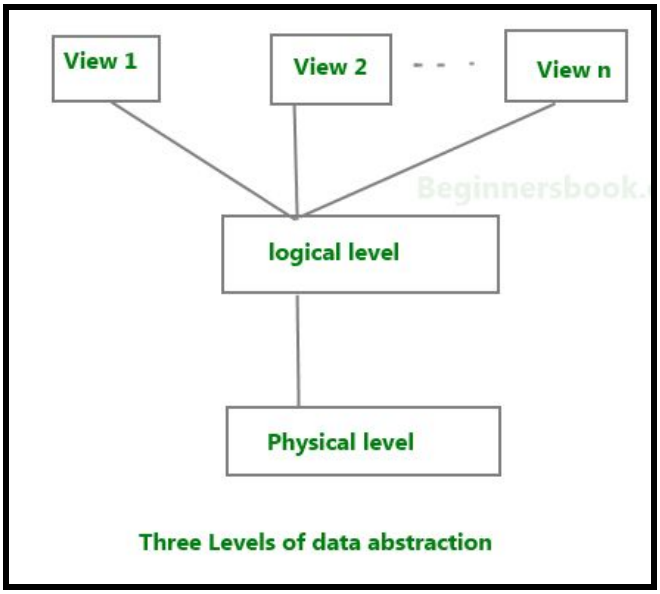
Lecture Notes:

- **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** 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(a1, a2, a3)
  B(b1, b2, 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.

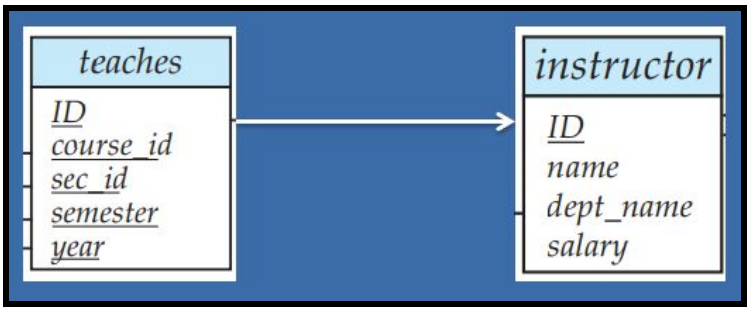
- **E.g.** Given the below relation schemas, find the foreign key(s). The primary key(s) are all underlined.
  instructor(ID,name,dept_name,salary)
  department(dept_name,building,budget)
  teaches(ID,course_id,sec_id,semester,year)

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.

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 ∪ 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 tuple 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:

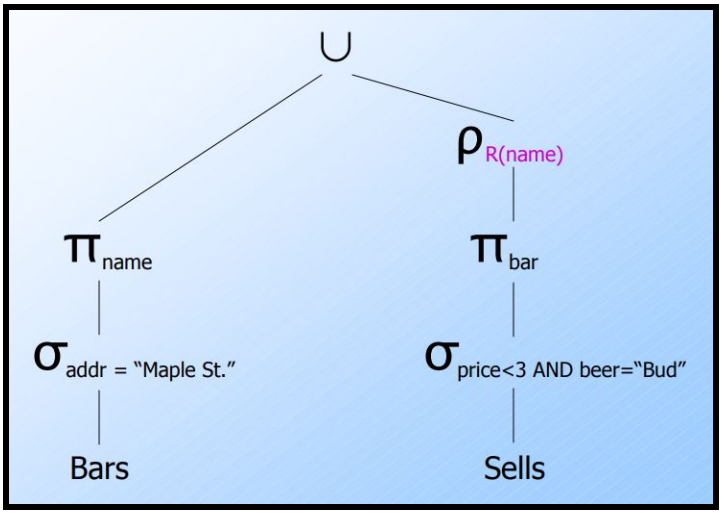
Table A ∩ B	
column 1	column 2
1	1

- **Note:** r ∩ s = r–(r–s)

Building Complex Expressions:

- Combine operators with parentheses and precedence rules.
- Three notations:
  1. **Sequences of assignment statements:**
     a. Create temporary relation names.
     b. Renaming can be implied by giving relations a list of attributes.
     c. E.g. R3 = R1 $\bowtie_C$ R2 can be written as:
        R4 = R1 X R2
        R3 = σ_C (R4)
  2. **Expressions with several operators:**
     a. E.g. R3 = R1 $\bowtie_C$ R2 can be written as R3 = σ_C (R1 X R2).
     b. Precedence of relational operators:
        1. [σ, π, ρ] (Highest)
        2. [X, $\bowtie$]
        3. ∩
        4. [∪ , —] (Lowest)
  3. **Expression trees:**
     - Leaves are operands. Variables stand for relations.
     - Interior nodes are operators, applied to their child or children.
     - E.g. Using the relations Bars(name, addr) and Sells(bar, beer, price), find the names of all the bars that are either on Maple St. or sell Bud for less than $3.

       Solution:



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($\bowtie$)	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.

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...);**

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:

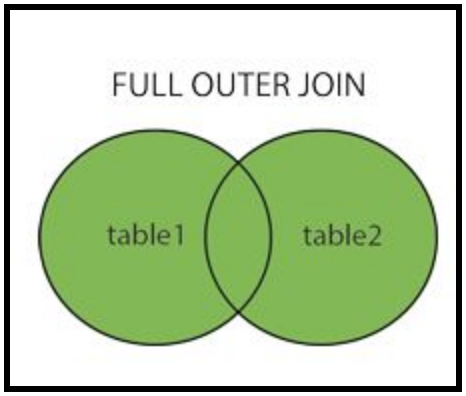
- The EXCEPT operator compares the result sets of two queries and returns the distinct rows from the first query that are not output by the second query. In other words, the EXCEPT subtracts the result set of a query from another.
- Syntax: **query1 EXCEPT query2;**

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:

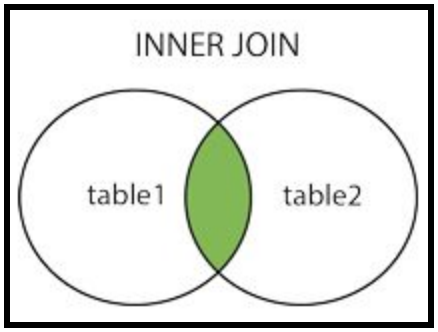
- The GROUP BY statement groups rows that have the same values into summary rows.
- The GROUP BY statement is often used with aggregate functions to group the result-set by one or more columns.
- Syntax: **SELECT column_name(s) FROM table_name WHERE condition GROUP BY column_name(s);**

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:

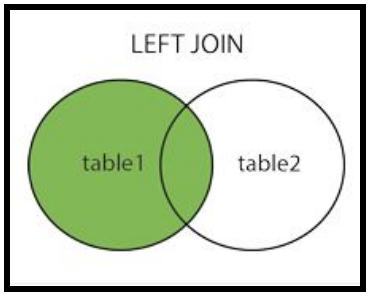
- The INSERT INTO operator is used to insert new records in a table.
- Syntax: **insert into table_name (col1, col2, …, coln) values (value1, value2, …, value(n));**
- **Note:** All primary key values will be updated automatically.

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:

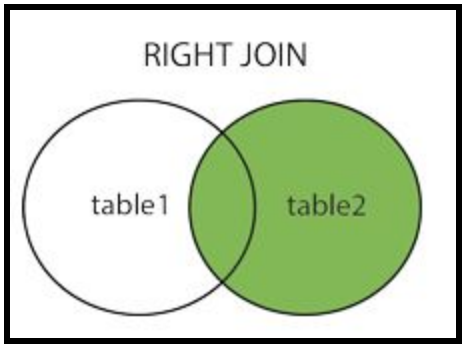
- The MIN() function returns the smallest value of the selected column.
- The MAX() function returns the largest value of the selected column.
- Syntax: **SELECT MIN|MAX(column_name)FROM table_name WHERE condition;**

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.
- There are 3 possible syntaxes for update:
    1. **UPDATE table_name SET column1 = value1, column2 = value2, …, column(n) = value(n)**
    2. **UPDATE table_name SET column1 = value1, column2 = value2, …, column(n) = value(n) WHERE condition;**
    3. **UPDATE table_name SET column = CASE**
       WHEN predicate1 THEN result1
       WHEN predicate2 THEN result2 ...
       WHEN predicate(n) THEN result(n)
       ELSE result0
       END;

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;**
- List of Operators for the WHERE clause:

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.

- Syntax for AND: **select column1, column2, …, column(n) from table_name where condition1 and condition2 and condition3 ...;**
- Syntax for OR: **select column1, column2, …, column(n) from table_name where condition1 or condition2 or condition3 ...;**
- Syntax for NOT: **select column1, column2, …, column(n) from table_name where not condition;**

Null Value:

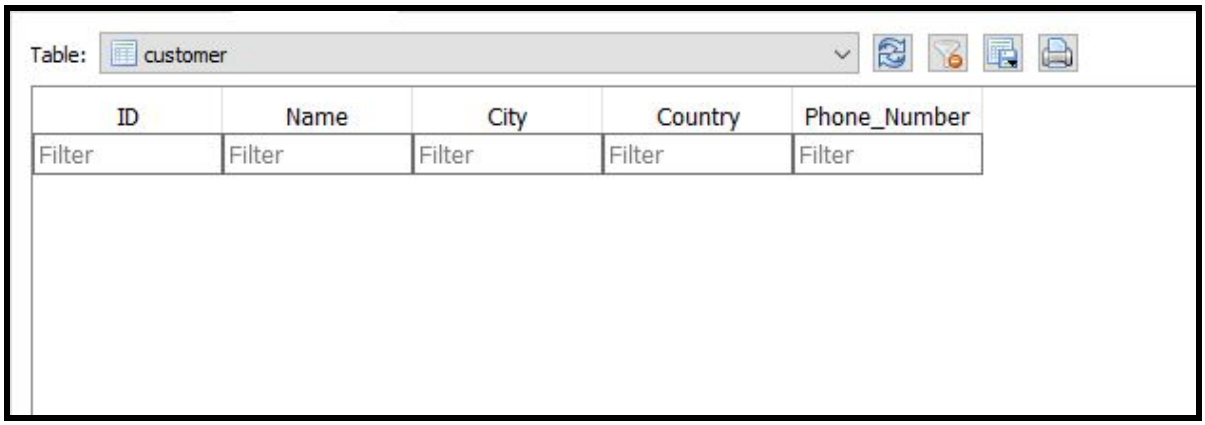
- Every type can have the special value null.
- The NULL term is used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.
- A field with a NULL value is a field with no value.
  Note: A NULL value is different than a zero value or a field that contains 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.

Create table:

- Let's say we want to create a customer table such that the relational schema is this customer(ID, name, city, country, phone_number)
- The code is this: **CREATE TABLE customer('ID' INTEGER PRIMARY KEY AUTOINCREMENT, 'Name' TEXT, 'City' TEXT, 'Country' TEXT, 'Phone_Number' TEXT);**

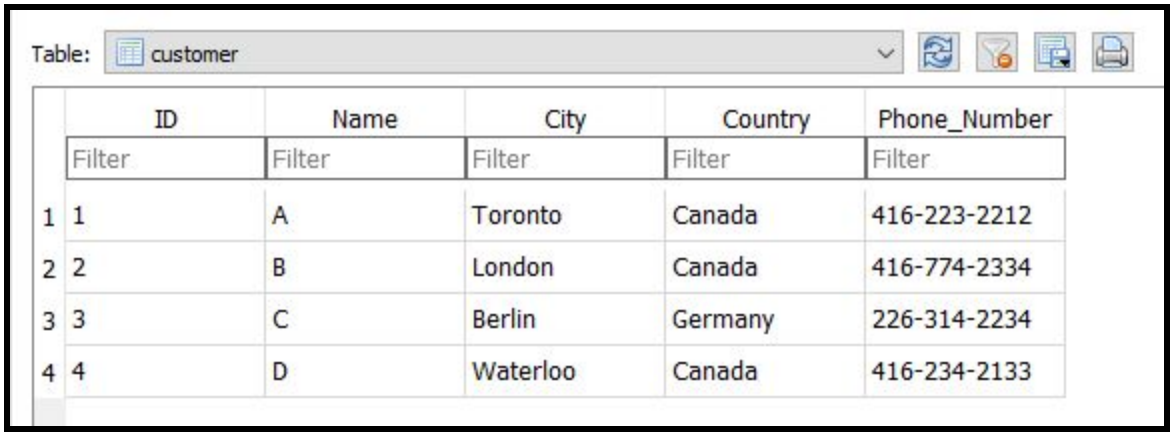
Insert Into:

- To add information into a table, we use the insert into command.
- Right now, the customer table is empty, as shown below.



- If we run the below commands, we get the following table:

INSERT INTO customer(Name, City, Country, Phone_Number) VALUES("A", "Toronto", "Canada", "416-223-2212");
INSERT INTO customer(name, city, country, phone_number) VALUES ('B', 'London', 'Canada', '416-774-2334');
INSERT INTO customer (name, city, country, phone_number) VALUES ('C', 'Berlin', 'Germany', '226-314-2234');
INSERT INTO customer(name, city, country, phone_number) VALUES ('D', 'Waterloo', 'Canada', '416-234-2133');



	ID	Name	City	Country	Phone_Number
	Filter	Filter	Filter	Filter	Filter
1	1	A	Toronto	Canada	416-223-2212
2	2	B	London	Canada	416-774-2334
3	3	C	Berlin	Germany	226-314-2234
4	4	D	Waterloo	Canada	416-234-2133

Select:

- To get all the columns of the customer relation, we'd run this command:
  SELECT * FROM customer;
  The output table looks like this:

	ID	Name	City	Country	Phone_Number
1	1	A	Toronto	Canada	416-223-2212
2	2	B	London	Canada	416-774-2334
3	3	C	Berlin	Germany	226-314-2234
4	4	D	Waterloo	Canada	416-234-2133

- To get the column "Country", we'd run this command:
  SELECT country FROM customer;
  The output table looks like this:

	Country
1	Canada
2	Canada
3	Germany
4	Canada

- To get all the customers from Canada, we'd run this command:
  SELECT * FROM customer WHERE country IS 'Canada';
  The output table looks like this:

	ID	Name	City	Country	Phone_Number
1	1	A	Toronto	Canada	416-223-2212
2	2	B	London	Canada	416-774-2334
3	4	D	Waterloo	Canada	416-234-2133

- To get the ID column from the table, but rename is customerID, we'd run this command:
  SELECT ID AS CUSTOMERID FROM customer;
  The output table looks like this:

	CUSTOMERID
1	1
2	2
3	3
4	4

Select Distinct:

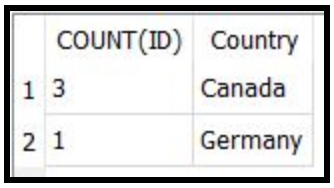
- To get all the different countries in the column "Country", we'd run this command:
  SELECT DISTINCT country FROM customer;

The output table looks like this:

	Country
1	Canada
2	Germany

Group by, Count, Having, Max, Min, Order By, Limit:

- If we want to get the number of customers in each country, we'd run this command: **SELECT COUNT(ID), Country FROM customer GROUP BY Country;**
  The output table looks like this:



	COUNT(ID)	Country
1	3	Canada
2	1	Germany

- If we want to get how many customers are in Canada, we'd run this command: **SELECT COUNT(ID), Country FROM customer GROUP BY Country HAVING Country IS 'Canada';**
  The output table looks like this:

	COUNT(ID)	Country
1	3	Canada

- If we want to get the country that has the maximum number of customers, we'd run this command: **SELECT country FROM (SELECT COUNT(ID) mycount, Country FROM customer GROUP BY Country) ORDER BY mycount DESC LIMIT 1;**
  The output table looks like this:

	Country
1	Canada

- If we want to get the country that has the least number of customers, we'd run this command:
  SELECT country FROM (SELECT COUNT(ID) mycount, country FROM customer GROUP BY country) ORDER BY mycount LIMIT 1;
  The output table looks like this:

	country
1	Germany

Update:

- If we want to change the phone number of customer A to "416-223-2222", we'd run this command: **UPDATE customer SET Phone_Number = '416-223-2222' where ID = 1;**

The customer relation now looks like this:

	ID	Name	City	Country	Phone_Number
	Filter	Filter	Filter	Filter	Filter
1	1	A	Toronto	Canada	416-223-2222
2	2	B	London	Canada	416-774-2334
3	3	C	Berlin	Germany	226-314-2234
4	4	D	Waterloo	Canada	416-234-2133

Suppose we have the following 2 relations:

Table: Student

	ID	name	address
	Filter	Filter	Filter
1	1	A	48 XYZ Drive
2	2	B	10 ABC Road
3	3	C	20 ABC Ave
4	4	D	20 ABC Ave
5	5	E	3452 XYZ Drive
6	7	G	1234 X Ave

and

Table: marks

	ID	mark	student_name	class
	Filter	Filter	Filter	Filter
1	1	85	A	science
2	2	67	B	math
3	3	90	C	math
4	4	56	D	math
5	5	85	E	english
6	6	75	F	french

Cross Join:

- To get a cartesian product of the student and marks relations, I'd run the command:
  SELECT * FROM Student CROSS JOIN Marks;
  The output relation is this:

ID	name	address	ID	mark	student_name	class
1	A	48 XYZ Drive	1	85	A	science
1	A	48 XYZ Drive	2	67	B	math
1	A	48 XYZ Drive	3	90	C	math
1	A	48 XYZ Drive	4	56	D	math
1	A	48 XYZ Drive	5	85	E	english
1	A	48 XYZ Drive	6	75	F	french
2	B	10 ABC Road	1	85	A	science
2	B	10 ABC Road	2	67	B	math
2	B	10 ABC Road	3	90	C	math
2	B	10 ABC Road	4	56	D	math
2	B	10 ABC Road	5	85	E	english

2	B	10 ABC Road	6	75	F	french
3	C	20 ABC Ave	1	85	A	science
3	C	20 ABC Ave	2	67	B	math
3	C	20 ABC Ave	3	90	C	math
3	C	20 ABC Ave	4	56	D	math
3	C	20 ABC Ave	5	85	E	english
3	C	20 ABC Ave	6	75	F	french
4	D	20 ABC Ave	1	85	A	science
4	D	20 ABC Ave	2	67	B	math
4	D	20 ABC Ave	3	90	C	math
4	D	20 ABC Ave	4	56	D	math
4	D	20 ABC Ave	5	85	E	english
4	D	20 ABC Ave	6	75	F	french
5	E	3452 XYZ Drive	1	85	A	science
5	E	3452 XYZ Drive	2	67	B	math
5	E	3452 XYZ Drive	3	90	C	math
5	E	3452 XYZ Drive	4	56	D	math
5	E	3452 XYZ Drive	5	85	E	english
5	E	3452 XYZ Drive	6	75	F	french
7	G	1234 X Ave	1	85	A	science
7	G	1234 X Ave	2	67	B	math
7	G	1234 X Ave	3	90	C	math
7	G	1234 X Ave	4	56	D	math
7	G	1234 X Ave	5	85	E	english
7	G	1234 X Ave	6	75	F	french

Left Join:

- Running the command **SELECT * FROM student LEFT JOIN marks ON student.id = marks.id;** gets me the following relation:

	ID	name	address	ID	mark	student_name	class
1	1	A	48 XYZ Drive	1	85	A	science
2	2	B	10 ABC Road	2	67	B	math
3	3	C	20 ABC Ave	3	90	C	math
4	4	D	20 ABC Ave	4	56	D	math
5	5	E	3452 XYZ Drive	5	85	E	english
6	7	G	1234 X Ave	*NULL*	*NULL*	*NULL*	*NULL*

- Running the command **SELECT * FROM marks LEFT JOIN student ON student.id = marks.id;** gets me the following relation:

	ID	mark	student_name	class	ID	name	address
1	1	85	A	science	1	A	48 XYZ Drive
2	2	67	B	math	2	B	10 ABC Road
3	3	90	C	math	3	C	20 ABC Ave
4	4	56	D	math	4	D	20 ABC Ave
5	5	85	E	english	5	E	3452 XYZ Drive
6	6	75	F	french	*NULL*	*NULL*	*NULL*

Inner Join:

- Running the command **SELECT * FROM marks INNER JOIN student ON student.id = marks.id;** gets me the following relation:

	ID	mark	student_name	class	ID	name	address
1	1	85	A	science	1	A	48 XYZ Drive
2	2	67	B	math	2	B	10 ABC Road
3	3	90	C	math	3	C	20 ABC Ave
4	4	56	D	math	4	D	20 ABC Ave
5	5	85	E	english	5	E	3452 XYZ Drive

Suppose we have the following 2 relations:

Table: A

	x	y
	Filter	Filter
1	1	a
2	2	b
3	3	c
4	4	d

and

Table: B

	x	y
	Filter	Filter
1	1	x
2	2	b
3	3	c

Union:

- If I run the command **SELECT * FROM A UNION SELECT * FROM B;** I'd get this table:

	x	y
1	1	a
2	1	x
3	2	b
4	3	c
5	4	d

Intersect:

- If I run the command **SELECT * FROM A INTERSECT SELECT * FROM B;** I'd get this table:

	x	y
1	2	b
2	3	c

Except:

- If I run the command **SELECT * FROM A EXCEPT SELECT * FROM B;** I'd get this table:

	x	y
1	1	a
2	4	d

Create View:

- Creating views can be used to break long SQL queries into smaller pieces.
- E.g. Suppose we want to find the names of all students with an average greater than or equal to 85. The relational schemas are given below and the primary ids are underlined:
  student(id, name)
  marks(id, average)

  To do this using views, we would run the following commands:
  CREATE VIEW v1 AS SELECT ID FROM marks WHERE Average >= 85;
  SELECT S.name FROM Student S NATURAL JOIN v1;

  Suppose we tested the above 2 lines on these 2 relations:

Table: Student

	ID	name
	Filter	Filter
1	1	A
2	2	B
3	3	C
4	4	D
5	5	E

and

Table: marks

	ID	Average
	Filter	Filter
1	1	85
2	2	67
3	3	90
4	4	56
5	5	85

The first line, **CREATE VIEW v1 AS SELECT ID FROM marks WHERE Average >= 85;**, would create a view shown below:

Table: v1

	ID
	Filter
1	1
2	3
3	5

The second line, **SELECT S.name FROM Student S NATURAL JOIN v1;**, would create the output table shown below:

	name
1	A
2	C
3	E

SQL Group By Examples

Group By:

- The GROUP BY clause is a SQL command that is used to group rows that have the same values.
- It is used in conjunction with aggregate functions to produce summary reports from the database.

Having:

- The HAVING clause is used with the GROUP BY clause to filter groups based on a specified list of conditions.
- The GROUP BY clause summarizes the rows into groups and the HAVING clause applies one or more conditions to these groups. Only groups that make the conditions evaluate to TRUE are included in the result. In other words, the groups for which the condition evaluates to FALSE or UNKNOWN are filtered out.

Example:

Given the table below, find the following items:

Table: student

	name	class	mark
	Filter	Filter	Filter
1	A	Math	78
2	A	English	67
3	A	Chemistry	83
4	B	Math	89
5	B	Chemistry	93
6	C	English	75
7	C	Biology	66
8	D	French	78

1. Find the names of the students taking exactly 3 classes.
   Solution: **select name from student group by name having count(class) = 3;**

	name
1	A

2. Find the names of the students taking more than 1 class.
   Solution: **select name from student group by name having count(class) > 1;**

	name
1	A
2	B
3	C

3. Find the average of each student.
   Solution: **select name, avg(mark) from student group by name;**

	name	avg(mark)
1	A	76.0
2	B	91.0
3	C	70.5
4	D	78.0

4. Find the name of the student with the highest mark in each class.
   Solution: **select name, class, max(mark) from student group by class;**

	name	class	max(mark)
1	C	Biology	66
2	B	Chemistry	93
3	C	English	75
4	D	French	78
5	B	Math	89

5. Find the lowest mark for each student.
   Solution: **select name, min(mark) from student group by name;**

	name	min(mark)
1	A	67
2	B	89
3	C	66
4	D	78

6. Find the names of students who have an average of 75 or higher.
   Solution: **select name from student group by name having avg(mark) >= 75;**

	name
1	A
2	B
3	D

CSCB20 Count, Sum, Avg, Min, Max Notes

Suppose we have this table:

Table: student

	name	class	mark
	Filter	Filter	Filter
1	A	Math	78
2	A	English	67
3	A	Chemistry	83
4	B	Math	89
5	B	Chemistry	93
6	C	English	75
7	D	Biology	78
8	C	French	66

Avg:

- The AVG function returns the average value of a numeric column.
- If we do **select avg(mark) from student;** we get this:

	avg(mark)
1	78.625

- However, if we use AVG in conjunction with GROUP BY, it gets the average value of a numeric column for each grouping.
- If we do **select name, avg(mark) from student group by name;** we get this:

	name	avg(mark)
1	A	76.0
2	B	91.0
3	C	70.5
4	D	78.0

Count:

- The COUNT function returns the number of rows that matches a specified criteria.
- If we do **select count(class) from student;** we get this:

	count(class)
1	8

- The COUNT function with the DISTINCT clause eliminates the repetitive appearance of the same data.
- If we do **select count(distinct class) from student;** we get this:

	count(distinct class)
1	5

- However, if we use COUNT in conjunction with GROUP BY, it gets the number of rows that matches a specified criteria for each grouping.

- If we do **select name, count(class) from student group by name;** we get:

	name	count(class)
1	A	3
2	B	2
3	C	2
4	D	1

Max:

- The MAX function returns the largest value of the selected column.
- If we do **select max(mark) from student;** we get:

	max(mark)
1	93

- However, if we use MAX in conjunction with GROUP BY, it gets the largest value of the selected column for each grouping.
- If we do **select name, max(mark) from student group by name;** we get:

	name	max(mark)
1	A	83
2	B	93
3	C	75
4	D	78

Min:

- The MIN function returns the smallest value of the selected column.
- If we do **select min(mark) from student;** we get:

	min(mark)
1	66

- However, if we use MIN in conjunction with GROUP BY, it gets the smallest value of the selected column for each grouping.
- If we do **select name, min(mark) from student group by name;** we get:

	name	min(mark)
1	A	67
2	B	89
3	C	66
4	D	78

Sum:

- The SUM function returns the total sum of a numeric column.
- If we do **select sum(mark) from student;** we get:

	sum(mark)
1	629

- However, if we use SUM in conjunction with GROUP BY, it gets the sum of a numeric column for each grouping.
- If we do **select name, sum(mark) from student group by name;** we get:

	name	sum(mark)
1	A	228
2	B	182
3	C	141
4	D	78

Relational Algebra Examples

Consider the 2 relations below:

A

Num	Square
1	1
2	4

B

Num	Square
1	1
3	9

1. Selection
   a) $\sigma_{Num>1}(A)$ will give us:

Num	Square
2	4

   b) $\sigma_{num=3}(B)$ will give us:

Num	Square
3	9

- Denoted by $\sigma_{p}(x)$ where
  - σ is the selection symbol.
  - P is the propositional logic.
  - X is the name of the relation.

- Seleetion gets all rows from the given relation that satisfies the propositional logic.

2. Projection

- Denoted by $\Pi_{col1, col2, ..., coln}(X)$ where
  - Π is the projection symbol.
  - col1, col2, ..., coln are the names of columns.
  - X is the name of the relation.

- Gets all the columns listed from the given relation.

a) $\Pi_{Num}(A)$ will give us:

Num
1
2

3. Natural Join

- Denoted by $r \bowtie s$ where r and s are relations.
- It combines 2 relations into 1. It can only be used if there is at least 1 column in both relations that have the same name and same domain. It only gets rows that are in both relations.
- Also called inner join.

a) $A \bowtie B$ will give us

Num	A. square	B. square
1	1	1

4. Left Join
   - Denoted by $R \bowtie_L S$ where R and S are relations.
   - keeps all tuples in the left relation and tries to find matching tuples in the right relation. If there is no matching tuple, a null value is used.

   a) $A \bowtie_L B$ will give us:

Num	A. square	B. square
1	1	1
2	4	NULL

5. Right Join
   - Denoted by $R \bowtie_R S$ where R and S are relations.
   - keeps all tuples in the right relation and tries to find matching tuples in the left relation. If there is no matching tuple, a null value is used.

   a) $A \bowtie_R B$ will give us

Num	A. square	B. square
1	1	1
3	NULL	3

6. Full Join:
   - Denoted by R ⟗ S where R and S are relations.
   - All tuples from both relations are included in the result.

a) A ⟗ B will give us:

Num	A. square	B. square
1	1	1
2	4	NULL
3	NULL	9

7. Cartesian Join:
   - Denoted by R X S where R and S are relations.
   - Produces all pairs of rows of the 2 relations that are possible.
   - If relation R has X rows and relation S has Y rows, then R X S will have X·Y rows.
   - Also called cross join.

a) A X B will give us:

A. n	A. s	B. n	B. s
1	1	1	1
1	1	3	9
2	4	1	1
2	4	3	9

8. Theta Join:
   - Denoted by R ⋈c S where R and S are relations and c is a propositional logic or condition.
   - With R ⋈c S, you first find R X S and then do σc on that.

a) A ⋈(B.S > A.S) B

We got A X B in the last example. Filtering out the rows where B.square > A.square, we get

A.n	A.S	B.n	B.S
1	1	3	9
2	4	3	9

9. Union:
   - Denoted by R ∪ S where R and S are relations.
   - It will give a relation with tuples which are either in R or S. It will also eliminate all duplicate tuples.
   - For a union operation to be valid, the following conditions must hold
     - The 2 relations must have the same number of columns.
     - The domain of the columns must be compatible.

a) A ∪ B will give us:

Num	Square
1	1
2	4
3	9

10. Difference:
- Denoted by R-S where R and S are relations.
- It returns a relation consisting of all tuples in R and not in S.

a) A - B will give us:

Num	Square
2	4

b) B - A will give us:

Num	Square
3	9

11. Intersection:
- Denoted by R∩S where R and S are relations.
- It returns a table consisting of all tuples in both R and S.
- R ∩ S is equalivent to R-(R-S)

a) A ∩ B will give us:

Num	Square
1	1

12. Finding "every" using relational algebra:

Consider the 2 relational schemas below:

Student(id, name)
Marks (id, class, mark)

Find the id of the students taking every class.

Soln:

1. $R_1 = \Pi_{id}(S) \times \Pi_{class}(M)$
2. $R_2 = R_1 - \Pi_{id, class}(M)$
3. $R_3 = \Pi_{id}(R_1) - \Pi_{id}(R_2)$

R_1 is a relation with every possible permutation of [illegible] id and classes. Therefore, it is a table of every student taking every class.

When you do $R_1 - \Pi_{id, class}(M)$, you are removing all instances of a student actually taking a class from R_1. Therefore, R_2 is a table of the students not taking class(es). If a student is taking every class, their id wouldn't be in R_2.

Since $\Pi_{id}(R_1)$ contains the id of every student and $\Pi_{id}(R_2)$ contains the id of the students who are not taking some classes, $\Pi_{id}(R_1) - \Pi_{id}(R_2)$ will give us a table of the ids of the students taking every class.

13. Finding the biggest or highest:

Using the schemas in 12, find the id of the students who has the highest mark.

Soln:

1. $R_1 = \Pi_{id, mark}(M)$
2. $R_2 = \rho_{R2}(R1)$
3. $R_3 = \rho_{R3}(R1)$
4. $R_4 = R_2 \bowtie_{(R2.mark < R3.mark)} R_3$
5. $R_5 = \Pi_{id}(R_1) - \Pi_{R2.id}(R_4)$

R_1 is just a relation with only the id and mark columns from Marks.
R_2 and R_3 are just renamed instances of R_1.
R_4 is a relation of all possible permutations between R_2 and R_3 where $R_2.mark < R_3.mark$. That means $R_2.id$ in R_4 does not contain the id of the students with the highest mark.

However, R2.id in R4 contains the id of all other students. Therefore, $\Pi_{id}(R_1) - \Pi_{R2.id}(R_4)$ gets back a table with the id of the students with the highest mark.

14 Finding the second biggest or second highest:

Using the schemas in 12, find the id of the students who have the second highest mark.

Soln:

$R_1 = \Pi_{id,\ mark}(M)$

$R_2 = \rho_{R2}(R_1)$

$R_3 = \rho_{R3}(R_1)$

$R_4 = R_2 \bowtie_{(R2.mark < R3.mark)} R_3$

$R_5 = \Pi_{R2.id,\ R2.mark}(R_4)$

$R_6 = \rho_{R6}(R_5)$

$R_7 = R_5 \bowtie_{(R5.mark < R6.mark)} R_6$

$R_8 = \Pi_{id}(R_5) - \Pi_{R5.id}(R_7)$

The steps R_1 to R_4 are used to remove the id of the students with the highest mark. In R_4, R2.id is a relation that does not contain the id of the students with the highest mark. R_5 is a table consisting of the R2.id and R2.mark cols from R_4.

As such, it doesn't have the id of the students with the highest mark. R6 is a renamed instance of R5. R7 is a relation where R5.id doesn't contain the id of the student with the current highest mark. They used to be the students with the second highest marks. R8 is a table with the id of the students with the second highest mark. $\Pi_{id}(R5)$ contains all the id of the students that do not have the highest mark. $\Pi_{R5.id}(R7)$ contains the id of the students who do not have the highest or second highest marks. Therefore, $\Pi_{id}(R5) - \Pi_{R5.id}(R7)$ gets you the id of the students who have the second highest mark.

15. Finding "at least" using relational algebra.

Using the 2 schemas from 12, find the id of the students taking at least 2 courses.

Soln:

$R_1 = \Pi_{id, class}(M)$

$R_2 = \rho_{R2}(R_1)$

$R_3 = R_1 \bowtie (R1.id = R2.id \text{ and } R1.class != R2.class) R2$

You're finding all the rows of $R_1 \times R_2$ where R1.id equals to R2.id but R1.class doesn't equal to R2.class.

Using the schemas from 12, find the id of the students taking at least 3 classes.

Soln:

$R_1 = \Pi_{id, class}(M)$

$R_2 = \rho_{R2}(R_1)$

$R_3 = \rho_{R3}(R_1)$

$R_4 = R_1 \times R_2 \times R_3$

$R_5 = \sigma$(R1.id = R2.id and R1.id = R3.id and R1.class != R2.class and R1.class != R3.class and R2.class != R3.class)(R4)

16. Finding "exactly" using relational algebra.

Using the schemas from 12, find the id of the students taking exactly 2 courses.

Soln:

$R_1 = \Pi_{id, Class}(M)$

$R_2 = \rho_{R2}(R_1)$

$R_3 = \rho_{R3}(R_1)$

$R_4 = R_1 \bowtie (R1.id = R2.id \text{ and } R1.class != R2.class) R2$

$R_5 = R_1 \times R_2 \times R_3$

$R_6 = \sigma_{(R1.id = R2.id \text{ and } R1.id = R3.id \text{ and } R1.class != R2.class \text{ and } R1.class != R3.class \text{ and } R2.class != R3.class)}(R_5)$

$R_7 = \Pi_{R1.id}(R_4) - \Pi_{R1.id}(R_6)$

To find exactly n of something, find at least n and at least n+1 and subtract at least n from at least n+1. In this example, to find the ids of the students taking exactly 2 courses, we found the id of the students taking at least 2 courses and subtracted it from the id of the students taking at least 3 courses.

Examples of Joins in Relational Algebra

Suppose we have the following 2 relations below:

A

Num	Square
1	1
2	4
3	9

B

Num	Cube
2	8
3	27
4	64

Natural Join:

$A \bowtie B$ will get us the relation:

Num	Square	Cube
2	4	8
3	9	27

Left Join:

$A \bowtie_L B$ will get us the relation:

Num	Square	Cube
1	1	Null
2	4	8
3	9	27

Right Join:

$A \bowtie_R B$ will get us the relation:

Num	Square	Cube
2	4	8
3	9	27
4	Null	64

Examples of Joins in Relational Algebra

Full Outer Join:

A $\bowtie_O$ B will get us the relation:

Num	**Square**	**Cube**
1	1	Null
2	4	8
3	9	27
4	Null	64

Cartesian Join:

A X B will get us the relation:

A.num	A.square	B.num	B.cube
1	1	2	8
1	1	3	27
1	1	4	64
2	4	2	8
2	4	3	27
2	4	4	64
3	9	2	8
3	9	3	27
3	9	4	64

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.
- **<p>:** 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.

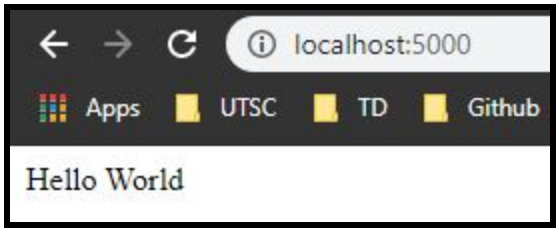
- A Flask application is started by calling the run() method. However, while the application is under development, it should be restarted manually for each change in the code. To avoid this inconvenience, enable debug support. The server will then reload itself if the code changes. It will also provide 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.

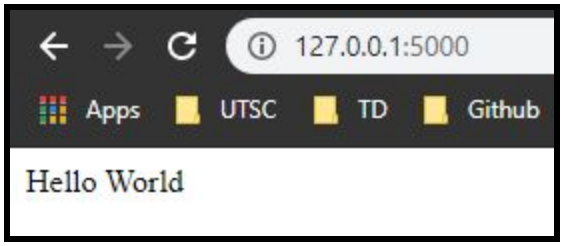
```
from flask import Flask
app = Flask(__name__)

@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 -
```

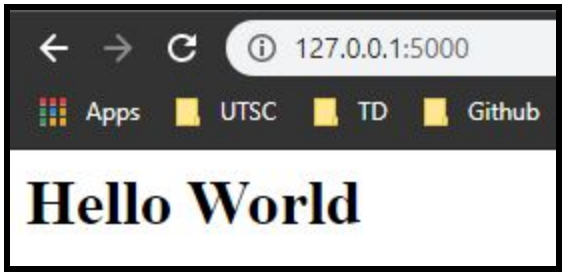
However, if I put the debug statements in my code like such:

```python
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

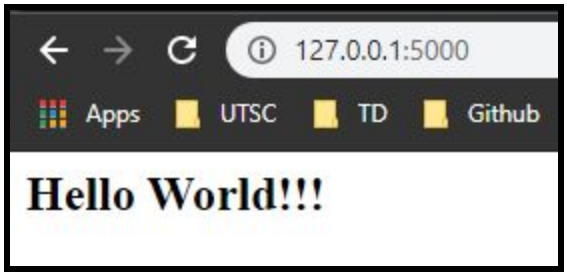


- Suppose I modify my code:

```python
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:

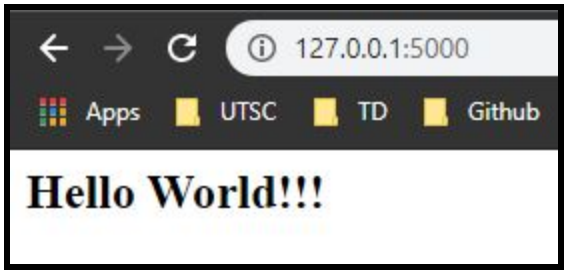
```python
from flask import Flask
app = Flask(__name__)

@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)
```

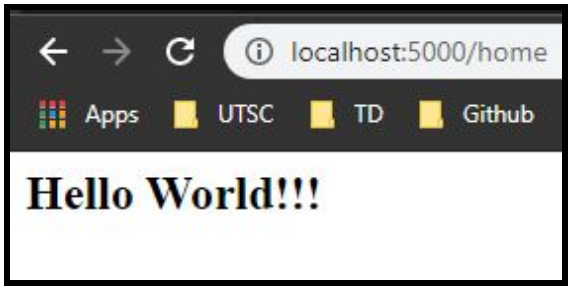
- If I type in http://localhost:5000/about in my browser, I see this:

- 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.

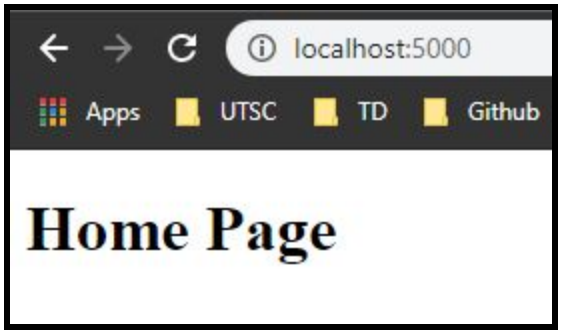
Flask1.py | home.html

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h1> Home Page </h1>
</body>
</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.



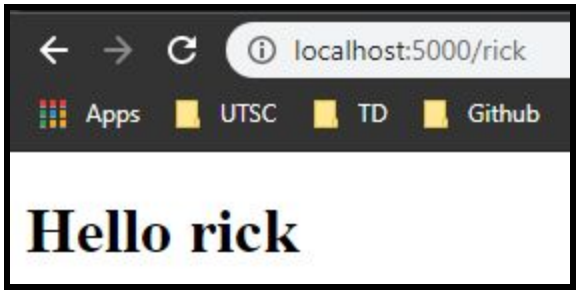
Flask Variables:

- 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:



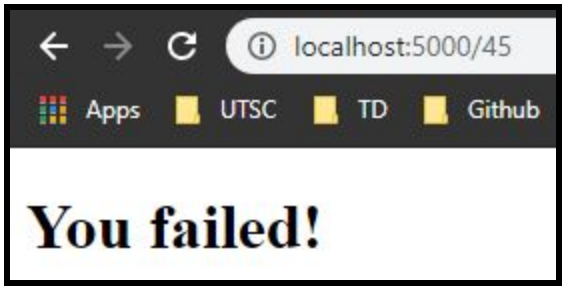
Flask If Statements:

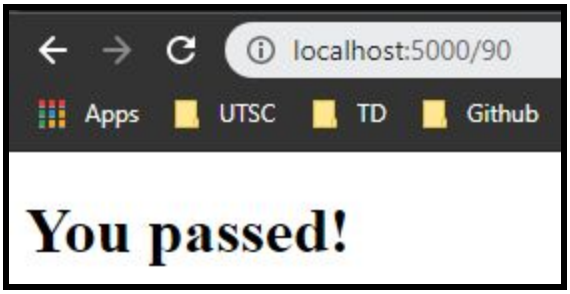
- In Flask, if-else and endif are enclosed in delimiter {%..%}.
- You need to end an if-else statement with endif.
- E.g. Suppose we have these 2 pieces of code.

```
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:





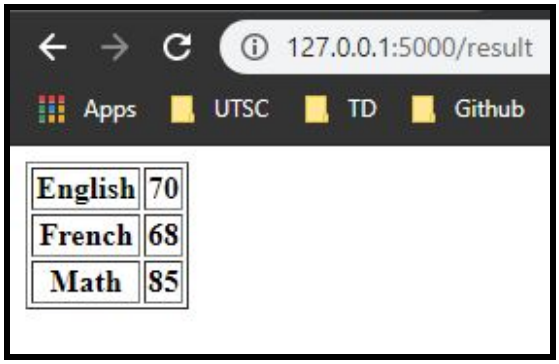
Flask For Loops:

- 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:

HTTP Protocol:

- HTTP protocol is the foundation of data communication in world wide web. Different methods of data retrieval from specified URL are defined in this protocol.
- Here is a table of HTTP Methods:

HTTP Method	Function
Get	Sends data in unencrypted form to the server. Most common method.
Head	Same as GET, but without a response body.
Post	Used to send HTML form data to a server. Data received by the POST method is not cached by server.
Put	Replaces all current representations of the target resource with the uploaded content.
Delete	Removes all current representations of the target resource given by a URL.

- By default, the Flask route responds to the GET requests.

Starting a Flask app:

- A basic Flask app is structured as follows:

```
/home/my-flask-app/
|___ app.py
|___ templates/
     |___ base.html
|___ static/
     |___ style.css
```

- **/home/my-flask-app** would be the directory of your Flask application.
- **app.py** is your main file and is what runs the app. In order to start your application, you can run python app.py or python3 app.py depending on your Python version. This will open up your application at http://localhost:5000.
- The **templates/** directory contains all the HTML templates for what you will serve to the browser (i.e. what the user will see). Flask expects HTML files to be in a folder called templates/ so ensure your folder is named appropriately. If this is named improperly, render_template() will not work
- The **static/** directory contains all of the static, non-changing elements of the site (i.e. such as the CSS / JavaScript). If you choose to write CSS, you would place it in this folder.

- Your basic barebones app.py file will looks as follows:

```
# required imports.
# You may add more in the future; currently, we will only use
# render_template
from flask import Flask, render_template

# tells Flask that "this" is the current running app
app = Flask(__name__)

# setup the default route
# this is the page the site will load by default (i.e. like the home page)
@app.route('/')
def root():
    # tells Flask to render the HTML page called index.html
    return render_template('index.html')

# run the app when app.py is run
if __name__ == '__main__':
    app.run()
```

Getting Input From a Form:

- Flask can send data to app.py from the browser using an HTTP request. In order to receive data from the web page, Flask receives a GET request from a **<form></form>** block whenever the user presses the corresponding **<input type="submit"></input>** button.
  A block in a webpage that can send input to Flask looks as follows:

```
<form action="/add-comment">
        <input type="text" name="comment-input" placeholder="enter a new
        comment">
        <input type="submit" name="comment-submit" value="submit">
</form>
```

- **NOTE:** The GET request is not supposed to be used to modify data (this is a violation of the standard). Don't worry about this for now but in a later lab we will improve this example by refactoring this.
- Flask automatically sends a GET request to the endpoint provided in the action attribute inside the **<form></form>** tag. We can catch this particular endpoint in app.py by specifying it using **app.route('/add-comment')** or substituting **/add-comment** for any other endpoint you may decide to use. To get the input the user has submitted, we make use of a module of Flask called requests. Requests searches for any **<input></input>** tags inside the corresponding **<form></form>** tags and captures these values. To access them, in app.py we can using **request.args.get('comment-input')** as follows:

```
@app.route('/add-comment')
def add_comment():
    global comments
    comments.append(request.args.get('comment-input')
    ...
```

The **request.args.get()** command returns the value inside the input that matches the name attribute inside the **<input></input>** tags. Notice how comment-input matches the corresponding value in the HTML code. Additionally, notice how the variable comments is referred to using the global keyword. This is Python syntax for creating a global variable for the comments outside of the add_comment() function. Generally, this practice of using global variables is not recommended but we will refactor this later when we'll use an SQLite database instead of a global variable.

- To use send data from Flask to the browser, we can pass in additional parameters to the render_template() function. Consider the sample code:

```
@app.route('/')
def root():
    return render_template('home.html',data=posts, title="abbas")
```

The browser is now able to access these variables using the keys they were given.

Form:

- The HTML **<form>** element defines a form that is used to collect user input.
- An HTML form contains form elements. Form elements are different types of input elements, like text fields, checkboxes, radio buttons, submit buttons, and more.
- There are two basic components of a web form: **the shell** which is the part that the user fills out and **the script** which processes the information.
- The shell has three important parts:
    1. The <FORM> tag, which includes the address of the script which will process the form.
    2. The form elements, like text boxes and radio buttons.
    3. The submit button which triggers the script to send the entered information to the server
- The general syntax of form is:
  <form>
  form elements
  </form>
- The action attribute defines the action to be performed when the form is submitted. Normally, the form data is sent to a web page on the server when the user clicks on the submit button. We add it to the opening <form> tag.
  E.g. **<form action = "server name">**
- **Note:** Flask automatically sends a GET request to the endpoint provided in the action attribute inside the <form></form> tag. As such, make sure app.route() matches the action attribute in the form tags.

Input:

- The **<input>** element is the most important form element.
- The **<input>** element can be displayed in several ways, depending on the type attribute.
  E.g. Here is a table of the different types of inputs.

Type	Description
<input type="text">	Defines a one-line text input field
<input type="submit">	Defines a submit button (for submitting the form)

- **<input type="text">** defines a single-line input field for text input.
- **<input type="submit">** defines a button for submitting the form data to a form-handler. The form-handler is typically a server page with a script for processing input data. The form-handler is specified in the form's action attribute.

  E.g. Here is a snippet of code that has both input type="text" and input type="submit".

```
<form action="/add-comment">
   <input type="text" name="comment-input" placeholder="enter a new comment">
   <input type="submit" name="comment-submit" value="submit">
</form>
```

- The **name attribute** in <input> is used to specify a name for an <input> element. It is used to reference the form-data after submitting the form. Each input field must have a name attribute to be submitted. If the name attribute is omitted, the data of that input field will not be sent at all. For Flask, if you are using request.args.get(), ensure that your

input tags have the name attribute with the matching name as the input in request.args.get().
I.e. The input in request.args.get() must be the same as the name attribute in <input>.

- The **value attribute** in <input> specifies an initial value for an input field.
- The **placeholder attribute** in <input> specifies a short hint that describes the expected value of an input field (e.g. a sample value or a short description of the expected format). The short hint is displayed in the input field before the user enters a value.

SELECT:

- The <select> element is used to create a drop-down list.
- The <option> tags inside the <select> element define the available options in the list.

Label:

- The <label> tag in HTML is used to provide usability improvement for mouse users. I.e, if a user clicks on the text within the <label> element, it toggles the control. The <label> tag defines the label for <button>, <input>, <meter>, <output>, <progress>, <select>, or <textarea> element.
- General syntax: **<label> form content </label>**
- We can put the <input> tag use directly inside the <label> tag.

Creating a form shell:

- Type **<FORM> METHOD=POST ACTION=url>**
- Create the form elements.
- End with a closing </FORM> tag.

The url specifies where to send the form-data when the form is submitted.
The POST request method requests that a web server accepts the data enclosed in the body of the request message, most likely for storing it.

Creating text boxes:

- To create a text box, type **<INPUT TYPE="text" NAME="name" VALUE="value" SIZE="n" MAXLENGTH="n">**
- The <input> tag specifies an input field where the user can enter data.
- The NAME, VALUE, SIZE, and MAXLENGTH attributes are optional.
- The NAME attribute is used to identify the text box to the processing script.
- The VALUE attribute is used to specify the text that will initially appear in the text box.
- The SIZE attribute is used to define the size of the box in characters.
- The MAXLENGTH attribute is used to define the maximum number of characters that can be typed in the box.

Creating larger text areas:

- To create larger text areas, type **<TEXTAREA NAME="name" ROWS=n1 COLS=n2 WRAP> Default Text </TEXTAREA>**
- n1 is the height of the text box in rows and n2 is the width of the text box in characters.
- The WRAP attribute causes the cursor to move automatically to the next line as the user types.

Creating radio buttons:

- To create a radio button, type **<INPUT TYPE="radio" NAME="name" VALUE="Data">Label**
- The <input> tag specifies an input field where the user can enter data.
- "Data" is the text that will be sent to the server if the button is checked.
- "Label" is the text that identifies the button to the user.

Creating checkboxes:

- To create a checkbox, type **<INPUT TYPE="checkbox" NAME="name" VALUE="value">Label**
- **Note:** If you give a group of radio buttons or checkboxes the same name, the user will only be able to select one button or box at a time.
- The <input> tag specifies an input field where the user can enter data.

Creating drop-down menus:

- To create a drop-down menu, type
  <SELECT NAME="name" SIZE=n MULTIPLE>
  <OPTION VALUE= "value1">Label1
  <OPTION VALUE= "value2">Label2
  ...
  </SELECT>
- The SIZE attribute specifies the height of the menu in lines. This is optional.
- MULTIPLE allows users to select more than one menu option. This is optional.
- The <select> element is used to create a drop-down list.
- The <option> tags inside the <select> element define the available options in the list.

Creating a Submit Button:

- To create a submit button, type **<INPUT TYPE="submit" VALUE="NAME">**
- The <input> tag specifies an input field where the user can enter data.

Creating a Reset Button:

- To create a reset button, type **<INPUT TYPE="reset" VALUE="name">**
- The <input> tag specifies an input field where the user can enter data.

Form Element:

- The HTML <form> element defines a form that is used to collect user input.
- General syntax:
  <form>
  … form elements ...
  </form>
- An HTML form contains form elements.
- Form elements are different types of input elements, like: text fields, checkboxes, radio buttons, submit buttons, and more.

Action Attribute:

- The action attribute defines the action to be performed when the form is submitted.
- Usually, the form data is sent to a page on the server when the user clicks on the submit button.
- For Flask, you should set the action attribute to be the same as app.route. For example, if you have

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

  Then, in your index.html file, you need to set action="/" in the form tag.
  I.e.

```
<form action="/">
…
</form>
```

Method Attribute:

- The method attribute specifies the HTTP method (GET or POST) to be used when submitting the form data.
  - **Notes on GET:**
    - In the GET method, after the submission of the form, the form values will be visible in the address bar of the new browser tab.
      I.e. It appends form-data into the URL in name/value pairs.
    - It has a limited size of 2048 characters.
    - It is only useful for non-secure data not for sensitive information. Never use GET to send sensitive data as it will be visible in the URL. GET is better for non-secure data, like query strings in Google.
    - It is useful for form submissions where a user wants to bookmark the result.
    - The default HTTP method when submitting form data is GET.
      I.e. If you do not specify which HTTP method you want to use, it will be GET.
  - **Notes on POST:**
    - In the post method, after the submission of the form, the form values will not be visible in the address bar of the new browser tab.
    - It appends form data inside the body of the HTTP request.
    - It has no size limitation.
    - This method does not support bookmark the result.
- General Syntax:
  <form method="GET|POST">

…
</form>

- **Note:** In Flask, the default HTTP method is also GET. If you want to use POST, you must also specify it in your app.route() and in your function.

Label Element:

- The <label> tag defines a label for many form elements.
  I.e. The <label> element represents a caption for an item in a user interface.
- The <label> element is useful for screen-reader users, because the screen-reader will read out load the label when the user is focused on the input element.
- The <label> element also helps users who have difficulty clicking on very small regions, such as radio buttons or checkboxes, because when the user clicks the text within the <label> element, it toggles the radio button/checkbox.
- To associate the <label> with an <input> element, you need to give the <input> an id attribute. The <label> then needs a for attribute whose value is the same as the input's id. Alternatively, you can nest the <input> directly inside the <label>, in which case the for and id attributes are not needed because the association is implicit.
- General syntax:
  <label> … </label>
- E.g. You can use the label tag like such:
  1.
  <label for="FirstName"> Enter your first name: </label>
  <input type="text" id="FirstName" name="first name">
  2.
  <label> Enter your first name:
  <input type="text" name="first name">
  </label>

Name Attribute:

- Each input field, explained below, must have a name attribute to be submitted.
- If the name attribute is omitted, the data of that input field will not be sent at all.

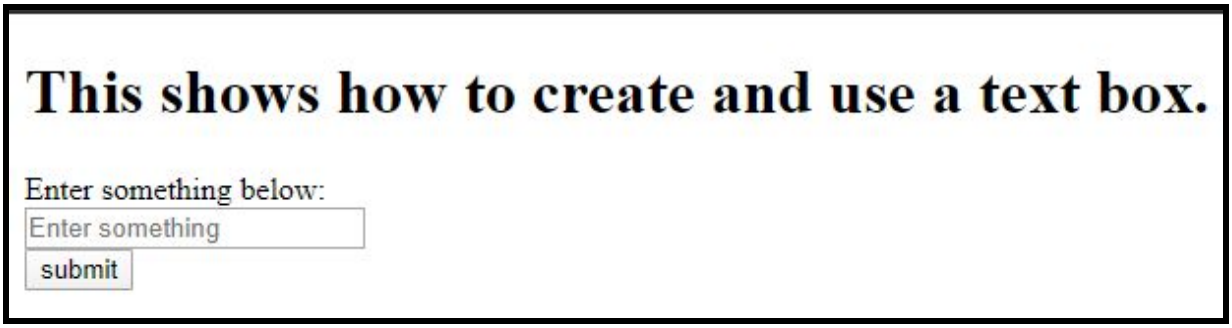
Input Element:

- **Text:**
  - General syntax:
    <input type="text">
    Note: There is no closing tag for input.
  - It defines a single-line text input field.

- E.g. Consider the code below:

```
<!DOCTYPE html>
<html>
<head>
    <title> Text </title>
</head>
<body>
    <h1> This shows how to create and use a text box. </h1>

    <!-- Uses the HTTP GET request as no specific request was stated. -->
    <form action="/">

        <!-- The value of the for attribute in label must match the id attribute attribute of input. -->
        <label for="text"> Enter something below: </label> <br>
        <input type="text" id="text" name="text" placeholder="Enter something"> <br>

        <!-- Creates a submit button. -->
        <input type="submit" value="submit">
    </form>
</body>
</html>
```

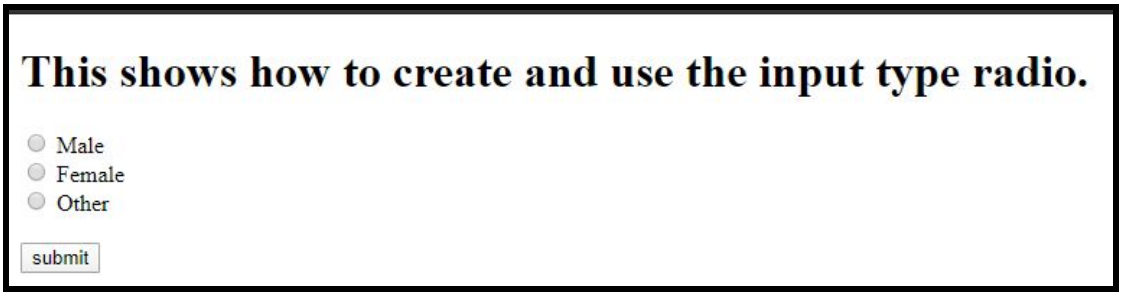
This is what the HTML page looks like:



- **Radio:**
  - General syntax:
    <input type="radio">
  - It defines a radio button. Radio buttons let a user select ONLY ONE of a limited number of choices.
  - E.g. Consider the code below:

```
<!DOCTYPE html>
<html>
<head>
    <title> Radio </title>
</head>
<body>
    <h1> This shows how to create and use the input type radio. </h1>
    <!-- Uses the HTTP GET request as no specific request was stated. -->
    <form action="/">

        <!-- The value of the for attribute in label must match the id attribute attribute of input. -->
        <input type="radio" id="male" name="gender" value="male">
        <label for="male">Male</label><br>

        <input type="radio" id="female" name="gender" value="female">
        <label for="female">Female</label><br>

        <input type="radio" id="other" name="gender" value="other">
        <label for="other">Other</label> <br> <br>

        <!-- Creates a submit button. -->
        <input type="submit" value="submit">
    </form>
</body>
</html>
```

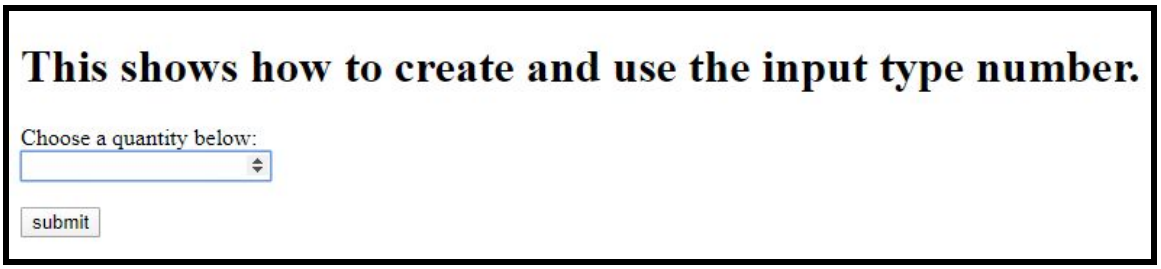
This is what the HTML page looks like:



- **Number:**
  - General syntax:
    <input type="number">
  - It defines a numeric input field.
  - E.g. Consider the code below:

```
<!DOCTYPE html>
<html>
<head>
    <title> Number </title>
</head>
<body>
    <h1> This shows how to create and use the input type number. </h1>
    <!-- Uses the HTTP GET request as no specific request was stated. -->
    <form action="/">

        <!-- The value of the for attribute in label must match the id attribute attribute of input. -->
        <label for="quantity">Choose a quantity below:</label><br>
        <input type="number" id="quantity" name="quantity"><br><br>

        <!-- Creates a submit button. -->
        <input type="submit" value="submit">
    </form>
</body>
</html>
```

This is what the HTML page looks like:

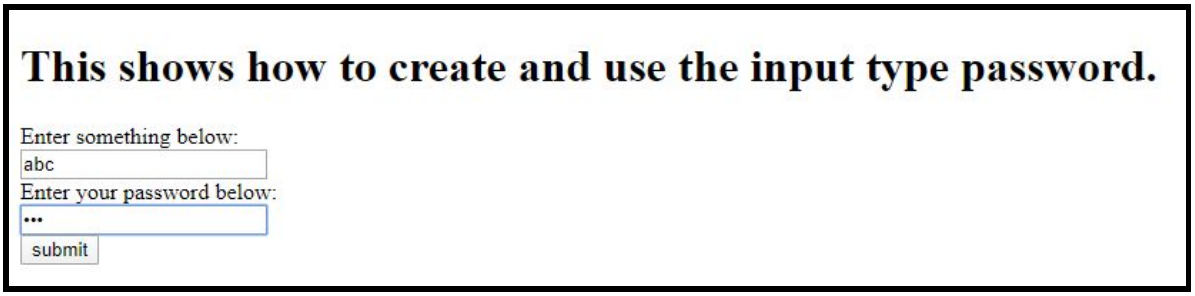


- **Password:**
  - General syntax:
    <input type="password">
  - It defines a password field.
  - Note: The characters in a password field are masked (shown as asterisks or circles).

- E.g. Consider the code below:

```
<!DOCTYPE html>
<html>
<head>
    <title> Password </title>
</head>
<body>
    <h1> This shows how to create and use the input type password. </h1>
    <!-- Uses the HTTP GET request as no specific request was stated. -->
    <form action="/">

        <!-- The value of the for attribute in label must match the id attribute attribute of input. -->
        <label for="text"> Enter something below: </label> <br>
        <input type="text" id="text" name="text" placeholder="Enter something"> <br>

        <label for="password"> Enter your password below: </label> <br>
        <input type="password" id="password" name="password" placeholder="Password"> <br>

        <!-- Creates a submit button. -->
        <input type="submit" value="submit">
    </form>
</body>
</html>
```

This is what the HTML page looks like:



- **Submit:**
  - General syntax:
    <input type="submit">
  - It defines a button for submitting form data to a form-handler.
  - The form-handler is typically a server page with a script for processing input data.
  - The form-handler is specified in the form's action attribute.

Input Attributes:

- **Maxlength:**
  - The input maxlength attribute specifies the maximum number of characters allowed in an input field.
  - **Note:** When a maxlength is set, the input field will not accept more than the specified number of characters. However, this attribute does not provide any feedback. So, if you want to alert the user, you must write JavaScript code.
- **Min and Max:**
  - The input min and max attributes specify the minimum and maximum values for an input field.
  - The min and max attributes work with the following input types: number, range, date, datetime-local, month, time and week.
  - **Note:** Use the max and min attributes together to create a range of legal values.
- **Pattern and Title:**
  - The input pattern attribute specifies a regular expression that the input field's value is checked against, when the form is submitted.

    - The pattern attribute works with the following input types: text, date, search, url, tel, email, and password.
    - **Note:** Use the title attribute to describe the pattern to help the user.
    - The title attribute specifies extra information about an element.
    - The information is most often shown as a tooltip text when the mouse moves over the element.
- **Placeholder:**
    - The input placeholder attribute specifies a short hint that describes the expected value of an input field (a sample value or a short description of the expected format).
    - The short hint is displayed in the input field before the user enters a value.
    - The placeholder attribute works with the following input types: text, search, url, tel, email, and password.
- **Required:**
    - The input required attribute specifies that an input field must be filled out before submitting the form.
    - The required attribute works with the following input types: text, search, url, tel, email, password, date pickers, number, checkbox, radio, and file.
- **Size:**
    - The input size attribute specifies the visible width, in characters, of an input field.
    - The default value for size is 20.
    - **Note:** The size attribute works with the following input types: text, search, tel, url, email, and password.
- **Value:**
    - The value attribute in HTML is used to specify the value of the element with which it is used. It has different meanings for different HTML elements. It can be used with the following elements: <input>, <button>, <meter>, <li>, <option> <progress> and <param>. The input value attribute specifies an initial value for an input field.

Select Element:

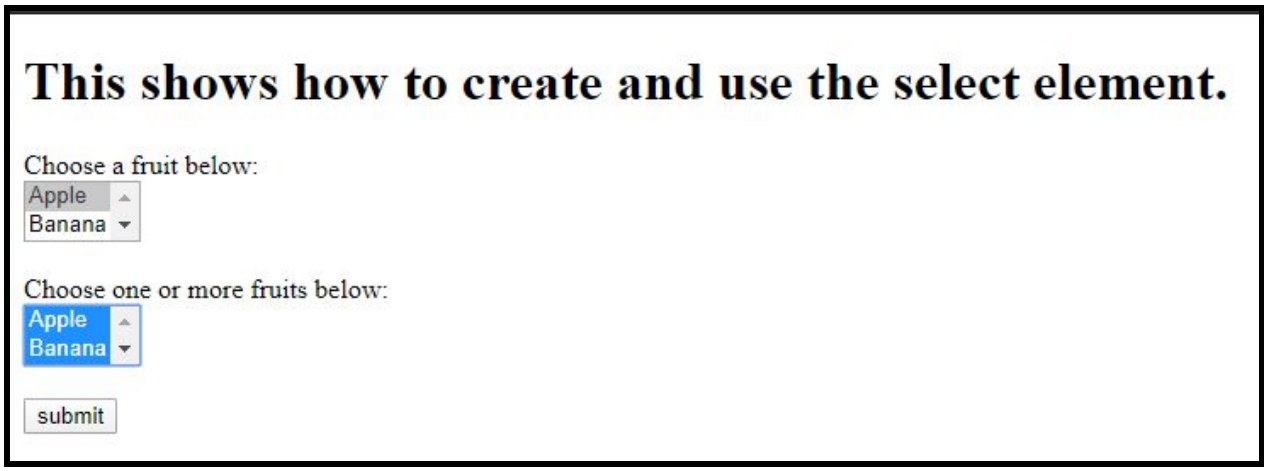
- The <select> element defines a drop-down list.
- The <option> element defines an option that can be selected.
- By default, the first item in the drop-down list is selected.
- To define a pre-selected option, add the selected attribute to the option.
- You can use the size attribute to specify the number of visible values.
- You can use the multiple attribute to allow the user to select more than one value.
- General syntax:

```
<select>
        <option> Option 1 </option>
        <option> Option 2 </option>
        ...
</select>
```

- E.g. Consider the code below:

```
<!DOCTYPE html>
<html>
<head>
    <title> Select </title>
</head>
<body>
    <h1> This shows how to create and use the select element. </h1>
    <!-- Uses the HTTP GET request as no specific request was stated. -->
    <form action="/">

        <!-- The value of the for attribute in label must match the id attribute attribute of select. -->
        <!-- Here, you can only choose one of the items. -->
        <label for="fruits"> Choose a fruit below: </label> <br>
         <select id="fruits" name="fruit" size="2">
            <option value="apple">Apple</option>
            <option value="banana">Banana</option>
            <option value="grape">Grape</option>
            <option value="orange">Orange</option>
         </select><br><br>

         <!-- Here, you can only more than one of the items. -->
         <!-- To select multiple options, hold down the Ctrl (windows) / Command (Mac) button. -->
        <label for="fruits"> Choose one or more fruits below: </label> <br>
         <select id="fruits" name="fruit" size="2" multiple>
            <option value="apple">Apple</option>
            <option value="banana">Banana</option>
            <option value="grape">Grape</option>
            <option value="orange">Orange</option>
         </select><br><br>

        <!-- Creates a submit button. -->
        <input type="submit" value="submit">
    </form>
</body>
</html>
```

This is what the HTML page looks like:



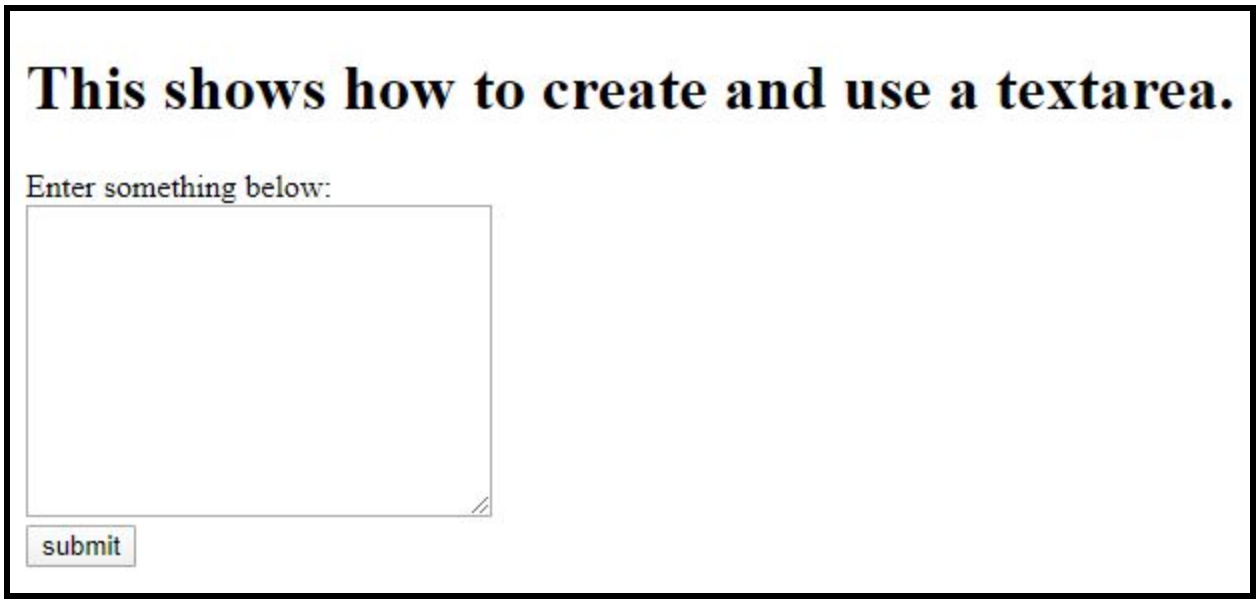
Textarea Element:

- General syntax:
  <textarea>
  …
  </textarea>
- The <textarea> element defines a multi-line input field.
- The rows attribute specifies the visible number of lines in a text area.
- The cols attribute specifies the visible width of a text area.

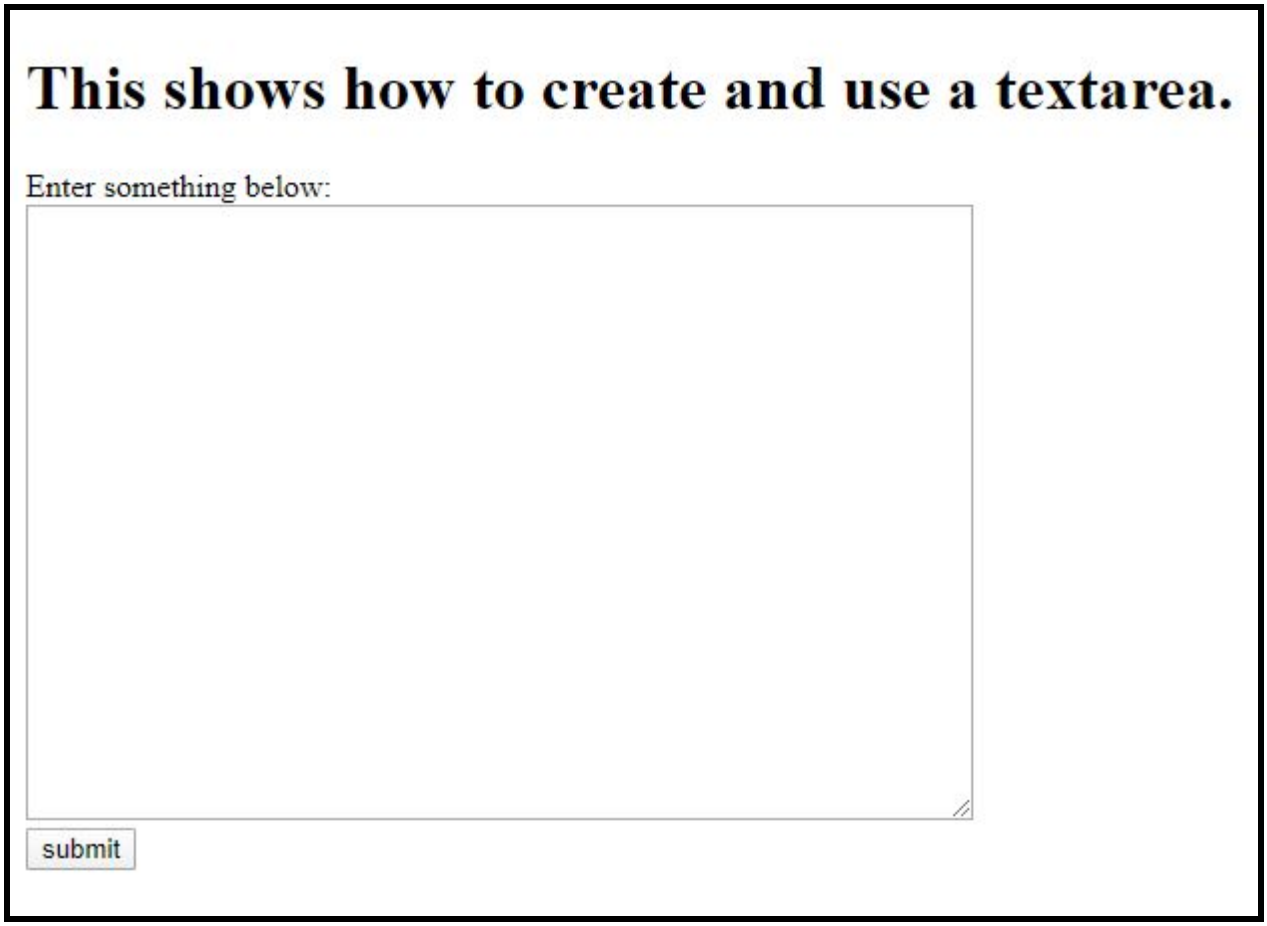
- Note: By default, the size of the textarea box is changeable. You can use CSS to make it so that the size of the textarea box cannot be changed.
  E.g. Consider the code below:

```
<!DOCTYPE html>
<html>
<head>
    <title> Textarea </title>
</head>
<body>
    <h1> This shows how to create and use a textarea box. </h1>
    <!-- Uses the HTTP GET request as no specific request was stated. -->
    <form action="/">

        <!-- The value of the for attribute in label must match the id attribute attribute of input. -->
        <label for="textarea"> Enter something below: </label> <br>
        <textarea id="textarea" name="text" placeholder="Enter something" rows="10" cols="30">

        </textarea><br>

        <!-- Creates a submit button. -->
        <input type="submit" value="submit">
    </form>
</body>
</html>
```

This is what the HTML page looks like:



The user can drag the 2 lines on the bottom right corner to change the size of the textarea box, as such:



Lecture Notes:

- **Files** and **HTTP** are protocols. A **file** means that it's a file on the user's home computer and not on the internet. So, say that a person has a file called index.html on their home computer. When they open index.html with a browser, the browser knows that index.html is a file and so it doesn't need to go on the internet to search for index.html.
- HTTP means that the browser needs to go onto the internet to find the HTTP file. The browser goes to a DNS to get the IP address of the given url. To make a url available on the internet, we need to use a web server.
- E.g.
  This is a file:
  

  This is HTTP:
  

- To use flask in Python, you need to import it using the statement: **from flask import Flask**
  flask is a module/package in Python that contains several related files, one of which is Flask.
- The difference between HTTP and HTTPS is that HTTPS is that HTTPS is more secure. With HTTPS, any information you send using the get request will be encrypted, where as with HTTP, the information will not be encrypted.
- When using Flask, make sure that all your method/function names are different.
- Flask looks at all the different routing names to determine which to choose. It doesn't do top to bottom.
  E.g. Consider the below code snippet:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def func1():
        return "Hello World"

@app.route('/<name>')
def func2(name):
        return "Hello {0)".format(name)

@app.route('/rick')
def func3():
        return "Hello Rick"

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

  If the user starts a web server, runs this python program and then types in http://localhost:5000/rick, the program will run func3 as opposed to func2. This is because there is a route called /rick.

- **Note:** __name__ is a built-in variable which evaluates to the name of the current module. Furthermore, because there is no main() function in Python, when the command to run a python program is given to the interpreter, the code that is at level 0 indentation is to be executed. However, before doing that, it will define a few special variables. __name__ is one such special variable. If the source file is executed as the main program, the interpreter sets the __name__ variable to have a value "__main__". If this file is being imported from another module, __name__ will be set to the module's name. I.e. __name__ == "__main__" acts as the main() function if you are running that program directly. If you are running a program via another program, __name__ == "__main__" will not work.
  E.g. Consider these 2 python programs:

File1.py File2.py

```
1  print("File1 __name__ = %s" %__name__)
2
3  if __name__ == "__main__":
4      print("File1 is being run directly")
5  else:
6      print("File1 is being imported")
```

File1.py File2.py

```
1  import File1
2
3  print("File2 __name__ = %s" %__name__)
4
5  if __name__ == "__main__":
6      print("File2 is being run directly")
7  else:
8      print("File2 is being imported")
```

If I run File1.py, I get this output:

```
File1 __name__ = __main__
File1 is being run directly
```

If I run File2.py, I get this output:

```
File1 __name__ = File1
File1 is being imported
File2 __name__ = __main__
File2 is being run directly
```

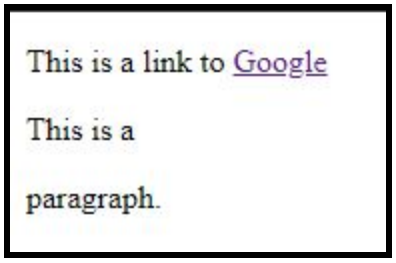
HTML:

- HTML stands for HyperText Markup Language. It is not a programming language. It is the markup language for creating web pages and is the building block of the web.
- All HTML files must end with the .html extension.
- index.html is the root or home page of a website.

- In general, HTML elements can be divided into two categories: block level and inline elements.
- **Inline elements** do not start a new line and take only the necessary width. Inline elements are those which only occupy the space bounded by the tags defining the element. They are usually within other HTML elements. Examples of inline elements are <img>, <a>.
- **Block-level elements** always start on a new line and take up the full width of a page, from left to right. A block-level element can take up one line or multiple lines and has a line break before and after the element. It can contain other block level elements as well as inline elements. Block level elements create larger structures than inline elements. Examples of block level elements are , <h1> - <h6> and .
- E.g. Consider the HTML code below:

```
<!DOCTYPE html>
<html>
<body>
    <p> This is a link to <a href="https://www.google.com"> Google </a> </p>
    <p> This is a <p> paragraph. </p> </p>
</body>
</html>
```

It looks like this on the browser:



Notice how for the line ** This is a link to Google ** the <a> tag doesn't start on a new line, but for the line ** This is a paragraph. ** the inner tag starts on a new line.

- All HTML tags can have **attributes** and attributes provide more information about an element. Attributes are always placed within the start tag. Some attributes are necessary, such as the href attribute for the <a> tag, while others are optional. Attributes are always formatted as key/value pairs. I.e. attribute="something"
  E.g. In the case of ** Link to Google **, href is the key and "https://www.google.com" is the value.
- All Web pages share a common structure:

```
<!DOCTYPE html>
<HTML>

  <HEAD>
    <TITLE> … </TITLE>
  </HEAD>

  <BODY>
  …
```

</BODY>

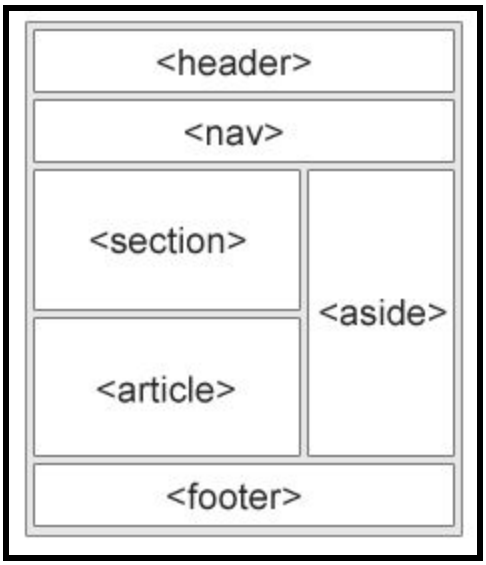
</HTML>

- All Web pages should contain a pair of <HTML>, <HEAD>, <TITLE>, and <BODY> tags.
- The **<TITLE>** tag is required in all HTML documents and it defines the title of the document. The <title> tag 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.
- The **class** attribute is used to define equal styles for elements with the same class name. This way, all HTML elements with the same class attribute will get the same style. The class attribute can be used on any HTML element. The class name is case sensitive. Different tags can have the same class name. A class cannot start with a number.
- The ** ** tag defines a division or a section in an HTML document. The element is often used as a container for other HTML elements to style them with CSS or to perform certain tasks with JavaScript.
- The **id** attribute specifies a unique id for an HTML element (the value must be unique within the HTML document). The id attribute can be used on any HTML element. The id value is case-sensitive. The id value must contain at least one character, and must not contain spaces. Furthermore, an id cannot start with a number. The difference between the id attribute and the class attribute is that an HTML element can only have one unique id that belongs to that single element, while a class name can be used by multiple elements.
  I.e. There can not be multiple of the same ids while there can be multiple of the same classes.
  HTML bookmarks are used to allow readers to jump to specific parts of a Web page. Bookmarks can be useful if your webpage is very long. To make a bookmark, you must first create the bookmark, and then add a link to it. When the link is clicked, the page will scroll to the location with the bookmark.
  E.g.
  1. First, create a bookmark with the id attribute:
     <h2 id="C4">Chapter 4</h2>
  2. Then, add a link to the bookmark ("Jump to Chapter 4"), from within the same page:
     Jump to Chapter 4
  3. Or, add a link to the bookmark ("Jump to Chapter 4"), from another page:
     Jump to Chapter 4

Semantic Elements:

- A **semantic element** clearly describes its meaning to both the browser and the developer.
- Examples of non-semantic elements: and . Tells nothing about its content.
- Examples of semantic elements: <form>, <table>, and <article>. Clearly defines its content.
- In HTML there are some semantic elements that can be used to define different parts of a web page:
  - <article>
  - <aside>
  - <details>

- <figcaption>
- <figure>
- <footer>
- <header>
- <main>
- <mark>
- <nav>
- <section>
- <summary>
- <time>



- Here's a table that describes what each semantic element does:

Tag	**Description**
<article>	Defines an article
<aside>	Defines content aside from the page content
<details>	Defines additional details that the user can view or hide
<figcaption>	Defines a caption for a <figure> element
<figure>	Specifies self-contained content, like illustrations, diagrams, photos, code listings, etc.
<footer>	Defines a footer for a document or section
<header>	Specifies a header for a document or section
<main>	Specifies the main content of a document
<mark>	Defines marked/highlighted text

<nav>	Defines navigation links
<section>	Defines a section in a document
<summary>	Defines a visible heading for a <details> element
<time>	Defines a date/time

CSS:

- CSS stands for **Cascading Style Sheets**. It is a styling language that describes the presentation of HTML.
- In layman's terms, HTML provides the structure and the content of a website whereas CSS describes the look and feel of the site. One odd misconception that comes up once in a while is that some people think that HTML and CSS are the same thing. HTML and CSS are separate languages however they rely on one another and are thus often linked together in conjunction.
- **Note:** CSS properties must be spelled exactly the way they are listed. Like HTML, CSS will not complain or throw any errors if it is incorrect and your browser will not give any indication as to what is wrong with your code.
- A comment in CSS starts with /* and ends with */.
  E.g.

```
p {
  color: red;  /* Set text color to red */
}
```

- The main way of including CSS alongside an HTML file is by serving a separate CSS file to the HTML file of which your browser will read and serve accordingly. This is done by providing a <link> tag inside the <head> tag of your HTML code. This looks as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css">       <!-- css file linked here -->
    <title>My Website</title>
</head>
<body>
    ...
</body>
```

  The <link> tag indicates that you are associating a separate file alongside the given HTML file. The rel attribute indicates that this file is a stylesheet or in other words a CSS file. The href attribute refers to the hyperlink reference and is the path to the stylesheet file so that the HTML file can find it. When normally using CSS, the location of the CSS file does not matter. However when using Flask, the stylesheet must be in a folder called **static** for Flask to use it properly.
- Using the style attribute inside an HTML tag is called an **inline style**. Although this functionality is supported by HTML5, it is not recommended as it makes your code less modular and less maintainable over time. Suppose you had many tags in your code with inline styles and one day you wished to change them. If all of your tags have inline styles, it would be a nightmare to go back through your code and change everything. As

a general rule of thumb, avoid inline styles wherever possible and also opt to include styles in an external CSS file. This is an example of inline style:

```
<div style="background-color: green">
   this is a div with green background
</div>
```

- A CSS rule-set consists of a selector and a declaration block. Each declaration block contains a property and a value such that the general form for a declaration block is property:value. The general syntax for css is:

```
selector{
declaration1;
declaration2;
…
declaration(n);
}
```

- E.g.

```
h1{
color:blue;
font-size:12px;
}
```

In this example, h1 is the selector, color:blue is the first declaration, and font-size:12px is the second declaration. In the first declaration, color is the property and blue is the value. In the second declaration, font-size is the property and 12px is the value. It means that all <h1> elements will be in the colour blue and have a font size of 12px.

- E.g.

```
p {
color:red;
text-align:center;
}
```

In this example, p is the selector, color:red is the first declaration with color being the property and red being the value and text-align:center is the second declaration with text-align being the property and center being the value. It means that all <p> elements will be center-aligned and with a red text color.

- CSS works by associating various style attributes to certain elements of HTML code. CSS can be bound to the following HTML elements:
    1. **Tags:**
        - The element selector selects HTML elements based on the element name.
        - E.g. Here, all <p> elements on the page will be center-aligned, with a red text color:

```
p {
  text-align: center;
  color: red;
}
```

    2. **Classes:**
        - The class selector selects HTML elements with a specific class attribute.
        - To select elements with a specific class, write a period (.) character, followed by the class name.

- E.g. In this example all HTML elements with class="center" will be red and center-aligned:

```
.center {
  text-align: center;
  color: red;
}
```

- You can also specify that only specific HTML elements should be affected by a class.
- E.g. In this example only <p> elements with class="center" will be center-aligned:

```
p.center {
  text-align: center;
  color: red;
}
```

3. **Ids:**
   - The id selector uses the id attribute of an HTML element to select a specific element.
   - The id of an element is unique within a page, so the id selector is used to select one unique element!
   - To select an element with a specific id, write a hash ,#, character, followed by the id of the element.
   - E.g. The CSS rule below will be applied to the HTML element with id="para1":

```
#para1 {
  text-align: center;
  color: red;
}
```

4. **Everything:**
   - The universal selector ,*, selects all HTML elements on the page.
   - This is usually used for global styles such as a font size, font family, font color.
   - E.g. The CSS rule below will affect every HTML element on the page:

```
* {
  text-align: center;
  color: blue;
}
```

5. **CSS Grouping Selector:**
   - The grouping selector selects all the HTML elements with the same style definitions. To group selectors, separate each selector with a comma.
   - E.g. In the following CSS code the h1, h2, and p elements have the same style definitions:

```
h1 {
  text-align: center;
  color: red;
}
h2 {
  text-align: center;
```

```
  color: red;
}
p {
  text-align: center;
  color: red;
}
```

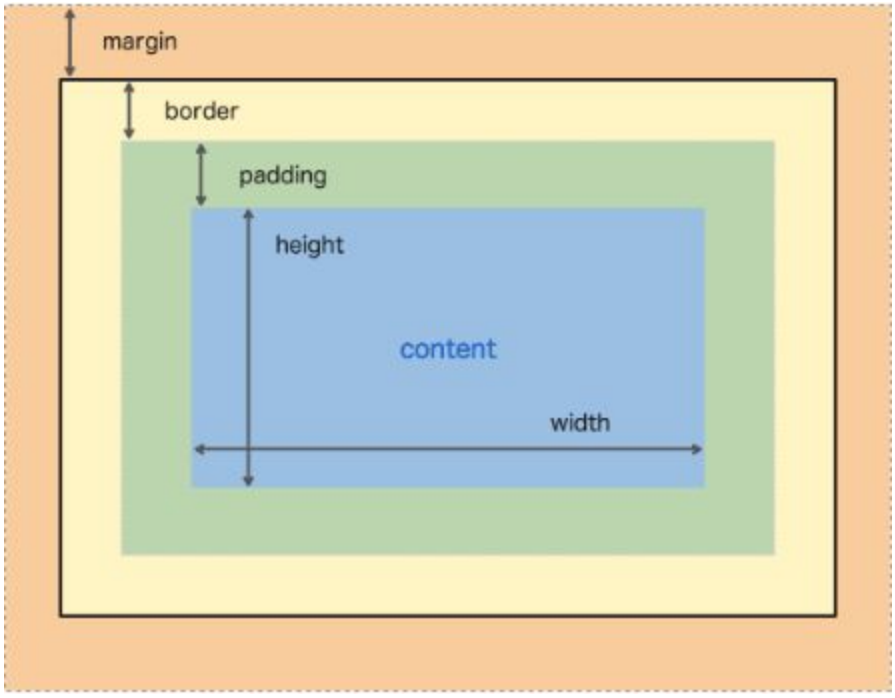
It will be better to group the selectors, to minimize the code.

We can group the selectors from the code above as such:

```
h1, h2, p {
  text-align: center;
  color: red;
}
```

The CSS Box Model:

- Every element in HTML is made of a box that wraps around the element. Each box is made up of four properties as shown below:



- Properties:
  - **Margin:** The area outside of the border. This area is transparent.
  - **Border:** The area on the outskirts of the content.
  - **Padding:** The area between the border and the content; this area is transparent.
  - **Content:** The area where your actual content goes (e.g. text, images, other divs, etc).
- **Common CSS Properties:**
  Here is a table of common CSS properties

Property	Description	Values
background-color	Sets the background color of an element	color-rgb, color-hex, color-name, transparent

border	Sets the border around an element	border-width, border-style, border-color
visibility	Sets if an element should be visible or not	visible, hidden collapse
float	Sets where an image or a text will appear in another element	left, right, none
width, height	Sets dimensions of an element	none, length, %
font-family	Sets the font type of an element. In CSS, there are two types of font family names: 1. Generic family: A group of font families with a similar look (E.g. "Serif", "Monospace") 2. Font family: A specific font family (E.g. "Times New Roman", "Arial") You can have more than one font in a font-family property. The font-family property should hold several font names as a fallback system. If the browser does not support the first font, it tries the next font, and so on. Start with the font you want, and end with a generic family, to let the browser pick a similar font in the generic family, if no other fonts are available. If the name of a font family is more than one word, it must be in quotation marks, like "Times New Roman".	Arial, Times New Roman, Serif.
font-size	Sets the size of the font	%, px
font-weight	Sets how bold font should be	normal, bold, bolder, 100, 200
margin	Sets the space outside of an element’s borders	auto, length, %
padding	Sets the space inside of an element’s borders	auto, length, %
color	Sets the color of text (not the color of the background!)	color-rgb, color-hex, color-name, transparent
text-align	Sets the alignment of text	left, right, center, justify
text-decoration	Adds decorations to text (note: bold is not considered a decoration)	underline, strikethrough, blink
text-transform	Controls the letters of an element	uppercase, lowercase, capitalize, none
cursor	Sets the type of cursor to be displayed	default, pointer, crosshair, move

HTTP:

- The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers.
- HTTP works as a request-response protocol between a client and server.
- Communication between client computers and web servers is done by sending HTTP Requests and receiving HTTP Responses like such:
    - A client (a browser) sends an HTTP request to the web.
    - A web server receives the request.
    - The server runs an application to process the request.
    - The server returns an HTTP response (output) to the browser.
    - The client (the browser) receives the response.
- A web browser may be the client, and an application on a computer that hosts a web site may be the server.
- Example: A client (browser) submits an HTTP request to the server. Then the server returns a response to the client. The response contains status information about the request and may also contain the requested content.
- **Note:** HTTP does not encrypt data but HTTPS does.

HTTP Methods:

- Some of the HTTP methods are:
    - GET
    - POST
    - PUT
    - HEAD
    - DELETE
    - PATCH
    - OPTIONS
- The two most common HTTP methods are GET and POST.
- **GET Method:**
    - The GET method is used to retrieve information from the given server using a given URL.
    - GET is used to request data from a specified resource. It should not modify the data in any way.
    - GET requests can be cached.
    - GET requests remain in the browser history.
    - GET requests can be bookmarked.
    - GET requests should never be used when dealing with sensitive data since the query is visible.
    - GET requests have length restrictions.
    - Note that the query string (name/value pairs) is sent in the URL of a GET request.
      E.g. /test/demo_form.php?name1=value1&name2=value2
- **POST Method:**
    - POST is used to send data to a server to create/update a resource.
    - A POST request is used to send data to the server, such as customer information or file upload using HTML forms.

- The data sent to the server with POST is stored in the request body of the HTTP request.
  E.g.
  POST /test/demo_form.php HTTP/1.1
  Host: w3schools.com
  name1=value1&name2=value2
- POST requests are never cached.
- POST requests do not remain in the browser history.
- POST requests cannot be bookmarked.
- POST requests have no restrictions on data length.

Tags:

<!-- -->:

- This is a comment in HTML.
- Syntax: **<!-- … -->**

<!DOCTYPE HTML>:

- The <!DOCTYPE HTML> declaration must be the very first thing in your HTML document, before the <html> tag.
- It is an instruction to the web browser about what version of HTML the page is written in.

<A>:

- The <A> tag defines a hyperlink, which is used to link from one page to another.
- A link has three parts: a destination, a label, and a target. The destination specifies the address of the web page or file the user will access when he/she clicks on the link. The label is the text that will appear underlined or highlighted on the page.
- The href attribute specifies the URL of the page the link goes to.
  I.e. The href attribute specifies the destination.
- Syntax: **<A href=destination> label </A>**
- The LINK, VLINK, and ALINK attributes can be inserted in the <BODY> tag to define the color of a link.
- LINK defines the color of links that have not been visited.
  Syntax: **<body link="color">**
- VLINK defines the color of links that have already been visited.
  Syntax: **<body vlink="color">**
- ALINK specifies the color of an active link in a document (a link is activated when it is clicked).
  Syntax: **<body alink="color">**
- **Note:** By default, links will appear as follows in all browsers:
  - An unvisited link is underlined and blue.
  - A visited link is underlined and purple.
  - An active link is underlined and red.
- To create a link to an email address, do the following:
  <A href="mailto:email_address"> Label </A>
- Anchors enable a user to jump to a specific place on a website. Two steps are necessary to create an anchor. First you must create the anchor itself. Then you must create a link to the anchor from another point in the document.
- To create the anchor, do the following at the point in the webpage where you want the user to jump to:
  <A name="anchor name"> Label </A>
  The name attribute specifies the name of an anchor.
  To create the link, do the following at the point in the text where you want the link to appear:
  <A href="#anchor name"> Label </A>

<ABBR>:

- The <abbr> tag defines an abbreviation or an acronym.
- An abbreviation and an acronym are both shortened versions of something else.
- Syntax: **<abbr title="Full Name"> Abbreviation </abbr>**

<ARTICLE>:

- The <article> tag specifies independent, self-contained content.
- An article should make sense on its own and it should be possible to distribute it independently from the rest of the site.
- Potential sources for the <article> element include:
  - Forum post
  - Blog post
  - News story
  - Comment
- Syntax:
  <ARTICLE>
  …
  </ARTICLE>

<ASIDE>:

- The <aside> tag defines some content aside from the content it is placed in.
- The aside content should be related to the surrounding content.
- Syntax:
  <ASIDE>
  …
  </ASIDE>

<B>:

- Makes the text bold.
- Syntax: **<B> … </B>**

<BASEFONT>:

- Specifies a default text-color, font-size, or font-family for all the text in a document.
- **Note:** There is no closing tag.
- The color attribute specifies the color of the text inside a <basefont> element.
- The face attribute specifies the font of the text inside a <basefont> element.
- The size attribute specifies the size of the text inside a <basefont> element.
- Syntax with color attribute: **<BASEFONT COLOR = 'color'>**
- Syntax with face attribute: **<BASEFONT FACE = "font name">**
- Syntax with size attribute: **<BASEFONT SIZE = n>**
  Note that n is a number from 1 to 7.
- These attributes can be combined to change the font, size, and color of the text all at once by doing: **<BASEFONT SIZE=n COLOR= 'color' FACE='font name'>**

<BLOCKQUOTE>:

- The <blockquote> tag specifies a section that is quoted from another source.
- Browsers usually indent <blockquote> elements.
- Syntax:
  <BLOCKQUOTE cite="url">
  ….
  </BLOCKQUOTE>

<BODY>:

- The <body> tag defines the document's body.
- The <body> element contains all the contents of an HTML document, such as text, hyperlinks, images, tables, lists, etc.
- Syntax:

  <body>

  …

  </body>

**
:**

- Inserts a single line break.
- **Note:** There is no closing tag.
- Syntax: **
**

<DETAILS>:

- The <details> tag specifies additional details that the user can view or hide on demand.
- The <details> tag can be used to create an interactive widget that the user can open and close. Any sort of content can be put inside the <details> tag.
- The content of a <details> element should not be visible unless the open attribute is set or the <summary> tag is used.
- The <summary> tag defines a visible heading for the <details> element. The heading can be clicked to view/hide the details.
- Syntax:

  <details>

  <summary> … </summary>

  …

  </details>

<DIV>:

- The tag defines a division or a section in an HTML document.
- The element is often used as a container for other HTML elements to style them with CSS or to perform certain tasks with JavaScript.
- Syntax: ** … **

<FIGURE>:

- The <figure> tag specifies self-contained content, like illustrations, diagrams, photos, code listings, etc.
- While the content of the <figure> element is related to the main flow, its position is independent of the main flow, and if removed it should not affect the flow of the document.
- The <figcaption> tag defines a caption for a <figure> element.
- The <figcaption> element can be placed as the first or last child of the <figure> element.
- Syntax:

  <figure>

  <figcaption> … </figcaption>

  …

  </figure>

<FONT>:

- Specifies the font size, font face and color of text.
- The color attribute specifies the color of the text inside a <font> element.
- The face attribute specifies the font of the text inside a <font> element.
- The size attribute specifies the size of the text inside a <font> element.

- Syntax with color attribute: **<FONT COLOR = 'color'> … </FONT>**
- Syntax with face attribute: **<FONT FACE = "font name"> … </FONT>**
- Syntax with size attribute: **<FONT SIZE = n> … </FONT>**
  Note that n is a number from 1 to 7.
- These attributes can be combined to change the font, size, and color of the text all at once by doing: **<FONT SIZE=n COLOR= 'color' FACE='font name'> … </FONT>**

<FOOTER>:

- The <footer> tag defines a footer for a document or section.
- A <footer> element should contain information about its containing element.
- A <footer> element typically contains:
  - authorship information
  - copyright information
  - contact information
  - sitemap
  - back to top links
  - related documents
- You can have several <footer> elements in one document.
- Syntax:
  <footer>
  …
  </footer>

<FORM>:

- An HTML form is an area of the document that allows users to enter information into fields.
- A form may be used to collect personal information, opinions in polls, user preferences and other kinds of information.
- There are two basic components of a web form: **the shell** which is the part that the user fills out and **the script** which processes the information.
- The shell has three important parts:
  1. The <FORM> tag, which includes the address of the script which will process the form.
  2. The form elements, like text boxes and radio buttons.
  3. The submit button which triggers the script to send the entered information to the server
- **Creating a form shell:**
  - Type **<FORM> METHOD=POST ACTION=url>**
  - Create the form elements.
  - End with a closing </FORM> tag.

  The url specifies where to send the form-data when the form is submitted.
  The POST request method requests that a web server accepts the data enclosed in the body of the request message, most likely for storing it.
- **Creating text boxes:**
  - To create a text box, type **<INPUT TYPE="text" NAME="name" VALUE="value" SIZE="n" MAXLENGTH="n">**
  - The <input> tag specifies an input field where the user can enter data.
  - The NAME, VALUE, SIZE, and MAXLENGTH attributes are optional.
  - The NAME attribute is used to identify the text box to the processing script.

  - The VALUE attribute is used to specify the text that will initially appear in the text box.
  - The SIZE attribute is used to define the size of the box in characters.
  - The MAXLENGTH attribute is used to define the maximum number of characters that can be typed in the box.
- **Creating larger text areas:**
  - To create larger text areas, type **<TEXTAREA NAME="name" ROWS=n1 COLS=n2 WRAP> Default Text </TEXTAREA>**
  - n1 is the height of the text box in rows and n2 is the width of the text box in characters.
  - The WRAP attribute causes the cursor to move automatically to the next line as the user types.
- **Creating radio buttons:**
  - To create a radio button, type **<INPUT TYPE="radio" NAME="name" VALUE="Data">Label**
  - The <input> tag specifies an input field where the user can enter data.
  - "Data" is the text that will be sent to the server if the button is checked.
  - "Label" is the text that identifies the button to the user.
- **Creating checkboxes:**
  - To create a checkbox, type **<INPUT TYPE="checkbox" NAME="name" VALUE="value">Label**
  - **Note:** If you give a group of radio buttons or checkboxes the same name, the user will only be able to select one button or box at a time.
  - The <input> tag specifies an input field where the user can enter data.
- **Creating drop-down menus:**
  - To create a drop-down menu, type
    <SELECT NAME="name" SIZE=n MULTIPLE>
    <OPTION VALUE= "value1">Label1
    <OPTION VALUE= "value2">Label2
    ...
    </SELECT>
  - The SIZE attribute specifies the height of the menu in lines. This is optional.
  - MULTIPLE allows users to select more than one menu option. This is optional.
  - The <select> element is used to create a drop-down list.
  - The <option> tags inside the <select> element define the available options in the list.
- **Creating a Submit Button:**
  - To create a submit button, type **<INPUT TYPE="submit" VALUE="NAME">**
  - The <input> tag specifies an input field where the user can enter data.
- **Creating a Reset Button:**
  - To create a reset button, type **<INPUT TYPE="reset" VALUE="name">**
  - The <input> tag specifies an input field where the user can enter data.

<H1> - <H6>:

- Defines a heading.
- <H1> is the biggest size while <H6> is the smallest size.
- Syntax: **<Hi> … </Hi>** where i is an integer between 1 to 6, inclusive.

<HEAD>:

- The <head> element is a container for all the head elements.
- The <head> element can include a title for the document, scripts, styles, meta information, and more.
- Syntax:
  <head>
  …
  </head>

<HEADER>:

- The <header> element represents a container for introductory content or a set of navigational links.
- A <header> element typically contains:
  - one or more heading elements (<h1> - <h6>).
  - Logo or icon.
  - Authorship information
- You can have several <header> elements in one document.
- **Note:** A <header> tag cannot be placed within a <footer>, <address> or another <header> element.
- **Note:** A <header> tag is not the same as a <head> tag.
- Syntax:
  <header>
  …
  </header>

<HTML>:

- The <html> tag tells the browser that this is an HTML document.
- The <html> tag represents the root of an HTML document.
- The <html> tag is the container for all other HTML elements (except for the <!DOCTYPE> tag).
- Syntax:
  <HTML>
  …
  </HTML>

<I>:

- Italicize the text.
- Syntax: **<I> … </I>**

<IMG>:

- The <img> tag defines an image in an HTML page.
- The <img> tag has two required attributes: src and alt.
- **Note:** The <img> tag has no closing tag.
- **Note:** To link an image to another document, simply nest the <img> tag inside <a> tags.
- The SRC attribute specifies the URL of an image.
- The ALT attribute specifies an alternate text for an image. Some browsers don't support images. In this case, the ALT attribute can be used to create text that appears instead of the image.
- The height attribute specifies the height of an image.
- The width attribute specifies the width of an image.
- The border attribute specifies the width of the border around an image.
- Syntax: **<img src="url" alt="name" height="n" width="m">**

<LI>

- The <LI> tag defines a list.
- There are 2 types of lists:
    1. Ordered:
        a. Ordered lists are a list of numbered items. An ordered list starts with the <OL> tag. Each list item starts with the <LI> tag. The list items will be marked with numbers by default.
        b. Syntax:
        <OL>
        <LI> … </LI>
        <LI> … </LI>
        <LI> … </LI>
        </OL>
        c. The TYPE attribute allows you to change the kind of symbol that appears in the list.
            i. “A” is for capital letters.
            ii. “a” is for lowercase letters.
            iii. “I” is for capital roman numerals.
            iv. “i” is for lowercase roman numerals.
            v. “1” is for numbers. This is the default value.
    2. Unordered:
        a. An unordered list is a list of bulleted items.
        b. It is denoted with the <UL> tag. Each list item starts with the <LI> tag.
        c. The list items will be marked with bullets by default.
        d. Syntax:
        <UL>
        <LI> … </LI>
        <LI> … </LI>
        <LI> … </LI>
        </UL>
        e. The TYPE attribute allows you to change the type of bullet that appears.
            i. circle corresponds to an empty round bullet.
            ii. square corresponds to a square bullet.
            iii. disc corresponds to a solid round bullet. This is the default value.

<MAIN>:

- The <main> tag specifies the main content of a document.
- The content inside the <main> element should be unique to the document. It should not contain any content that is repeated across documents such as sidebars, navigation links, copyright information, site logos, and search forms.
- **Note:** There must not be more than one <main> element in a document.
- **Note:** The <main> element must NOT be a descendant of an <article>, <aside>, <footer>, <header>, or <nav> element.
- Syntax:
<main>
…
</main>

<MARK>:

- The <mark> tag defines marked text.
- Use the <mark> tag if you want to highlight parts of your text.
- Syntax: **<mark> … </mark>**

<NAV>:

- The <nav> tag defines a set of navigation links.
- Note that not all links of a document should be inside a <nav> element.
- The <nav> element is intended only for major blocks of navigation links.
- Syntax:
  <nav>
  <a href=...>
  <a href=...>
  …
  </nav>

<P>:

- Defines a paragraph.
- Syntax: **<P> … </P>**

<SECTION>:

- The <section> tag defines sections in a document, such as chapters, headers, footers, or any other sections of the document.
- Syntax:
  <section>
  …
  </section>

<TABLE>:

- The <TABLE> tag is used to create a table.
- The <TR> tag defines the beginning of a row.
- The <TD> tag defines the beginning of a cell that contains data. These cells are called **standard cells**.
- The <TH> tag defines the beginning of a cell that contains the headers. These cells are called **header cells**.
- **Adding a Border:**
- The BORDER=n attribute allows you to add a border n pixels thick around the table.
- To make a solid border color, use the BORDERCOLOR="color" attribute.
- To make a shaded colored border, use BODERCOLORDARK="color" and BORDERCOLORLIGHT="color".
- **Adjusting the Width:**
- When a web browser displays a table, it often adds extra space. To eliminate this space use the WIDTH = n attribute in the <TABLE> and <TD> tags. Note that a cell cannot be smaller than its contents, and if you make a table wider than the browser window, users will not be able to see parts of it.
- **Centering a Table:**
- There are two ways to center a table:
  1. **<TABLE ALIGN=CENTER>**
  2. Enclose the <TABLE> tags between the opening and closing <CENTER> tags.
- **Wrapping Text around a Table:**
- It is possible to wrap text around a table. This technique is often used to keep images and captions together within an article.

- To wrap text around a table, type **<TABLE ALIGN = LEFT>** to align the table to the left while the text flows to the right.
- **Adding Space around a Table:**
- To add space around a table, use the HSPACE=n and VSPACE=n attributes in the <TABLE> tag.
- The hspace attribute specifies the amount of whitespace on the left and right side of an object.
- The vspace attribute specifies the amount of whitespace above and below an object.
- **Spanning Cells Across Columns and Rows:**
- To span a cell across many columns, type **<TD COLSPAN=n>**, where n is the number of columns to be spanned.
- To span a cell across many rows, type **<TD ROWSPAN=n>**, where n is the number of rows to be spanned.
- **Aligning Cell Content:**
- By default, a cell's content is aligned horizontally to the left and vertically in the middle.
- Use VALIGN=direction to change the vertical alignment, where "direction" is top, middle, bottom, or baseline.
- Use ALIGN=direction to change the horizontal alignment where "direction" is left, center, or right.
- **Controlling Cell Spacing:**
- **Cell spacing** is the space between cells while **cell padding** is the space around the contents of a cell.
- To control both types of spacing, use the CELLSPACING=n and CELLPADDING=n attributes in the <TABLE> tag.
- **Changing a Cell's Color:**
- To change a cell's color, add the BGCOLOR="color" attribute to the <TD> tag.
- **Dividing Your Table into Column Groups:**
- You can divide your table into two kinds of column groups: structural and nonstructural.
- **Structural column groups** control where dividing lines are drawn while **non-structural groups** do not.
- To create structural column groups, type **<COLGROUP SPAN=n>** after the <TABLE> tag, where n is the number of columns in the group.
- To create non-structural column groups, type **<COL SPAN=n>**, where n is the number of columns in the group.
- **Dividing Table into Horizontal Sections:**
- You can also create a horizontal section consisting of one or more rows. This allows you to format the rows all at once.
- To create a horizontal section, type <THEAD>, <TBODY>, or <TFOOT> before the first <TR> tag of the section.
- The <thead> tag is used to group header content in an HTML table.
- The <thead> element is used in conjunction with the <tbody> and <tfoot> elements to specify each part of a table (header, body, footer).
- Browsers can use these elements to enable scrolling of the table body independently of the header and footer. Also, when printing a large table that spans multiple pages, these elements can enable the table header and footer to be printed at the top and bottom of each page.

- The <thead> tag must be used in the following context: As a child of a <table> element, after any <caption>, and <colgroup> elements, and before any <tbody>, <tfoot>, and <tr> elements.
- The <tbody> tag is used to group the body content in an HTML table.
- The <tbody> tag must be used in the following context: As a child of a <table> element, after any <caption>, <colgroup>, and <thead> elements.
- The <tfoot> tag is used to group footer content in an HTML table.
- The <tfoot> tag must be used in the following context: As a child of a <table> element, after any <caption>, <colgroup>, <thead>, and <tbody> elements.
- **Controlling Line Breaks:**
- Unless you specify otherwise a browser will divide the lines in a cell as it sees fit.
- The NOWRAP attribute placed within the <TD> tag forces the browser to keep all the text in a cell on one line.

<TIME>:

- The <time> tag defines a human-readable date/time.
- This element can also be used to encode dates and times in a machine-readable way so that user agents can offer to add birthday reminders or scheduled events to the user's calendar, and search engines can produce smarter search results.
- The time element does not render as anything special in any of the major browsers.
- Syntax: **<time datetime="year-month-day hour:minute"> … </time>**

<TITLE>:

- The <TITLE> tag is required in all HTML documents and it defines the title of the document.
- It 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.
- Syntax: **<TITLE> … </TITLE>**

<U>:

- Underlines the text.
- Syntax: **<U> … </U>**

Attributes:

Align:

- The align attribute specifies the alignment of the text within a paragraph.
- There are 4 values for the align attribute:
    1. Left: Left aligns the text.
    2. Right: Right aligns the text.
    3. Center: Centers the text.
    4. Justify: Stretches the lines so that each line has equal width.
- The ALIGN attribute can be inserted in the <P> and header tags.
- Syntax: **<P align='left|right|center|justify'> … </P>**
- For example, **<H1 align=CENTER> The New York Times </H1>** would create a centered heading of the largest size.

BGCOLOR:

- Used to define the background color.
- Is in the <BODY> tag.
- Syntax: **<BODY BGCOLOR="color">**

Class:

- Used to define equal styles for elements with the same class name. This way, all HTML elements with the same class attribute will get the same style.
- The class attribute can be used on any HTML element.
- The class name is case sensitive. Different tags can have the same class name. A class cannot start with a number.

ID:

- Specifies a unique id for an HTML element (the value must be unique within the HTML document).
- The id attribute can be used on any HTML element.
- The id value is case-sensitive. The id value must contain at least one character, and must not contain spaces. Furthermore, an id cannot start with a number.
- The difference between the id attribute and the class attribute is that an HTML element can only have one unique id that belongs to that single element, while a class name can be used by multiple elements.
  I.e. There can not be multiple of the same ids while there can be multiple of the same classes.

TEXT:

- Used to define the text color.
- Is in the <BODY> tag.
- Syntax: **<BODY TEXT="color">**

Flexbox:

- **Flexbox** is a layout model that allows elements to align and distribute space within a container. Using flexible widths and heights, elements can be aligned to fill a space or distribute space between elements, which makes it a great tool to use for responsive design systems.
- Oftentimes we want to align elements on a page side by side as opposed to just vertically down. This is a bit of a challenge at times without an effective strategy to help us. The **flexbox strategy** is a method of aligning and distributing items in a container even if their size is unknown and/or dynamic. It is one of several strategies of laying out div content on an HTML page. Another popular one is the grid strategy.
- The main idea behind the flex layout is to give the container the ability to change its items' width, height, and order in order to fill the available space. A flex container expands items to fill the remaining free space or conversely shrink them to prevent them from overflowing from the container itself.
- From now on, when you plan to align items, you will use the following strategy:
    - Create a flex container.
    - Add properties to the flex container to satisfy your alignment requirements.
    - Create flex items inside the container.
- The flexbox strategy makes use of two components:
    1. **Flex container:** The div surrounding the items to be aligned.
    2. **Flex items:** The items to be aligned.
- To setup a flexbox, we can set up a container div with the CSS property flex as follows:

```
.container {
   display: flex;
}
```

- Some properties of flex container are:
    1. **flex-direction:**
        - The flex-direction property defines in which direction the container wants to stack the flex items.
        - The row value stacks the flex items horizontally from left to right.
        - The row-reverse value stacks the flex items horizontally but from right to left.
        - The column value stacks the flex items vertically from top to bottom.
        - The column-reverse value stacks the flex items vertically but from bottom to top.
        - E.g.

```
.container {
   flex-direction: row;          /* default order (left to right) */
   flex-direction: row-reverse;    /* right to left order */
   flex-direction: column;       /* top to bottom order */
   flex-direction: column-reverse; /* bottom to top order */
}
```

        - Example with flex-direction row:

```
.flex-container {
  display: flex;
  flex-direction: row;
}
```

- Example with flex-direction row-reverse:

```
.flex-container {
  display: flex;
  flex-direction: row-reverse;
}
```

- Example with flex-direction column:

```
.flex-container {
  display: flex;
  flex-direction: column;
}
```

- Example with flex-direction column-reverse:

```
.flex-container {
  display: flex;
  flex-direction: column-reverse;
}
```

2. **flex-wrap:**
   - The flex-wrap property specifies that the flex items will wrap if necessary.
   - The wrap value specifies that the flex items will wrap if necessary.
   - The nowrap value specifies that the flex items will not wrap. This is default.
   - The wrap-reverse value specifies that the flexible items will wrap if necessary, in reverse order.
   - E.g.

```
.container {
    flex-wrap: wrap;     /* wrap onto multiple lines from top to bottom */
    flex-wrap: nowrap;  /* default (all items on the same line */
    /* wrap onto multiple lines from bottom to top */
    flex-wrap: wrap-reverse;
}
```

   - Example with wrap:

```
.flex-container {
  display: flex;
  flex-wrap: wrap;
}
```

   - Example with nowrap:

```
.flex-container {
  display: flex;
  flex-wrap: nowrap;
}
```

   - Example with wrap-reverse:

```
.flex-container {
  display: flex;
  flex-wrap: wrap-reverse;
}
```

3. **flex-flow:**
   - The flex-flow property is a shorthand property for setting both the flex-direction and flex-wrap properties.

- Example with flex-flow:

```
.flex-container {
  display: flex;
  flex-flow: row wrap;
}
```

4. **justify-content:**
   - The justify-content property is used to align the flex items horizontally.
   - It defines the alignment along the main axis. It helps distribute extra free space leftover when either all the flex items on a line are inflexible, or are flexible but have reached their maximum size. It also exerts some control over the alignment of items when they overflow the line.
   - The center value aligns the flex items at the center of the container.
   - E.g.

```
.flex-container {
  display: flex;
  justify-content: center;
}
```

   - The flex-start value aligns the flex items at the beginning of the container. This is default.
   - E.g.

```
.flex-container {
  display: flex;
  justify-content: flex-start;
}
```

   - The flex-end value aligns the flex items at the end of the container.
   - E.g.

```
.flex-container {
  display: flex;
  justify-content: flex-end;
}
```

   - The space-around value displays the flex items with space before, between, and after the lines.
   - E.g.

```
.flex-container {
  display: flex;
  justify-content: space-around;
}
```

   - The space-between value displays the flex items with space between the lines.
   - E.g.

```
.flex-container {
  display: flex;
  justify-content: space-between;
}
```

5. **align-items:**
   - The align-items property is used to align the flex items vertically.
   - The align-items property defines the default behavior for how items are laid out along the cross axis, which is perpendicular to the main axis.

- You can think of align-items as the justify-content version for the cross-axis.
- The center value aligns the flex items in the middle of the container.
- E.g.

```
.flex-container {
  display: flex;
  height: 200px;
  align-items: center;
}
```

- The flex-start value aligns the flex items at the top of the container.
- E.g.

```
.flex-container {
  display: flex;
  height: 200px;
  align-items: flex-start;
}
```

- The flex-end value aligns the flex items at the bottom of the container.
- E.g.

```
.flex-container {
  display: flex;
  height: 200px;
  align-items: flex-end;
}
```

- The stretch value stretches the flex items to fill the container. This is default.
- E.g.

```
.flex-container {
  display: flex;
  height: 200px;
  align-items: stretch;
}
```

- The baseline value aligns the flex items such as their baselines aligns.
- E.g.

```
.flex-container {
  display: flex;
  height: 200px;
  align-items: baseline;
}
```

6. align-content:

- The align-content property is used to align the flex lines.
- It helps to align a flex container's lines within it when there is extra space in the cross-axis, similar to how justify-content aligns individual items within the main-axis.
- **Note:** This property has no effect when the flexbox has only a single line.
- The space-between value displays the flex lines with equal space between them.

- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: space-between;
}
```
- The space-around value displays the flex lines with space before, between, and after them.
- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: space-around;
}
```
- The stretch value stretches the flex lines to take up the remaining space. This is default.
- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: stretch;
}
```
- The center value displays the flex lines in the middle of the container.
- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: center;
}
```
- The flex-start value displays the flex lines at the start of the container.
- E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: flex-start;
}
```
- The flex-end value displays the flex lines at the end of the container. E.g.
```
.flex-container {
  display: flex;
  height: 600px;
  flex-wrap: wrap;
  align-content: flex-end;}
```

- Some properties of flex item are:
  1. **order:**
     - The order property specifies the order of the flex items. By default, they are ordered in the order in which they appear.
     - The order value must be a number. The default value is 0.
     - E.g.

```
<div class="flex-container">
  <div style="order: 3">1</div>
  <div style="order: 2">2</div>
  <div style="order: 4">3</div>
  <div style="order: 1">4</div>
</div>
```

  2. **flex-grow:**
     - The flex-grow property specifies how much a flex item will grow relative to the rest of the flex items.
     - By default, all items have flex-grow set to 0.
     - The value must be a number. The default value is 0.
     - E.g. This makes the third flex item grow eight times faster than the other flex items.

```
<div class="flex-container">
  <div style="flex-grow: 1">1</div>
  <div style="flex-grow: 1">2</div>
  <div style="flex-grow: 8">3</div>
</div>
```

  3. **flex-shrink:**
     - The flex-shrink property specifies how much a flex item will shrink relative to the rest of the flex items.
     - The value must be a number. The default value is 1.
     - E.g. This does not let the third flex item shrink as much as the other flex items.

```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="flex-shrink: 0">3</div>
  <div>4</div>
  <div>5</div>
  <div>6</div>
  <div>7</div>
  <div>8</div>
  <div>9</div>
  <div>10</div>
</div>
```

  4. **flex-basis:**
     - The flex-basis property specifies the initial length of a flex item.
     - E.g. This sets the initial length of the third flex item to 200 pixels.

```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
```

```
  <div style="flex-basis: 200px">3</div>
  <div>4</div>
</div>
```

5. **flex:**
   - The flex property is a shorthand property for the flex-grow, flex-shrink, and flex-basis properties.
   - E.g. This makes the third flex item not growable (0), not shrinkable (0), and with an initial length of 200 pixels.

```
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="flex: 0 0 200px">3</div>
  <div>4</div>
</div>
```

6. **align-self:**
   - The align-self property specifies the alignment for the selected item inside the flexible container.
   - The align-self property overrides the default alignment set by the container's align-items property.
   - E.g. This aligns the third flex item in the middle of the container.

```
<div class="flex-container">
  <div>1</div>
  <div style="align-self: flex-start">2</div>
  <div style="align-self: flex-end">3</div>
  <div>4</div>
</div>
```

JavaScript:

- Javascript is an object oriented programming language that is used to make web pages interactive.
- It's an interpreted language and runs on the client's computer/browser.
- With respect to the other 3 web-development languages:
  1. HTML defines the content of a web page. It defines the information that we see on the page.
  2. CSS defines the style of a web page. It defines what certain things are supposed to look like.
  3. Javascript defines the functionality of a web page. It defines what happens when we click on things or input data.
- Javascript can be included in HTML code using the <script></script> tag. Similarly to CSS, it can be included using either inline Javascript or using external Javascript.
- Here's an example of inline Javascript:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
  <p>This is some content</p>
  <script>
```

```
    // we can write our javascript code here as we wish
    console.log('keviniscool');
  </script>
</body>
</html>
```

- Here's an example of external javascript:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
  <p>This is some content</p>
  <script src="script.js"></script>
</body>
</html>
```

- There are two main differences with including Javascript when compared to including CSS:
    1. We include both inline and external Javascript using the <script></script> tag.
    2. Javascript is included at the bottom of the file as opposed to the <head> tag.

  The second point is a very important difference between the two. Unlike CSS, Javascript needs to be added to the page after the HTML and CSS have loaded. If you do not do this, depending on the speed of your browser, the functionality of the site may load before you can actually see the page's components and you may get some unexpected results. CSS on the other hand, does not face this problem because even if it loads in first, it just waits for the HTML to finish loading to apply the styles. There is no consequence in the CSS loading before elements have fully loaded.
- **Frontend vs Backend:**
- The difference between JS and Flask is that Flask is used for the backend. Javascript is only meant to handle client interactions, it is not supposed to logically process data. Javascript is intentionally limited for security reasons.
- Example: Javascript cannot be used if we want to send data back to a server or retrieve data from a database.
- Example: If you were making a game, the Javascript is only there to process that a character wants to move. It does not process whether or not it was legal or not for that character to move
- Example: Javascript cannot read or write files while Flask can.
- **Variables:**
- Variables are defined using one of three keywords, **var**, **let** and **const**, and the = operator:
- E.g.
    1. var x = 420
    2. let x = 420
    3. const x = 420
- **var** and **let** have minor differences in terms of scoping. It is recommended to use let over var. However, in the past it was common to use var.

- **const** defines a constant variable that will never change after being instantiated. You cannot change a const's value after it has been instantiated otherwise Javascript will throw a TypeError.
- Similarly to Python, Javascript is dynamically typed meaning that you do not have to indicate the type of variable when you create it. Unlike Python, you have to prefix any variable you create with either var, let, or const to indicate that it is a variable.
- **Equality:**
- There are two main ways of checking for equality in Javascript. We can use either the == operator or the === operator. However, the == operator will sometimes produce odd results. **Therefore, it is recommended that you only use the === operator and to never use the == operator.**
- **Comments:**
- Comments are denoted as **//** or **/*...*/**.
- Single line comments start with //.
- Multi-line comments start with /* and end with */.
- **Operators:**
- A table of the different arithmetic operators:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

- A table of the different assignment operators:

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y

/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

- A table of the different comparison operators:

Operator	**Description**
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

- A table of the different logical operators:

Operator	**Description**
&&	logical and
||	logical or
!	logical not

- A table of the different type operators:

Operator	**Description**
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

- **If statements:**
- Syntax:

```
if (condition 1) {
...
}
else if (condition 2){
```

```
…
}
else{
…
}
```

- **Note:** You must have the if block and only 1 if block. You can have as many else if blocks as you want, and you can have at most 1 else block.
- **While Loops:**
- The **while loop** loops through a block of code as long as a specified condition is true.
- Syntax:

```
while (condition) {
  ...
}
```

- E.g.

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

- The do/while loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested.
- Syntax:

```
do{
  …
}
while (condition);
```

- E.g.

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

- **For Loops:**
- There are 3 types of for loops:
    1. **for:** Loops through a block of code a number of times.
        - Syntax:

```
for (statement 1; statement 2; statement 3) {
  ...
}
```

        - Statement 1 is executed one time before the execution of the code block.
        - Statement 2 defines the condition for executing the code block.
        - Statement 3 is executed every time after the code block has been executed.
        - E.g.

```
for (i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

2. **for/in:** Loops through the properties of an object.
    - E.g.
      ```
      var person = {fname:"John", lname:"Doe", age:25};
      var text = "";
      var x;
      for (x in person) {
        text += person[x];
      }
      ```
3. **for/of:** Loops through the values of an iterable object. for/of lets you loop over data structures that are iterable such as Arrays, Strings, Maps, NodeLists, and more.
    - Syntax:
      ```
      for (variable of iterable) {
        ...
      }
      ```
    - **Variable:** For every iteration the value of the next property is assigned to the variable. Variable can be declared with const, let, or var.
    - **Iterable:** An object that has iterable properties.
    - E.g.
      ```
      var cars = ['BMW', 'Volvo', 'Mini'];
      var x;

      for (x of cars) {
        document.write(x + "<br >");
      }
      ```

- **Arrays:**
- JavaScript arrays are used to store multiple values in a single variable.
- An **array** is a special variable, which can hold more than one value at a time.
- Javascript arrays do not have to contain elements of the same type.
- Syntax: **var variable_name = [item1, item2, …, itemn];**
- E.g. **var cars = ["Saab", "Volvo", "BMW"];**
- You access an array element by referring to the index number. All arrays start at index 0.
- E.g. **var name = cars[0];**
- **Objects:**
- An object in Javascript is similar to a dictionary in Python. Unlike in Python, the keys of the object must be String but the value can be any valid type. This type of structure is called a **JSON or Javascript Object Notation**.
- E.g.
  ```
  myObject = {
    "firstName": "kevin",
    "lastName": "zhang",
    "age": 69,
    "cool": true
  }
  ```
- The name:values pairs in JavaScript objects are called properties.
- You can access object properties in two ways:
    1. **objectName.propertyName**
    2. **objectName["propertyName"]**

- Objects can also have methods.
- Methods are actions that can be performed on objects.
- Methods are stored in properties as function definitions.
- E.g.

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id     : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

- You access an object method with the following syntax:

```
objectName.methodName()
```

- E.g.

```
name = person.fullName();
```

- **Functions:**
- A JavaScript function is a block of code designed to perform a particular task.
- A JavaScript function is executed when something invokes it. There are 3 ways we can invoke a function:
    1. When an event occurs (when a user clicks a button)
    2. When it is invoked (called) from JavaScript code
    3. Automatically (self invoked)
- You declare a function by prefixing it with the function keyword.
- Syntax:

```
function function_name(...){
…
}
```

- E.g.

```
function myFunction(p1, p2) {
  return p1 * p2;   // The function returns the product of p1 and p2
}
```

- Variables declared within a JavaScript function, become local to the function. **Local variables** can only be accessed from within the function. Since local variables are only recognized inside their functions, variables with the same name can be used in different functions. Local variables are created when a function starts, and deleted when the function is completed.
- **Print Statements:**
- JavaScript can output data in different ways:
    - Writing into an HTML element, using **innerHTML**.
    - Writing into the HTML output using **document.write()**.
    - Writing into an alert box, using **window.alert()**.
    - Writing into the browser console, using **console.log()**.
      Note: You can view console output in Chrome by pressing CMD+OPTION+J on a Mac or CTRL+SHIFT+J on Windows.
- Using INNERHTML:
    - To access an HTML element, JavaScript can use the document.getElementById(id) method. This is the most common way of obtaining

an element to modify. Since IDs in HTML are guaranteed to be unique, we can access a pre-existing element in an HTML form using this strategy.

- The id attribute defines the HTML element. The innerHTML property defines the HTML content.
- E.g.

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My First Paragraph</p>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>

</body>
</html>
```

- E.g.

HTML (index.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
  <!-- we will not see the text here because it will be overwritten
   by the Javascript -->
  <p id="my-content">This is the original content</p>
  <script src="script.js"></script>
</body>
</html>
```

Javascript (script.js):

```
document.getElementById('my-content').innerHTML = 'keviniscool';
```

- **Note:** There is a similar command called **document.getElementsByClassName()** however since classes are not unique, this command will always return the result in an array.

- Using document.write():
  - For testing purposes, it is convenient to use document.write().
  - **Note:** Using document.write() after an HTML document is loaded, will delete all existing HTML.
  - E.g.

```
<!DOCTYPE html>
<html>
<body>
```

```
<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<button type="button" onclick="document.write(5 + 6)">Try it</button>

</body>
</html>
```

- Using window.alert():
  - You can use an alert box to display data.
  - E.g.

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
window.alert(5 + 6);
</script>
</body>
</html>
```

- Using console.log()
  - For debugging purposes, you can use the console.log() method to display data.
  - E.g.

```
<!DOCTYPE html>
<html>
<body>
<script>
console.log(5 + 6);
</script>
</body>
</html>
```

- **Events:**
- An **HTML event** can be something the browser does or something a user does.
- Here are some examples of HTML events:
  - An HTML web page has finished loading
  - An HTML input field was changed
  - An HTML button was clicked
- Often, when events happen, you may want to do something. JavaScript lets you execute code when events are detected.
- HTML allows event handler attributes, along with JavaScript code, to be added to HTML elements.
- A table of HTML event:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element

onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

- Event handlers can be used to handle, and verify, user input, user actions, and browser actions:
    - Things that should be done every time a page loads.
    - Things that should be done when the page is closed.
    - Action that should be performed when a user clicks a button.
    - Content that should be verified when a user inputs data.
- Many different methods can be used to let JavaScript work with events:
    - HTML event attributes can execute JavaScript code directly.
    - HTML event attributes can call JavaScript functions.
    - You can assign your own event handler functions to HTML elements.
    - You can prevent events from being sent or being handled.
- E.g. In the past labs, we've made forms that don't do anything. Using Javascript, we can retrieve input from forms and perform interactive behaviour with the input we've collected. The most common way of doing this is by accessing the .value attribute of an element and reacting to an event such as onclick().

HTML (index.html):

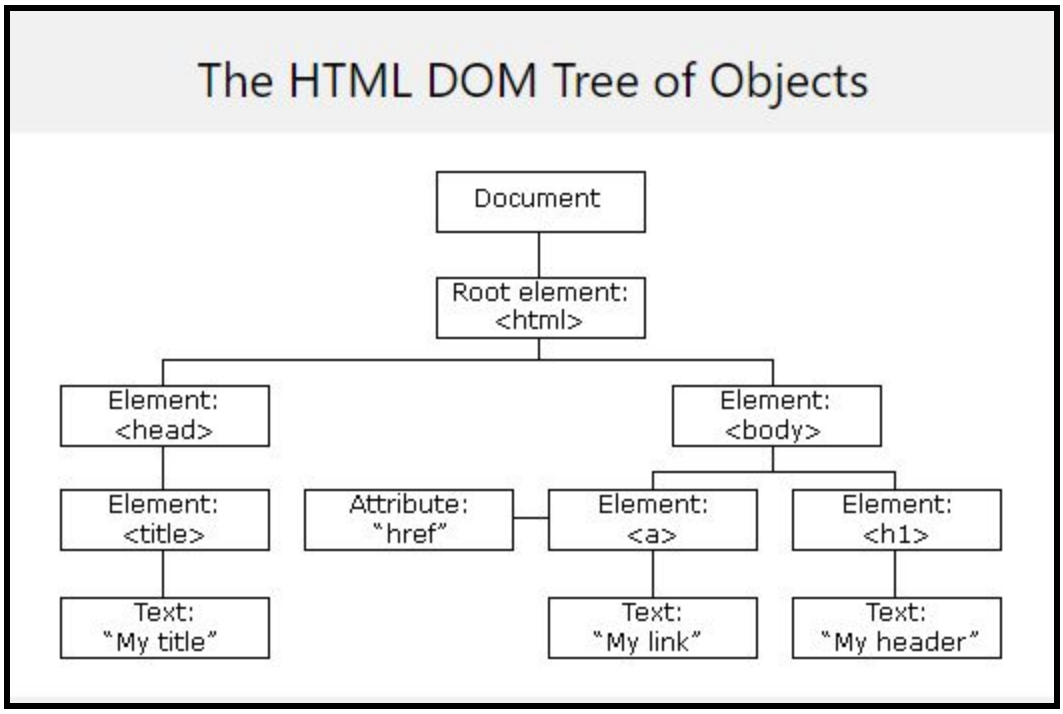
```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
   <label>What is your name?</label>
   <input type="text" id="name-form" placeholder="Enter your name">
   <button type="submit" onclick="nameResult()">Submit</button>
   <!-- we put a placeholder <p> here so that it can be written into
   later -->
   <!-- using document.write() would overwrite the entire body -->
   <p id="result"></p>
   <script src="script.js"></script>
</body>
</html>
```

Javascript (script.js):

```
// this function is called whenever the button is clicked
function nameResult() {
  // get the value inside the form
  name = document.getElementById('name-form').value;
  // add the value
  document.getElementById('result').innerHTML = 'your name is: ' + name;
}
```

DOM:

- With the HTML DOM (Document Object Model), JavaScript can access and change all the elements of an HTML document.
- When a web page is loaded, the browser creates a Document Object Model of the page.
- The HTML DOM document object is the owner of all other objects in your web page. The document object represents your web page.
- If you want to access any element in an HTML page, you always start with accessing the document object.
- The HTML DOM model is constructed as a tree of Objects:



- With the object model, JavaScript gets all the power it needs to create dynamic HTML:
  - JavaScript can change all the HTML elements in the page.
  - JavaScript can change all the HTML attributes in the page.
  - JavaScript can change all the CSS styles in the page.
  - JavaScript can remove existing HTML elements and attributes.
  - JavaScript can add new HTML elements and attributes.
  - JavaScript can react to all existing HTML events in the page.
  - JavaScript can create new HTML events in the page.
- The HTML DOM is a standard object model and programming interface for HTML. It defines:
  - The HTML elements as objects.
  - The properties of all HTML elements.
  - The methods to access all HTML elements.
  - The events for all HTML elements.

  Ie. The HTML DOM is a standard for how to get, change, add, or delete HTML elements.
- The HTML DOM can be accessed with JavaScript and with other programming languages.

  E.g. Consider the document.getElementById(). The “document” part references the DOM.

- In the DOM, all HTML elements are defined as objects.
- A **property** is a value that you can get or set (like changing the content of an HTML element).
- A **method** is an action you can do (like add or deleting an HTML element).
- E.g. Consider the code snippet below:

```
<html>
<body>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>

</body>
</html>
```

This code snippet changes the content (the innerHTML) of the <p> element with id="demo".
In the example above, getElementById is a method, while innerHTML is a property.

- Below are some examples of how you can use the document object to access and manipulate HTML:

1. **Finding HTML Elements:**

Method	**Description**
document.getElementById(id)	Find an element by element id
document.getElementsByTagName(name)	Find elements by tag name
document.getElementsByClassName(name)	Find elements by class name

2. **Changing HTML Elements:**

Property	**Description**
element.innerHTML = new html content	Change the inner HTML of an element
element.attribute = new value	Change the attribute value of an HTML element
element.style.property = new style	Change the style of an HTML element
Method	**Description**
element.setAttribute(attribute, value)	Change the attribute value of an HTML element

3. Adding and Deleting Elements:

Method	Description
document.createElement(element)	Create an HTML element
document.removeChild(element)	Remove an HTML element
document.appendChild(element)	Add an HTML element
document.replaceChild(new, old)	Replace an HTML element
document.write(text)	Write into the HTML output stream

4. Adding Events Handlers:

Method	Description
document.getElementById(id).onclick = function(){code}	Adding event handler code to an onclick event

JQuery:

- JQuery is a lightweight, "write less, do more", JavaScript library.
- The purpose of JQuery is to make it much easier to use JavaScript on your website.
- JQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that you can call with a single line of code.
- JQuery also simplifies a lot of the complicated things from JavaScript, like AJAX calls and DOM manipulation.

Adding JQuery to Your Web Pages:

- There are several ways to start using JQuery on your web site. You can:
  - Download the JQuery library from JQuery.com.
  - Include JQuery from a CDN, Content Delivery Network, like Google.
- You can download JQuery from jQuery.com. The JQuery library is a single JavaScript file, and you reference it with the HTML <script> tag.
  Note: The <script> tag should be inside the <head> section.
  E.g.
  <head>
  <script src="jquery-3.4.1.min.js"></script>
  </head>
- You should place the downloaded file in the same directory as the pages where you wish to use it.
- If you don't want to download and host jQuery yourself, you can include it from a CDN (Content Delivery Network), which is an external link.
- Both Google and Microsoft host jQuery.
- To use jQuery from Google or Microsoft, use one of the following:
- Google:
  <head>
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js">
  </script>

```
</head>
```

- Microsoft:

```
<head>
<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.min.js"></script>
</head>
```

JQuery Syntax:

- With jQuery you select (query) HTML elements and perform "actions" on them.
- The jQuery syntax is tailor-made for selecting HTML elements and performing some action on the element(s).
- Basic syntax: **$(selector).action()**
  - A $ sign to define/access jQuery.
  - A (selector) to "query" HTML elements.
  - A jQuery action() to be performed on the element(s).
- **Note:** jQuery uses CSS syntax to select elements.
- E.g.

  $(this).hide(): hides the current element.
  $("p").hide(): hides all <p> elements.
  $(".test").hide(): hides all elements with class="test".
  $("#test").hide(): hides the element with id="test".

jQuery Selectors:

- jQuery selectors allow you to select and manipulate HTML element(s).
- jQuery selectors are used to select HTML elements based on their name, id, classes, types, attributes, values of attributes and much more. It's based on the existing CSS selectors.
- All selectors in jQuery start with the dollar sign and parentheses: $().
- **The element Selector:**
- The jQuery element selector selects elements based on the element name.
- E.g. You can select all <p> elements on a page like this: **$("p")**
- E.g. When a user clicks on a button, all <p> elements will be hidden:

```
$(document).ready(function(){
  $("button").click(function(){
        $("p").hide();
  });
});
```

- **The #id Selector:**
- The jQuery #id selector uses the id attribute of an HTML tag to find the specific element.
- An id should be unique within a page, so you should use the #id selector when you want to find a single, unique element.
- To find an element with a specific id, write a hash character, followed by the id of the HTML element. E.g. **$("#test")**
- E.g. When a user clicks on a button, the element with id="test" will be hidden:

```
$(document).ready(function(){
  $("button").click(function(){
        $("#test").hide();
  });
});
```

- **The .class Selector:**
- The jQuery .class selector finds elements with a specific class.

- To find elements with a specific class, write a period character, followed by the name of the class. E.g. **$(".test")**
- E.g. When a user clicks on a button, the elements with class="test" will be hidden:

```
$(document).ready(function(){
  $("button").click(function(){
        $(".test").hide();
  });
});
```

- **More Examples of jQuery Selectors:**

Syntax	**Description**
$("*")	Selects all elements
$(this)	Selects the current HTML element
$("p.intro")	Selects all <p> elements with class="intro"
$("p:first")	Selects the first <p> element
$("ul li:first")	Selects the first element of the first
$("ul li:first-child")	Selects the first element of every
$("[href]")	Selects all elements with an href attribute
$("a[target='_blank']")	Selects all <a> elements with a target attribute value equal to "_blank"
$("a[target!='_blank']")	Selects all <a> elements with a target attribute value NOT equal to "_blank"
$(":button")	Selects all <button> elements and <input> elements of type="button"
$("tr:even")	Selects all even <tr> elements
$("tr:odd")	Selects all odd <tr> elements

JQuery Event:

- jQuery is tailor-made to respond to events in an HTML page.
- All the different visitors' actions that a web page can respond to are called **events**.
- An **event** represents the precise moment when something happens.
- Examples of events include:
    - Moving a mouse over an element.
    - Selecting a radio button.
    - Clicking on an element.

- Here are some common DOM events:

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

- Some commonly used JQuery methods include:
  - **click():**
  - The click() method attaches an event handler function to an HTML element.
  - The function is executed when the user clicks on the HTML element.
  - E.g. When a click event fires on a <p> element; hide the current <p> element:

```
$("p").click(function(){
  $(this).hide();
});
```

  - **dblclick():**
  - The dblclick() method attaches an event handler function to an HTML element.
  - The function is executed when the user double-clicks on the HTML element.
  - E.g.

```
$("p").dblclick(function(){
  $(this).hide();
});
```

  - **mouseenter():**
  - The mouseenter() method attaches an event handler function to an HTML element.
  - The function is executed when the mouse pointer enters the HTML element.
  - E.g.

```
$("#p1").mouseenter(function(){
  alert("You entered p1!");
});
```

  - **mouseleave():**
  - The mouseleave() method attaches an event handler function to an HTML element.
  - The function is executed when the mouse pointer leaves the HTML element.
  - E.g.

```
$("#p1").mouseleave(function(){
  alert("Bye! You now leave p1!");
});
```

  - **mousedown():**
  - The mousedown() method attaches an event handler function to an HTML element.
  - The function is executed, when the left, middle or right mouse button is pressed down, while the mouse is over the HTML element.

- E.g.
```
$("#p1").mousedown(function(){
  alert("Mouse down over p1!");
});
```
- **mouseup():**
- The mouseup() method attaches an event handler function to an HTML element.
- The function is executed, when the left, middle or right mouse button is released, while the mouse is over the HTML element.
- E.g.
```
$("#p1").mouseup(function(){
  alert("Mouse up over p1!");
});
```
- **hover():**
- The hover() method takes two functions and is a combination of the mouseenter() and mouseleave() methods.
  Note: hover() is not an actual event. If you use on() with this, nothing happens.
- The first function is executed when the mouse enters the HTML element, and the second function is executed when the mouse leaves the HTML element.
- E.g.
```
$("#p1").hover(function(){
  alert("You entered p1!");
},
function(){
  alert("Bye! You now leave p1!");
});
```
- **focus():**
- The focus() method attaches an event handler function to an HTML form field.
- The function is executed when the form field gets focus.
- E.g.
```
$("input").focus(function(){
  $(this).css("background-color", "#cccccc");
});
```
- **blur():**
- The blur() method attaches an event handler function to an HTML form field.
- The function is executed when the form field loses focus.
- E.g.
```
$("input").blur(function(){
  $(this).css("background-color", "#ffffff");
});
```
- **on():**
- The on() method attaches one or more event handlers for the selected elements.
- E.g. Attach a click event to a <p> element.
```
$("p").on("click", function(){
  $(this).hide();
});
```
- E.g. Attach multiple event handlers to a <p> element.
```
$("p").on({
  mouseenter: function(){
```

```
    $(this).css("background-color", "lightgray");
  },
  mouseleave: function(){
        $(this).css("background-color", "lightblue");
  },
  click: function(){
        $(this).css("background-color", "yellow");
  }
 });
```

- **Document Ready:**
- The document ready event is to prevent any jQuery code from running before the document is finished loading. It is good practice to wait for the document to be fully loaded and ready before working with it. This also allows you to have your JavaScript code before the body of your document, in the head section.
- Here are some examples of actions that can fail if methods are run before the document is fully loaded:
    - Trying to hide an element that is not created yet.
    - Trying to get the size of an image that is not loaded yet.
- Syntax:

```
$(document).ready(function(){
  // jQuery methods go here...
});
```

JavaScript:

- Javascript is an interpreted programming language that is used to make webpages interactive.
- It's object based.
- It runs on the client's browser.
- It uses events and actions to make webpages interactive.
- We can either have an external javascript file and link it to the html file or we can use inline javascript. Always create an external javascript file and link it. Either way, you'd put the javascript (link or code) in the html <script> tag.
  E.g.
  1. Linking to an external javascript file called index.js:
     <script src="index.js"> </script>
  2. Putting inline javascript:
     <script>
     // Some Javascript code.
     </script>
- **Comments:**
- Single line comments are denoted with //.
- Multi-line comments are denoted with /* and */.
- **Equality:**
- There are two main ways of checking for equality in Javascript. We can use either the == operator or the === operator. However, the == operator will sometimes produce odd results. The === operator checks for type equality as well as content. A strict comparison (===) is only true if the operands are of the same type and the contents match. However, (==) converts the operands to the same type before making the comparison. Therefore, it is recommended that you only use the === operator and to never use the == operator.
- E.g.

```
console.log(1 == 1); // expected output: true
console.log('1' == 1); // expected output: true
console.log(1 === 1); // expected output: true
console.log('1' === 1); // expected output: false
```

- **Operators:**
- A table of the different arithmetic operators:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus (Division Remainder)

++	Increment
--	Decrement

- A table of the different assignment operators:

Operator	**Example**	**Same As**
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

- A table of the different comparison operators:

Operator	**Description**
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

- A table of the different logical operators:

Operator	**Description**
&&	logical and
||	logical or

!	logical not

- A table of the different type operators:

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

- **Data Types:**
- Some Javascript data types include numbers, strings, and objects.
- JavaScript has **dynamic types**. This means that the same variable can be used to hold different data types.

1. **String:**

- A string is a series of characters.
- Strings are written with quotes. You can use single or double quotes.
- E.g.

```
var carName1 = "Volvo XC60";   // Using double quotes
var carName2 = 'Volvo XC60';   // Using single quotes
```

- You can use quotes inside a string, as long as they don't match the quotes surrounding the string.
- E.g.

```
var answer1 = "It's alright";           // Single quote inside double quotes
var answer2 = "He is called 'Johnny'";    // Single quotes inside double quotes
var answer3 = 'He is called "Johnny"';    // Double quotes inside single quotes
```

2. **Numbers:**

- JavaScript has only one type of numbers.
- Numbers can be written with, or without decimals.
- E.g.

```
var x1 = 34.00;     // Written with decimals
var x2 = 34;        // Written without decimals
```

- Extra large or extra small numbers can be written with scientific (exponential) notation.
- E.g.

```
var y = 123e5;      // 12300000
var z = 123e-5;     // 0.00123
```

3. **Boolean:**

- Booleans can only have two values: true or false.

4. **Objects:**

- JavaScript objects are written with curly braces {}.
- Object properties are written as name:value pairs, separated by commas.
- E.g.

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

5. **Arrays:**

- JavaScript arrays are written with square brackets.
- Array items are separated by commas. Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

- E.g.
  var cars = ["Saab", "Volvo", "BMW"];

6. Typeof Operator:

- You can use the JavaScript **typeof** operator to find the type of a JavaScript variable. The typeof operator returns the type of a variable or an expression.
- E.g.

```
typeof ""              // Returns "string"
typeof "John"          // Returns "string"
typeof "John Doe"      // Returns "string"
typeof 0               // Returns "number"
typeof 314             // Returns "number"
typeof 3.14            // Returns "number"
typeof (3)             // Returns "number"
typeof (3 + 4)         // Returns "number"
```

7. Undefined:

- In JavaScript, a variable without a value, has the value undefined. The type is also undefined.
- E.g.
  var car; // Value is undefined, type is undefined
- Any variable can be emptied, by setting the value to undefined. The type will also be undefined.
- E.g.
  car = undefined; // Value is undefined, type is undefined
- You can empty an object by setting it to undefined.
- E.g.

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = undefined;   // Now both value and type is undefined
```

8. Empty Values:

- An empty value has nothing to do with undefined.
- An empty string has both a legal value and a type.
- E.g.
  var car = ""; // The value is "", the typeof is "string"

9. Null:

- In JavaScript null is "nothing". It is supposed to be something that doesn't exist.
- In JavaScript, the data type of null is an object.
- You can empty an object by setting it to null.
- E.g.

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null;    // Now value is null, but type is still an object
```

- Undefined and null are equal in value but different in type.

10. Primitive Data:

- A primitive data value is a single simple data value with no additional properties and methods.
- The typeof operator can return one of these primitive types:
  - String
  - Number
  - Boolean
  - undefined

- E.g.
  typeof "John" // Returns "string"
  typeof 3.14 // Returns "number"
  typeof true // Returns "boolean"
  typeof false // Returns "boolean"
  typeof x // Returns "undefined" (if x has no value)

11. Complex Data:

- The typeof operator can return one of two complex types:
  - Function
  - Object
- The typeof operator returns "object" for objects, arrays, and null.
- The typeof operator does not return "object" for functions. Instead, it returns “function” for functions.
- Note: The typeof operator returns "object" for arrays because in JavaScript arrays are objects.
- E.g.
  typeof {name:'John', age:34} // Returns "object"
  typeof [1,2,3,4] // Returns "object"
  typeof null // Returns "object"
  typeof function myFunc(){} // Returns "function"
- **Variables:**
- JavaScript variables are containers for storing data values.
- Variables are defined using one of three keywords, **var**, **let** and **const**, and the = operator.
- **Note:** var is global while let is local. For this reason, it is better to use let than var.
- Javascript variables can only start with a letter, dollar sign ($) or underscore, are case sensitive and can only contain letters, numbers, underscores and dollar signs.
- A variable declared without a value will have the value undefined.
- E.g.
  let name; // name is undefined.
- **Note:** If you re-declare a JavaScript variable, it will not lose its value.
- E.g. The variable carName will still have the value "Volvo" after the execution of these statements:
  let carName = "Volvo";
  let carName;
- **Arrays:**
- JavaScript arrays are used to store multiple values in a single variable.
- E.g.
  let cars = ["Saab", "Volvo", "BMW"];
- An array is a special variable, which can hold more than one value at a time.
- Syntax for creating an array:
  let variableName = [item1, item2, … itemn];
- **Note:** Spaces and line breaks are not important. A declaration can span multiple lines.
- E.g.
  let variableName = [
  Item1,
  Item2,

```
…
Itemn
];
```

- **Note:** Another way we can create an array is using the new keyword.
  Syntax: **let variableName = new Array(item1, item2, …, itemn);**
- E.g.
  let cars = new Array("Saab", "Volvo", "BMW");
- You access an array element by referring to the index number. Index 0 refers to the first element, index 1 refers to the second element and so on.
- Arrays are a special type of objects. The typeof operator in JavaScript returns "object" for arrays. However, arrays always use numbers to access its elements while objects always use names to access its members.
- E.g. Consider the JS array and object below:
  let person = ["John", "Doe", 46];
  let person = {firstName:"John", lastName:"Doe", age:46};
  To access the first element of the array, you'd do person[0], but to access the first element of the object, you'd do person.firstName.
- The easiest way to add a new element to an array is using the push() method.
- E.g.
  let fruits = ["Banana", "Orange", "Apple", "Mango"];
  fruits.push("Lemon"); // adds a new element (Lemon) to fruits
- The length property of an array returns the length of an array (the number of array elements).
- E.g.
  var fruits = ["Banana", "Orange", "Apple", "Mango"];
  fruits.length; // the length of fruits is 4
- **For Loops:**
- Syntax:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

- Statement 1 is executed (one time) before the execution of the code block. Normally you will use statement 1 to initialize the variable used in the loop.
- Statement 2 defines the condition for executing the code block.
- Statement 3 is executed (every time) after the code block has been executed. Often statement 3 increments/decrements the value of the initial variable.
- E.g.

```
for (i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

- **For In Loops:**
- The JavaScript for/in statement loops through the properties of an object.
- E.g.

```
var person = {fname:"John", lname:"Doe", age:25};

var text = "";
var x;
for (x in person) {
```

```
  text += person[x];
}
```

- **For Of Loops:**
- The JavaScript for/of statement loops through the values of an iterable objects
- for/of lets you loop over data structures that are iterable such as Arrays, Strings, etc.
- Syntax:

```
for (variable of iterable) {
  // code block to be executed
}
```

- **variable:** For every iteration the value of the next property is assigned to the variable. variable can be declared with const, let, or var.
- **iterable:** An object that has iterable properties.
- **E.g. Looping over an array:**

```
var cars = ['BMW', 'Volvo', 'Mini'];
var x;

for (x of cars) {
  document.write(x + "<br >");
}
```

- E.g. Looping over a string:

```
var txt = 'JavaScript';
var x;

for (x of txt) {
  document.write(x + "<br >");
}
```

- The difference between for in and for of loops is that for in loops return a list of keys on the object being iterated, whereas for of loops return a list of values of the numeric properties of the object being iterated.
- E.g. Consider the below 2 pieces of code:

1.

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var txt = "";
var person = [1, 2, 3];
var x;
for (x in person) {
  txt += x + " ";
}
document.getElementById("demo").innerHTML = txt;
</script>

</body>
```

```
</html>
```

This returns 0 1 2.

2.

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var txt = "";
var person = [1, 2, 3];
var x;
for (x of person) {
  txt += x + " ";
}
document.getElementById("demo").innerHTML = txt;
</script>

</body>
</html>
```

This returns 1 2 3.

- **For Each Loop:**
- Used to loop over arrays.
- The forEach() method calls a function once for each element in an array, in order.
- **Note:** The function is not executed for array elements without values.
- Syntax: **array.forEach(function(currentValue, index, arr), thisValue)**

Parameter	Description
function	Required. A function to be run for each element in the array.
currentValue	Required. The value of the current element
index	Optional. The array index of the current element
arr	Optional. The array object the current element belongs to
thisValue	Optional. A value to be passed to the function to be used as its "this" value. If this parameter is empty, the value "undefined" will be passed as its "this" value

- E.g.

```
var sum = 0;
var numbers = [65, 44, 12, 4];
numbers.forEach(myFunction);
```

```
function myFunction(item) {
  sum += item;
  document.getElementById("demo").innerHTML = sum;
}
```

- **While Loops:**
- The while loop loops through a block of code as long as a specified condition is true.
- Syntax:

```
while (condition) {
  // code block to be executed
}
```

- E.g.

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

- **Do While Loop:**
- The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.
- Syntax:

```
do {
  // code block to be executed
}
while (condition);
```

- E.g.

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

- **If Statements:**
- Syntax:

```
if (condition 1) {
...
}
else if (condition 2){
…
}
else{
…
}
```

- **Note:** You must have the if block and only 1 if block. You can have as many else if blocks as you want, and you can have at most 1 else block.
- E.g.

```
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
```

```
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

- **Switch Statement:**
- The switch statement is used to perform different actions based on different conditions.
- Use the switch statement to select one of many code blocks to be executed.
- Syntax:

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

- This is how it works:
    - The switch expression is evaluated once.
    - The value of the expression is compared with the values of each case.
    - If there is a match, the associated block of code is executed.
- E.g.

  The getDay() method returns the weekday as a number between 0 and 6.
  (Sunday=0, Monday=1, Tuesday=2 ..)
  This example uses the weekday number to calculate the weekday name:

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
```

```
}
```

- When JavaScript reaches a **break** keyword, it breaks out of the switch block. This will stop the execution of inside the block.
- The **default** keyword specifies the code to run if there is no case match.
- **Note:** The default case does not have to be the last case in a switch block, but if default is not the last case in the switch block, remember to end the default case with a break.
- **Functions:**
- A JavaScript function is a block of code designed to perform a particular task.
- A JavaScript function is executed when something invokes it. There are 3 ways we can invoke a function:
    1. When an event occurs (when a user clicks a button)
    2. When it is invoked (called) from JavaScript code
    3. Automatically (self invoked)
- You declare a function by prefixing it with the function keyword.
- Syntax:

```
function function_name(...){
…
}
```

- E.g.

```
function myFunction(p1, p2) {
  return p1 * p2;   // The function returns the product of p1 and p2
}
```

- Variables declared within a JavaScript function, become local to the function. **Local variables** can only be accessed from within the function. Since local variables are only recognized inside their functions, variables with the same name can be used in different functions. Local variables are created when a function starts, and deleted when the function is completed.
- **Objects:**
- An object in Javascript is similar to a dictionary in Python. Unlike in Python, the keys of the object must be String but the value can be any valid type. This type of structure is called a **JSON or Javascript Object Notation**.
- E.g.

```
myObject = {
  "firstName": "kevin",
  "lastName": "zhang",
  "age": 69,
  "cool": true
}
```

- The name:values pairs in JavaScript objects are called properties.
- You can access object properties in two ways:
    1. **objectName.propertyName**
    2. **objectName["propertyName"]**
- Objects can also have methods.
- Methods are actions that can be performed on objects.
- Methods are stored in properties as function definitions.
- E.g.

```
var person = {
  firstName: "John",
```

```
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

- You access an object method with the following syntax:
  objectName.methodName()
- E.g.
  name = person.fullName();
- In a function definition, **this** refers to the "owner" of the function. It is equivalent to self in Python.
- In the example above, **this** is the person object that "owns" the fullName function.
  I.e. this.firstName means the firstName property of this object.
- When a JavaScript variable is declared with the keyword "new", the variable is created as an object. Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.
- E.g. Creating an object using the keyword “new”:
  let apple = new Object();
  apple.color = “red”;
  apple.shape = “round”;
  This is equivalent to **let var = {“color”: “red”, “shape”: “round”};**
- However, the above way is tedious and won’t work well if you want to create many similar objects. You can use a constructor pattern instead.
- E.g.

```
function Fruit(name, color, shape){
        this.name = name;
        this.color = color;
        this.shape = shape;
}

let apple = new Fruit(‘apple’, ‘red’, ‘round’);
let melon = new Fruit(‘melon’, ‘green’, ‘round’);
```

- **Print Statements:**
- JavaScript can output data in different ways:
  - Writing into an HTML element, using **innerHTML**.
  - Writing into the HTML output using **document.write()**.
  - Writing into an alert box, using **window.alert()**.
  - Writing into the browser console, using **console.log()**.
    Note: You can view console output in Chrome by pressing CMD+OPTION+J on a Mac or CTRL+SHIFT+J on Windows.
- Using innerHTML:
  - To access an HTML element, JavaScript can use the **document.getElementById(id)** method. This is the most common way of obtaining an element to modify. Since IDs in HTML are guaranteed to be unique, we can access a pre-existing element in an HTML form using this strategy.
  - The id attribute defines the HTML element. The innerHTML property defines the HTML content.

- E.g.

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My First Paragraph</p>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>

</body>
</html>
```

- E.g.

HTML (index.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
  <!-- we will not see the text here because it will be overwritten
   by the Javascript -->
  <p id="my-content">This is the original content</p>
  <script src="script.js"></script>
</body>
</html>
```

Javascript (script.js):

```
document.getElementById('my-content').innerHTML = 'keviniscool';
```

- **Note:** There is a similar command called **document.getElementsByClassName()** however since classes are not unique, this command will always return the result in an array.

- Using document.write():
  - For testing purposes, it is convenient to use document.write().
  - **Note:** Using document.write() after an HTML document is loaded, will delete all existing HTML.
  - E.g.

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>
```

```
<button type="button" onclick="document.write(5 + 6)">Try it</button>

</body>
</html>
```

- Using window.alert():
  - You can use an alert box to display data.
  - E.g.

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
window.alert(5 + 6);
</script>
</body>
</html>
```

- Using console.log()
  - For debugging purposes, you can use the console.log() method to display data.
  - E.g.

```
<!DOCTYPE html>
<html>
<body>
<script>
console.log(5 + 6);
</script>
</body>
</html>
```

- **Events:**
- An **HTML event** can be something the browser does or something a user does.
- Here are some examples of HTML events:
  - An HTML web page has finished loading
  - An HTML input field was changed
  - An HTML button was clicked
- Often, when events happen, you may want to do something. JavaScript lets you execute code when events are detected.
- HTML allows event handler attributes, along with JavaScript code, to be added to HTML elements.
- A table of HTML event:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element

onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

- Event handlers can be used to handle, and verify, user input, user actions, and browser actions:
    - Things that should be done every time a page loads.
    - Things that should be done when the page is closed.
    - Action that should be performed when a user clicks a button.
    - Content that should be verified when a user inputs data.
- Many different methods can be used to let JavaScript work with events:
    - HTML event attributes can execute JavaScript code directly.
    - HTML event attributes can call JavaScript functions.
    - You can assign your own event handler functions to HTML elements.
    - You can prevent events from being sent or being handled.
- E.g. In the past labs, we've made forms that don't do anything. Using Javascript, we can retrieve input from forms and perform interactive behaviour with the input we've collected. The most common way of doing this is by accessing the .value attribute of an element and reacting to an event such as onclick().

HTML (index.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Website</title>
</head>
<body>
   <label>What is your name?</label>
   <input type="text" id="name-form" placeholder="Enter your name">
   <button type="submit" onclick="nameResult()">Submit</button>
   <!-- we put a placeholder <p> here so that it can be written into
   later -->
   <!-- using document.write() would overwrite the entire body -->
   <p id="result"></p>
   <script src="script.js"></script>
</body>
</html>
```

Javascript (script.js):

```
// this function is called whenever the button is clicked
function nameResult() {
  // get the value inside the form
  name = document.getElementById('name-form').value;
  // add the value
  document.getElementById('result').innerHTML = 'your name is: ' + name;
}
```

Lecture Notes:

- **Front-end** refers to the client side while **back-end** refers to the server side. The languages that we use for front-end include HTML, JavaScript and CSS. We will be using Python and Flask for the back-end.
- A HTTP Get request will show the query in the url.
  E.g. **/test/demo_form.php?name1=value1&name2=value2**
- An URL is generally in the form:
  scheme://host/some/path/to/file?query1=value&query2=value&...&queryn=value
  The ? separates the path from the query and the & separates 2 queries.
  I.e. The ? delimits the storage details from the query string and the & is used to delimit query string parameters.
- render_template is a function you can import from flask in Python that allows you to modify your HTML file with data from your Python file.
- request is another function you can import from flask in Python that allows you to do Get requests.
- g is another function you can import from flask in Python that allows flask to interact with the sqlite database.
- Here is a simple example of how you can use SQLite 3 with Flask:

```
// These functions were gotten from
// https://flask.palletsprojects.com/en/1.1.x/patterns/sqlite3/
import sqlite3
from flask import Flask, render_template, g

DATABASE = '/path/to/database.db'

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db

def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv

def make_dicts(cursor, row):
    return dict((cursor.description[idx][0], value)
                for idx, value in enumerate(row))

@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()
```

- Here's what the get_db function does:
    1. The line "db=getattr(g, '_database', None)" creates a reference database called db and we're gonna use the getattr function to see if the attribute _database is defined on g. If g is not assigned to any value, we will assign it to None. Now, we know that db will be None if we are trying the database for the first time.
    2. The line "if db is None:" will run because of the above statement.
    3. The line "db=g._database=sqlite3.connect(DATABASE)" creates the connection to the database.

  In summary, get_db tries to make a connection with a database.
- The function query_db takes in an SQL query and will run it against the database. rv is an array that contains all the output rows of the query. The "one=False" argument means that all valid rows will be returned, not just the first.
- For each row in the database, the make_dict function will return it in dictionary form.
- Here's what the close_connection function does:
  For the line "db = getattr(g, '_database', None)", db is most likely not None as the user has connected it to a database in the get_db function, so this won't do anything. This is to double check that the user has opened a connection. In the case that the user did not connect to a database, db is set to None. Now, if db is not None, then we close the connection with the database.
  In summary, close_connection checks if db has been connected to a database (I.e. db is not None) and if db is not None, then we close the connection.
- Now, we'll build an actual application using Flask and SQLite.
  Assume that there is a database called database.db and that there's a table called employees. Furthermore, assume the employees table looks something like the following, but with more rows:

ID	First Name	Last Name	Position	Salary
1	Michael	Scott	Regional Manager	75, 000
2	Dwight	Schrute	Assistant to the Regional Manager	70, 000

Here will be the code:

```
import sqlite3
from flask import Flask, render_template, g

DATABASE = '/database.db'

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db

def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv
```

```
def make_dicts(cursor, row):
   return dict((cursor.description[idx][0], value)
           for idx, value in enumerate(row))

app = Flask(__name__)

@app.teardown_appcontext
def close_connection(exception):
   db = getattr(g, '_database', None)
   if db is not None:
      db.close()

@app.route('/')
def root():
  db = get_db()
  db.row_factory = make_dicts

  employees = []
  for employee in query_db('select * from employees'):
    employees.append(employee)

 db.close()
 return employees.__str__()
```

This will get you an array of dictionaries where each dictionary represents a row in the database table.

Flask Basic Authentication:

- Python/Flask Code:

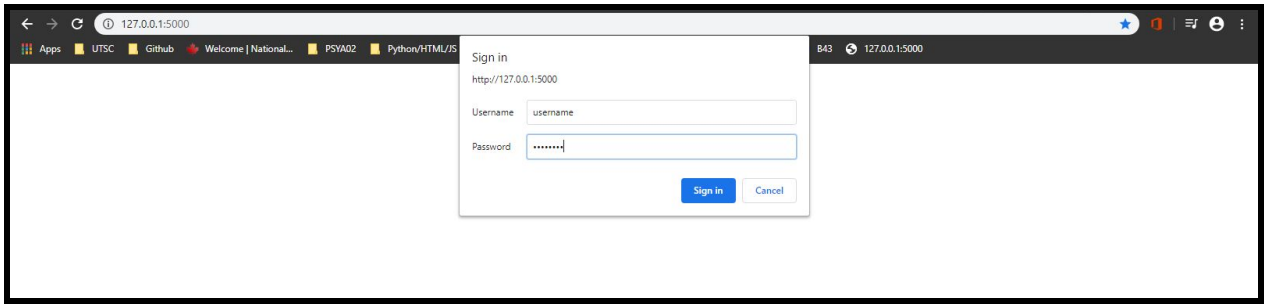
```python
from flask import Flask, request, make_response

app = Flask(__name__)

@app.route('/')
def index():
    #request.authorization.username gets the username the user entered.
    #request.authorization.password gets the password the user entered.

    # If the authorization exists inside the request and
    # both the username and password is correct, then the words "You are logged in" will appear.
    # Otherwise, a custom message will be displayed.
    if (request.authorization and request.authorization.username=="username1" and request.authorization.password=="password"):
        return "<h1> You are logged in. </h1>"

    # make_response takes 3 inputs:
    # 1. The message to the user.
    # 2. An HTTP Status Code. (Note: This is optional).
    # 3. Headers that are sent over that tells the browser that the route requires
    # HTTP basic authentication.
    return make_response('Could not verify', 401, {'WWW-Authenticate': 'Basic realm="Login Required"'})

@app.route('/page')
def page():
    return "<h1> You are in another page. </h1>"


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

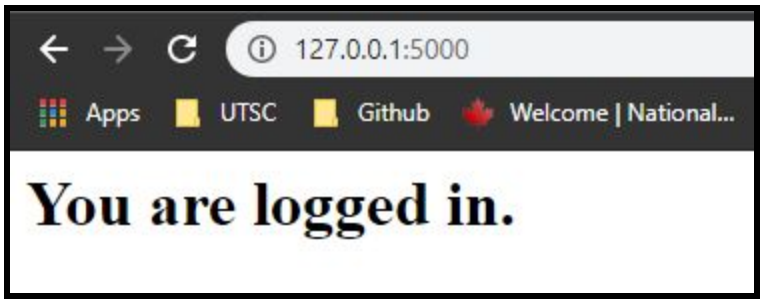
- Once you log in, the browser automatically sends the login information with each request afterwards. Furthermore, if only the homepage has the authentication requirement, if you log in and change your username/password, if you go to any page other than the homepage, you don't need to log in again. This is shown below:

  Original (Username is username and Password is password):

```
if (request.authorization and request.authorization.username=="username" and request.authorization.password=="password"):
    return "<h1> You are logged in. </h1>"
```

  Home Page (Needed to log in):





  Other Page (Didn't need to log in):

  Now, let's say I change my username to username1:

```
if (request.authorization and request.authorization.username=="username1" and request.authorization.password=="password"):
    return "<h1> You are logged in. </h1>"
```

When I refresh http://127.0.0.1:5000/page, I don't need to log in a second time.

However, if I go back to the home page, I do need to log in again.

- If you do want the user to enter their username and password at any page if they change their username or password, we can use decorators to do this. The code is shown below:

app.py

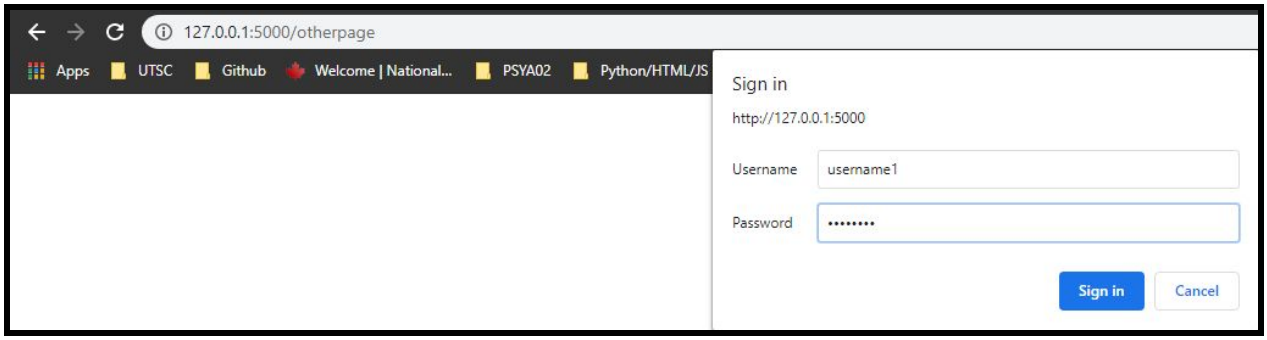
```python
from flask import Flask, request, make_response
from functools import wraps

app = Flask(__name__)

# Creates the decorator.
# It takes a function, which is the function that is being decorated.
def auth_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if (auth and auth.username == "username1" and auth.password == "password"):
            return f(*args, **kwargs)
        return make_response('Could not verify', 401, {'WWW-Authenticate': 'Basic realm="Login Required"'})
    return decorated

@app.route('/')
@auth_required
def index():
    return "<h1> You are logged in to the home page. </h1>"

@app.route('/page')
@auth_required
def page():
    return "<h1> You are logged in to another page. </h1>"

@app.route('/otherpage')
@auth_required
def other_page():
    return "<h1> You are logged in to the other page. </h1>"


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

- When I first log in, I need to enter the username and password:

- Now, if I go to page or other page, I don't need to reenter the login info:

- Now, if I change the username as such, I have to enter the new username/password on any of the 3 pages.

```
def auth_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if (auth and auth.username == "username1" and auth.password == "password"):
            return f(*args, **kwargs)
        return make_response('Could not verify', 401, {'WWW-Authenticate': 'Basic realm="Login Required"'})
    return decorated
```



- **make_response:**
  - This function can be called instead of using a return and you will get a response object which you can use to attach headers.
  - The HTTP WWW-Authenticate response header defines the authentication method that should be used to gain access to a resource.
  - The WWW-Authenticate header is sent along with a 401 Unauthorized response.
  - Syntax: **WWW-Authenticate: <type> realm=<realm>**
    <type>: This is the aAuthentication type. The most common authentication scheme is the "Basic" authentication scheme. The "Basic" HTTP authentication scheme transmits credentials as user ID/password pairs, encoded using base64. As the user ID and password are passed over the network as clear text (it is base64 encoded, but base64 is a reversible encoding), the basic authentication

scheme is not secure.
realm=<realm>: A description of the protected area. If no realm is specified, clients often display a formatted hostname instead.

- ***args and **kwargs:**
  - *args and *kwargs are special keywords which allow functions to take variable length arguments.
  - The special syntax *args in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.
  - The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.
  - What *args allows you to do is take in more arguments than the number of formal arguments that you previously defined. With *args, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
  - The special syntax **kwargs in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).
  - A **keyword argument** is where you provide a name to the variable as you pass it into the function. Keyword arguments do not care about the position of their arguments.
    E.g.

```
def num_comparator(num1, num2):
    if(num1 > num2):
        print(str(num1) + " is bigger than " + str(num2))
    elif(num1 < num2):
        print(str(num1) + " is smaller than " + str(num2))
    else:
        print(str(num1) + " is equal to " + str(num2))

# All 3 function calls will print out: "1 is less than 2"
num_comparator(1, 2)
num_comparator(num1=1, num2=2) # This is a keyword argument.
num_comparator(num2=2, num1=1) # This is also a keyword argument.
```

  - One can think of the kwargs as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the kwargs there doesn't seem to be any order in which they were printed out.
    E.g.

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        print (key + ": " + value)

# Driver code
myFun(first ='Hi', mid ='World')
```

These are the outputs:

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop/CSCB20 Code/Args and Kwargs$ python3 kwargs.py
mid: World
first: Hi
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop/CSCB20 Code/Args and Kwargs$ python3 kwargs.py
first: Hi
mid: World
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop/CSCB20 Code/Args and Kwargs$
```

Flask-Login:

- **Note:** You need to download the flask-login package to use it. To install it, you can do pip install flask-login.
- Flask-Login provides user session management for Flask. It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time.
- It will:
  - Store the active user's ID in the session, and let you log them in and out easily.
  - Let you restrict views to logged-in (or logged-out) users.
  - Handle the normally-tricky "remember me" functionality.
  - Help protect your users' sessions from being stolen by cookie thieves.
- However, it does not:
  - Impose a particular database or other storage method on you. You are entirely in charge of how the user is loaded.
  - Restrict you to using usernames and passwords, OpenIDs, or any other method of authenticating.
  - Handle permissions beyond "logged in or not."
  - Handle user registration or account recovery.
- The most important part of an application that uses Flask-Login is the LoginManager class. You should create one for your application somewhere in your code, like this:
  login_manager = LoginManager()
- The login manager contains the code that lets your application and Flask-Login work together, such as how to load a user from an ID, where to send users when they need to log in, and the like.
  I.e. LoginManager is used to hold the settings used for logging in.
- Once the actual application object has been created, you can configure it for login with:
  login_manager.init_app(app)
- By default, Flask-Login uses sessions for authentication. This means you must set the secret key on your application, otherwise Flask will give you an error message telling you to do so.
- The User class is the class that you use to represent users needs to implement the following properties and methods. You can inherit the methods from UserMixin.
- These are the methods:
  - **is_authenticated:** This property should return True if the user is authenticated. I.e. they have provided valid credentials.
  - **is_active:** This property should return True if this is an active user, in addition to being authenticated, they also have activated their account, not been suspended, or any condition your application has for rejecting an account.
  - **is_anonymous:** This property should return True if this is an anonymous user. Actual users should return False instead.
  - **get_id():** This method must return a unicode that uniquely identifies this user, and can be used to load the user from the user_loader callback. Note that this

must be a unicode. If the ID is natively an int or some other type, you will need to convert it to unicode.

- Views that require your users to be logged in can be decorated with the **login_required** decorator:
  E.g.

```
@app.route("/settings")
@login_required
def settings():
    pass
```

  E.g. When the user is ready to log out:

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(somewhere)
```

Redirect:

- The Flask class has a redirect() function. When called, it returns a response object and redirects the user to another target location with specified status code.
- General syntax: **redirect(location, statuscode, response)**
  The location parameter is the URL where response should be redirected.
  The statuscode parameter is sent to the browser's header. The default value is 302.
  The response parameter is used to instantiate response.
  Note: Of the 3 parameters above, only the location parameter is required.
- As shown below, the redirect function is very useful when combined with the url_for function.

URL_FOR:

- The **url_for()** function is very useful for dynamically building a URL for a specific function. The function accepts the name of a function as first argument, and one or more keyword arguments, each corresponding to the variable part of URL.
- General syntax: **url_for(function_name)**
- **Note:** url_for() can be used with redirect() to redirect the user to another function and another webpage.
- **Note:** We can also use url_for in our HTML code.

Sessions:

- **Session** is the time interval when a client logs into a server and logs out of it. The data, which is needed to be held across this session, is stored in the client's browser.
  Note: Suppose the client logs into Gmail using Google Chrome. Because the session data is stored in the client's browser (Google Chrome), if the client were to open up another tab and try to access Gmail again, then they would not need to log in a second time. However, if the client opens up a new browser, say Internet Explorer, and tries to access Gmail, then they would need to log in again.
- We cannot have session data used across browsers. This is due to security issues.
- A session with each client is assigned a Session ID. The Session data is stored on top of cookies and the server signs them cryptographically. For this encryption, a Flask application needs a defined SECRET_KEY. To create your secret key, you need to set app.secret_key to some string that you choose to be your secret key.
  Here is the general syntax to create this secret key:
  app.secret_key = 'your secret key'
  E.g. **app.secret_key = 'Rick'**
- The Session object is a dictionary object containing key-value pairs of session variables and associated values.
  General syntax for creating a Session object:
  Session[variable name] = value
  E.g. **Session['username'] = 'admin'**
- To remove a session variable, use the pop() method on the session object and mention the variable to be removed.
  General syntax: **session.pop(variable name, none)**
  E.g. **session.pop('username', None)**
- To create a logout feature, do something similar to below:

```
@app.route('/logout')
def logout():
  session.pop(variable name, None)
  return redirect(url_for(function_name'))
```

-

Flash:

- You can use the flash() function to display a message on the screen.
- General syntax: **flash(message, category)**
  The message parameter is the actual message to be flashed.
  The category parameter is optional. It can be either 'error', 'info' or 'warning'.
- On the HTML page(s), you can use the get_flashed_messages() function to get the flash message.
- Typically, the HTML side would look something like this:

```
{% with messages = get_flashed_messages() %}
        {% if messages %}
                {% for message in messages %}
                  ...
                {% endfor %}
          {% endif %}
{% endwith %}
```

- First, we get the flash message from get_flashed_messages(). Then, if there are any messages, we loop through all the messages and do something with them.

Building a login page:

- Using the above information, we can create a login page. Consider the code snippets below:

