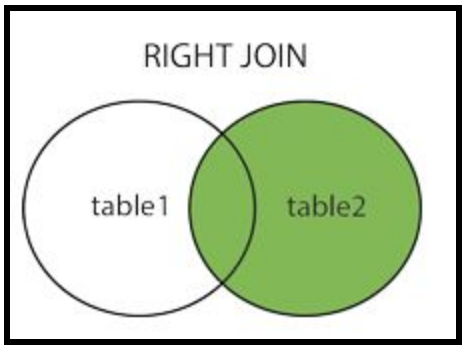
- The MIN() function returns the smallest value of the selected column.
- The MAX() function returns the largest value of the selected column.
- Syntax: **SELECT MIN|MAX(column_name)FROM table_name WHERE condition;**

ORDER BY:

- The ORDER BY keyword is used to sort the result-set in ascending or descending order.
- The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword. To sort the records in ascending order, use the ASC keyword.
- Syntax: **SELECT column1, column2, …, column(n) FROM table_name ORDER BY column1, column2, ... ASC|DESC;**
- We can also order by several columns.
- E.g. 1 The command **SELECT * FROM Customers ORDER BY Country, CustomerName;** will select all customers from the "Customers" table, sorted by the "Country" and "CustomerName" columns. This means that it orders by Country, but if some rows have the same Country, it orders them by CustomerName.
- E.g. 2 The command **SELECT * FROM Customers ORDER BY Country ASC, CustomerName DESC;** will select all customers from the "Customers" table, sort the column “Country” ascending and sort the column “CustomerName” descending.

RIGHT JOIN:

- The RIGHT JOIN keyword returns all records from table2, and the matched records from table1. The result is NULL from table1, when there is no match.



- Syntax: **SELECT select_list FROM table1 RIGHT JOIN table2 ON join_predicate;**

SELECT:

- The SELECT operator is used to select data from a database. The data returned is stored in a result table, called the **result-set**.
- Syntax: **SELECT select_list FROM table_name;**
- **Note:** If you want to select all columns, you can use the following syntax: **select * from table_name;**

SELECT DISTINCT:

- The SELECT DISTINCT operator is used to return only distinct (different) values.
- Syntax: **SELECT DISTINCT select_list FROM table_name;**

SELF JOIN:

- A self join allows you to join a table to itself.
- Syntax: **SELECT select_list FROM table1 T1, table1 T2 WHERE condition;**

SUM:

- The SUM() function returns the total sum of a numeric column.
- Syntax: **SELECT SUM(column_name) FROM table_name WHERE condition;**

UNION:

- The UNION operator is used to combine the result-set of two or more SELECT statements.
    - Each SELECT statement within UNION must have the same number of columns.
    - The columns must also have similar data types.
    - The columns in each SELECT statement must also be in the same order.
- Syntax: **query1 UNION query2;**
- The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL.
- Syntax: **query1 UNION ALL query2;**

UPDATE:

- The UPDATE operator is used to modify the existing records in a table.
- The general syntax for update is: **UPDATE table_name SET column1 = value1, column2 = value2, …, column(n) = value(n);**

VIEW:

- In SQL, a view is a virtual table based on the result-set of an SQL statement. A view is a result set of a stored query. Think of it as a subset of a table or a snapshot into a table.

- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
- You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.
- Views have several advantages in the real world:
    - A view can hide certain columns from a table. This is useful if you want users to see certain columns but not others.
    - Can provide time savings in writing queries by having a group of frequently accessed tables joined together in a view.
    - Provide more ways to manipulate data and easily get the information you are looking for.
- Syntax: **CREATE VIEW view_name AS SELECT column1, column2, …, column(n) FROM table_name WHERE condition;**

WHERE:

- The WHERE clause is used to extract only those records that fulfill a specified condition.
- Syntax: **select column1, column2, …, column(n) from table_name where condition;**

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<> or !=	Not equal.
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in a SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list as per the condition.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. The syntax of the between statement is **SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2;**
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criteria.

IN	The IN operator is used to compare a value to a list of literal values that have been specified. The IN operator is a shorthand for multiple OR conditions.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators. There are 2 wildcards used with LIKE. % represents 0, 1 or multiple characters. _ represents a single character. The syntax of like is **SELECT column FROM table_name WHERE column LIKE pattern;**
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. This is a negate operator.
OR	The OR operator is used to combine multiple conditions in a SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness.

Null Value:

- Every type can have the special value null.
- The NULL term is used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.
- A field with a NULL value is a field with no value.
  Note: A NULL value is different than a zero value or a field that contains spaces.
- You can use the IS NULL or IS NOT NULL operators to check for a NULL value.
- If we don't want null values, we can add a constraint.

Week 5 Notes:

HTML:

- HTML stands for HyperText Markup Language. It is a formatting system used by browsers to render the webpages we see on the internet. Using HTML, you can create a Web page with text, graphics, sound, and video
- It creates the structure or the skeleton of a webpage.
- HTML code is made up of elements called **tags** that denote the structure of a webpage. A tag is a keyword enclosed by angle brackets. There are opening and closing tags for many but not all tags. The affected text is between the two tags. The opening and closing tags use the same command except the closing tag contains an additional forward slash /.
- Whenever you have HTML tags within other HTML tags, you must close the nearest tag first.
- A comment in HTML is denoted by **<!-- -->**.
- A basic HTML web page looks as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
```

```
    <p>This is a paragraph</p>
  </body>
</html>
```

Here is a basic description of the above tags:

- **<!DOCTYPE html>:** This is here for legacy reasons. It mostly does nothing but when omitted some old browsers use a rendering mode that is incompatible with some specifications. Although it appears useless, it is necessary for any HTML file you produce.
- **<html>:** Tells the browser this is an HTML document. It's the container for all the other HTML elements, except the <!DOCTYPE> tag.
- **<head>:** A container for all the head elements (i.e. scripts, styles, meta information, etc).
- **<title>:** The name of the website. This will be shown in the "tab" on your browser
- **<body>:** The meat and potatoes of the website. This is where you will put all the "visual" elements.
- ** :** A paragraph tag that indicates that you are writing a paragraph.

- Below is a list of the tags and other useful stuff in alphabetical order.
  1. **Anchor:**
     a. Denoted with the **<a>** tag.
     b. The <a> tag defines a hyperlink, which is used to link from one page to another.
     c. The most important attribute of the <a> element is the href attribute, which indicates the link's destination.
     d. General Syntax: **...**
     e. E.g. **this is a link**

  2. **Bold:**
     a. Denoted with the **** tag.
     b. General syntax: ** … **
     c. E.g. ** This is normal text. This is bold text. **
  3. **Body:**
     a. Denoted with the **<body>** tag.
     b. The <body> tag defines the document's body.
     c. The <body> element contains all the contents of an HTML document, such as text, hyperlinks, images, tables, lists, etc.
     d. General syntax:
        <body>
        …
        </body>
     e. E.g.
        <body>
        The content of the document.
        </body>
  4. **Comments:**
     a. Denoted by **<!-- -->**

   b. E.g. **<!--This is a comment. Comments are not displayed in the browser-->**

5. **Doctype:**
   a. Denoted by **<!DOCTYPE html>**
   b. The <!DOCTYPE> declaration must be the very first thing in your HTML document, before the <html> tag.
   c. This is here for legacy reasons. It mostly does nothing but when omitted some old browsers use a rendering mode that is incompatible with some specifications. Although it appears useless, it is necessary for any HTML file you produce.
6. **Head:**
   a. Denoted with the **<head>** tag.
   b. The <head> element is a container for all the head elements.
   c. The <head> element can include a title for the document, scripts, styles, meta information, and more.
   d. The following elements can go inside the <head> element <title>, <style>, <base>, <link>, <meta>, <script>, <noscript>.
   e. General syntax:
   <head>
   …
   </head>
   f. E.g.
   <head>
   <title>Title of the document</title>
   </head>
7. **Headings:**
   a. Denoted with the **<h1>** to **<h6>** tags.
   b. **<h1>** defines the most important heading. **<h6>** defines the least important heading.
   c. It is generally used for titles. Search engines use the headings to index the structure and content of your web pages.
   d. General syntax: **<hi> … </hi>** where i is in 1, 2, 3, 4, 5, 6.
   e. E.g.
   <h1>Heading 1</h1>
   <h2>Heading 2</h2>
   <h3>Heading 3</h3>
   <h4>Heading 4</h4>
   <h5>Heading 5</h5>
   <h6>Heading 6</h6>
8. **HTML:**
   a. Denoted with the **<html>** tag.
   b. The <html> tag tells the browser that this is an HTML document.
   c. The <html> tag represents the root of an HTML document.
   d. The <html> tag is the container for all other HTML elements except for the <!DOCTYPE> tag.

e. General syntax:
<html>
…
</html>

f. E.g.
```
<!DOCTYPE HTML>
<html>
<head>
<title>Title of the document</title>
</head>

<body>
The content of the document.
</body>

</html>
```

9. **Images:**
   a. Denoted with the **<img>** tag.
   b. The **<img>** tag is empty, it contains attributes only, and does not have a closing tag.
   c. General syntax: **<img src = “url” alt = “Description” …>**
   d. The src attribute specifies the URL (web address) of the image.
   e. The alt attribute provides an alternate text for an image, if the user for some reason cannot view it (because of slow connection, an error in the src attribute, or if the user uses a screen reader). The value of the alt attribute should describe the image.
   f. E.g. **<img src="img_chania.jpg" alt="Flowers in Chania">**
10. **Italics:**
   a. Denoted with the **** tag.
   b. General syntax: ** … </>**
   c. E.g. ** This is normal text. This is italic text. **

11. **Lists:**
   a. Denoted with the **<li>** tag.
   b. There are 2 types of lists, ordered and unordered:
      i. Ordered Lists:
         - Denoted with the **<ol>** tag.
         - An ordered list can be numerical or alphabetical.
         - We use <li> to define the list items.
         - General syntax:
```
<ol>
  <li>...</li>
  <li>...</li>
  ...
</ol>
```
         - E.g.
```
<ol>
```

```
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ol>
```

ii. Unordered Lists:
- Denoted with the tag **<ul>**.
- An unordered list will be displayed with bullets.
- We use <li> to define the list items.
- General syntax:

```
<ul>
  <li>...</li>
  <li>...</li>
  ...
</ul>
```

- E.g.

```
<ul>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ul>
```

12. **Paragraph:**
    a. Denoted with the ** ** tag.
    b. A paragraph tag that indicates that you are writing a paragraph.
    c. General syntax: ** … **
    d. E.g. ** This is some text in a paragraph. **
13. **Title:**
    a. Denoted with the **<title>** tag.
    b. The <title> tag is required in all HTML documents and it defines the title of the document.
    c. The <title> element defines a title in the browser toolbar, provides a title for the page when it is added to favorites and displays a title for the page in search-engine results.
    d. General syntax: **<title> … </title>**
    e. E.g. **<title>HTML Reference</title>**
14. **Table:**
    a. Denoted with the **<table>** tag.
    b. The <tr> element defines a table row. A <tr> element contains one or more <th> or <td> elements.
    c. The <th> element defines a table header. The text in <th> elements are bold and centered by default. The <th> element creates header cells which contain header information.
    d. The <td> element defines a table cell. The text in <td> elements are regular and left-aligned by default. The <td> element creates standard cells which contain data.
    e. E.g. A simple HTML table with two header cells and two data cells.

```
<table>
 <tr>
```

```
  <th>Month</th>
  <th>Savings</th>
 </tr>
 <tr>
  <td>January</td>
  <td>$100</td>
 </tr>
</table>
```

- Some tags contain information inside the **leading tag** (the first tag) called **attributes**. Examples of attributes are <img> and <a>. Some tags don't have any attributes at all and some have a range of varying optional attributes.

JavaScript:

- JavaScript is the Programming Language for the Web.
- JavaScript can update and change both HTML and CSS.
- JavaScript can calculate, manipulate and validate data.

CSS:

- CSS stands for Cascading Style Sheets.
- CSS describes how HTML elements are to be displayed.
  I.e. It deals with the layout/design of the webpage.

Flask Introduction:

- Flask is a web application framework written in Python. A web application framework or web framework represents a collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management etc.
- A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated python virtual environments for them.

Flask Application:

- Here is a basic program with Flask:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
  return 'Hello World'

if __name__ == '__main__':
  app.run()
```

The flask constructor takes the name of the current module (__name__) as argument. The route() function of the Flask class is a decorator, which tells the application which URL should call the associated function. The general syntax for the route() function is: **app.route(rule, options)**. The rule parameter represents URL binding with the function.

The options is a list of parameters to be forwarded to the underlying Rule object.
In the above example, '/' URL is bound with the hello_world() function. Hence, when the home page of the web server is opened in a browser, the output of this function will be rendered.
Finally the run() method of Flask class runs the application on the local development server. The general syntax for the run() function is **app.run(host, port, debug, options)** but all parameters are optional. The host is the hostname to listen on. The default for host is 127.0.0.1 (localhost). Set the host to '0.0.0.0' to have the server available externally. The default for port is 5000. The default for debug is false. If it is set to true, it provides debug information. The options are to be forwarded to the underlying Werkzeug server.

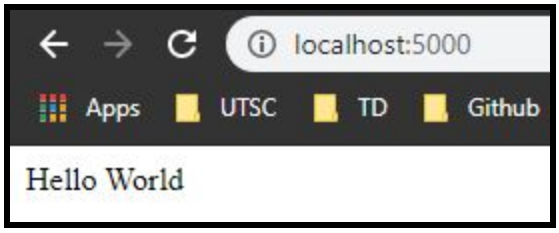
- A Flask application is started by calling the run() method. However, while the application is under development, it should be restarted manually for each change in the code. To avoid this inconvenience, enable debug support. The server will then reload itself if the code changes. It will also provide a useful debugger to track the errors if any, in the application.
- The debug mode is enabled by setting the debug property of the application object to true before running or passing the debug parameter to the run() method.
  E.g.
  app.debug = True
  app.run()
  app.run(debug = True)
- E.g. Suppose I am running this piece of code.

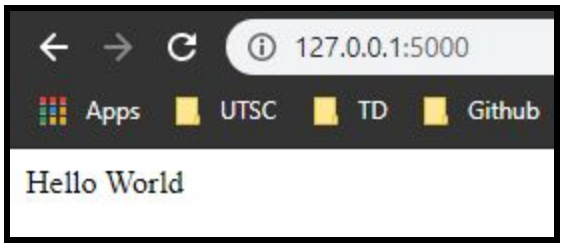
```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World'

if (__name__ == "__main__"):
    app.run()
```

If I run it on terminal, I'll see this

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ date
Tue Feb  4 12:59:52 STD 2020
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ python3 Flask_Test_1.py
 * Serving Flask app "Flask_Test_1" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

If I type in http://127.0.0.1:5000/ or http://localhost:5000/, I'll see this:





If I want to make any changes, I have to rerun my program.
Suppose I want to run this code:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<h1> Hello World </h1>'

if (__name__ == "__main__"):
    app.run()
```

I need to rerun my program, as shown here:

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ date
Tue Feb  4 13:03:18 STD 2020
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ python3 Flask_Test_1.py
 * Serving Flask app "Flask_Test_1" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [04/Feb/2020 13:03:28] "GET / HTTP/1.1" 200 -
```

However, if I put the debug statements in my code like such:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<h1> Hello World </h1>'

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

and I rerun my program

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ date
Tue Feb  4 13:04:36 STD 2020
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ python3 Flask_Test_1.py
 * Serving Flask app "Flask_Test_1" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 333-205-022
```

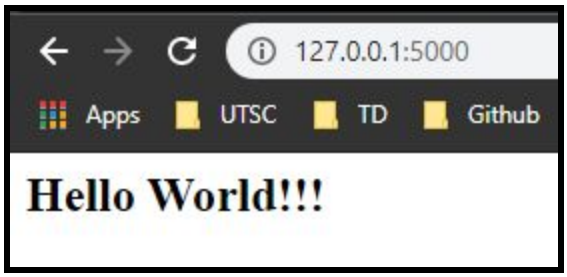
If I make any changes to my code, I don't need to rerun my program.
Right now, my website looks like this

- Suppose I modify my code:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<h2> Hello World!!! </h2>'

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

Without rerunning my program, my website changed.

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ date
Tue Feb  4 13:04:36 STD 2020
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ python3 Flask_Test_1.py
 * Serving Flask app "Flask_Test_1" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 333-205-022
 * Detected change in '/mnt/c/Users/rick/Desktop/Flask_Test_1.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 333-205-022
127.0.0.1 - - [04/Feb/2020 13:06:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [04/Feb/2020 13:06:37] "GET / HTTP/1.1" 200 -
```



Flask Routing:

- Modern web frameworks use the routing technique to help a user remember application URLs. It is useful to access the desired page directly without having to navigate from the home page. The route() decorator in Flask is used to bind URL to a function.
- Suppose I want to create an about page. My code would look like this:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<h2> Hello World!!! </h2>'

@app.route('/about')
def about():
    return '<h1> About Page </h1>'

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

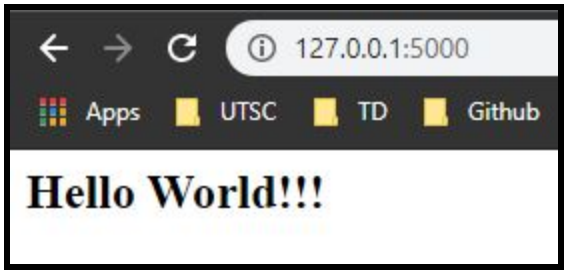
- If I type in http://localhost:5000/about in my browser, I see this:

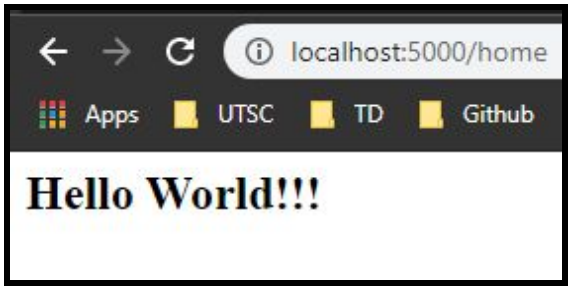
- If you want multiple routes handled by the same function, you add the multiple decorators before the function.

- E.g. Suppose I want to have / and /home route to the same page. My code looks like this:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
@app.route('/home')
def hello_world():
    return <h2> Hello World!!! </h2>

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

Furthermore, http://localhost:5000/ and http://localhost:5000/home routes to the same page.





Flask Templates:

- While it is possible to return the output of a function bound to a certain URL in the form of HTML, generating HTML content from Python code is cumbersome, especially when variable data and Python language elements like conditionals or loops need to be put. This would require frequent escaping from HTML. Instead of returning hardcode HTML from the function, a HTML file can be rendered by the render_template() function.
- To use templates, you'll need to create a template directory in the same directory where your python code is.

Flask will try to find the HTML file in the templates folder, in the same folder in which this script is present.

- Application folder
  - Hello.py
  - templates
    - hello.html

- You'll also need to import render_template from Flask.

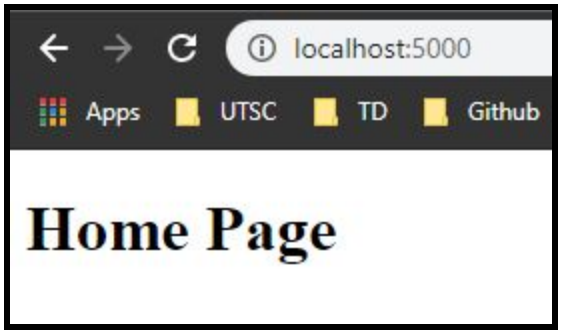
- E.g.
  I have this html file called home.html.

```
Flask1.py | home.html

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title></title>
5  </head>
6  <body>
7      <h1> Home Page </h1>
8  </body>
9  </html>
```

In my python program, I imported render_template from Flask and used it like such:

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
@app.route('/home')
def hello_world():
    return render_template('home.html')

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

The layout of my website didn't change.



Flask Variables:

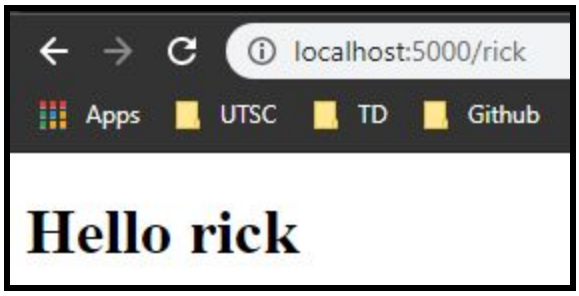
- It is possible to build a URL dynamically, by adding variable parts to the rule parameter. This variable part is marked as <variable-name>. It is passed as a keyword argument to the function with which the rule is associated.
- In addition to the default string variable part, rules can be constructed using the following int, float or path.
- The term 'web templating system' refers to designing an HTML script in which the variable data can be inserted dynamically. A web template system comprises of a template engine, some kind of data source and a template processor.

- Flask uses jinja2 template engine. A template language is simply HTML with variables and other programming constructs like conditional statement and for loops etc. This allows you to so some logic in the template itself. The template language is then rendered into plain HTML before being sent to the client. A web template contains HTML syntax interspersed placeholders for variables and expressions which are replaced values when the template is rendered.
- The jinja2 template engine uses the following delimiters for escaping from HTML.
    - {% ... %} for If statements and for loops
    - {{ ... }} for Expressions to print to the template output
    - {# ... #} for Comments not included in the template output
    - # ... ## for Line Statements
- E.g.

```python
from flask import Flask, render_template
app = Flask(__name__)

'''
This program shows how to pass information to an html file
'''

@app.route('/')
def hello_world():
    return render_template('home.html')

@app.route('/<name>')
def name(name):
    return render_template('hello.html', name=name)

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

```html
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h1> Hello {{name}} </h1>
</body>
</html>
```

The webpage would look like this:



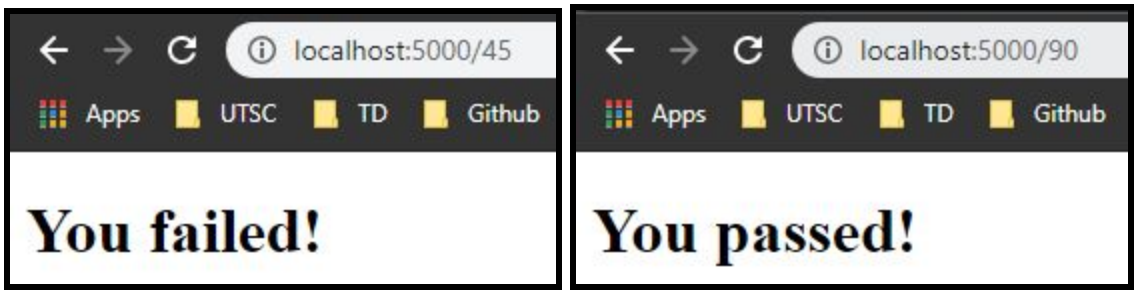
Flask If Statements:

- In Flask, if-else and endif are enclosed in delimiter {%..%}.
- You need to end an if-else statement with endif.
- E.g. Suppose we have these 2 pieces of code.

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
@app.route('/home')
def hello_world():
    return render_template('home.html')

@app.route('/<int:mark>')
def mark(mark):
    return render_template('mark.html', mark=mark)

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

```
1   <!DOCTYPE html>
2   <html>
3   <head>
4       <title></title>
5   </head>
6   <body>
7       {% if mark >= 50%}
8           <h1> You passed! </h1>
9       {%else%}
10          <h1> You failed! </h1>
11      {%endif%}
12  </body>
13  </html>
```

The webpage looks like this:



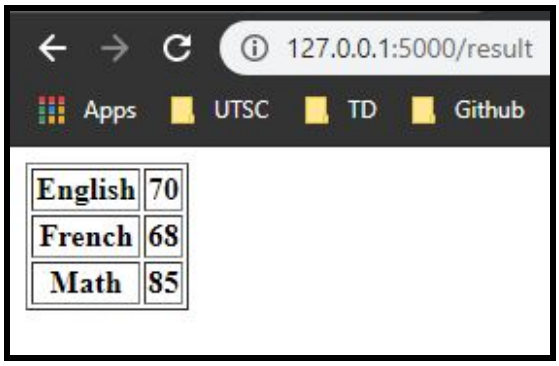
Flask For Loops:

- For loops are enclosed in {%...%}.
- Expressions are enclosed in {{...}}.
- E.g. Suppose I have these 2 pieces of code:

```python
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
@app.route('/home')
def hello_world():
    return render_template('home.html')

@app.route('/result')
def result():
    result_dict = {"Math": 85, "English": 70, "French": 68}
    return render_template('result.html', result = result_dict)

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

```html
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <table border = 1>
        {% for key, value in result.items() %}
        <tr>
            <th> {{key}} </th>
            <th> {{value}} </th>
        </tr>
        {% endfor %}
    </table>
</body>
</html>
```

The webpage looks like this:



Template Inheritance:

- Template inheritance is when a child template can inherit or extend a base template. This will allow you to define your template and a clear and concise way and avoid complexity.
- Imagine, you have an application with multiple pages and each page has the same header. If you want to change something in the header you would need to go through all the templates and change the header for each one which can be a tedious task. Thanks to template inheritance, you only need to create a base template that contains the common header code once and then make all the other templates extend the base template.
- You can use the {% extends %} and {% block %} tags to work with template inheritance. The {% extends %} tag is used to specify the parent template that you want to extend from your current template and the {% block %} tag is used to define and override blocks in the base and child templates. You need to end a {% block %} tag with the {% endblock %} tag. You don't need to put the name of the block in the endblock tag, but it's good practice to, especially if you have multiple blocks.
- E.g. Suppose I have these code snippets:

Template_Inheritance.py | BaseTemplate.html | home.html | about.html

```python
from flask import Flask, render_template
app = Flask(__name__)

'''
This program shows how to do template inheritance.
Template inheritance is when a child template can inherit or extend a base template.
This will allow you to define your template and a clear and concise way and avoid complexity.
'''

@app.route('/')
def hello_world():
    return render_template('home.html')

@app.route('/about')
def about():
    return render_template('about.html')

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

Template_Inheritance.py | home.html | about.html | BaseTemplate.html

```
<!-- The extends tag is used to specify the parent
     template that you want to extend from your current template.-->
{% extends "BaseTemplate.html" %}

<!-- The block tag is used to override blocks child templates.-->
{% block content %}
    <body>
        <h1> Home Page </h1>
    </body>
<!-- You don't need to put the name of the block in the endblock tag,
     but it's good practice to, especially if you have multiple blocks.-->
{% endblock content%}
```

Template_Inheritance.py | home.html | about.html | BaseTemplate.html

```
<!-- The extends tag is used to specify the parent
     template that you want to extend from your current template.-->
{% extends "BaseTemplate.html" %}

<!-- The block tag is used to override blocks child templates-->
{% block content %}
    <body>
        <h1> About Page </h1>
    </body>
<!-- You don't need to put the name of the block in the endblock tag,
     but it's good practice to, especially if you have multiple blocks.-->
{% endblock content%}
```

Template_Inheritance.py | home.html | about.html | BaseTemplate.html

```
<!DOCTYPE html>
<html>
<head>
    <title> Title </title>
    <h1> This will be in both home.html and about.html. </h1>
</head>
<body>
    <!-- The block tag is used to define blocks in the base template.
    The block and endblock tags must be positioned this way.-->
    {% block content %}{% endblock %}
</body>
</html>
```

The webpages look like this: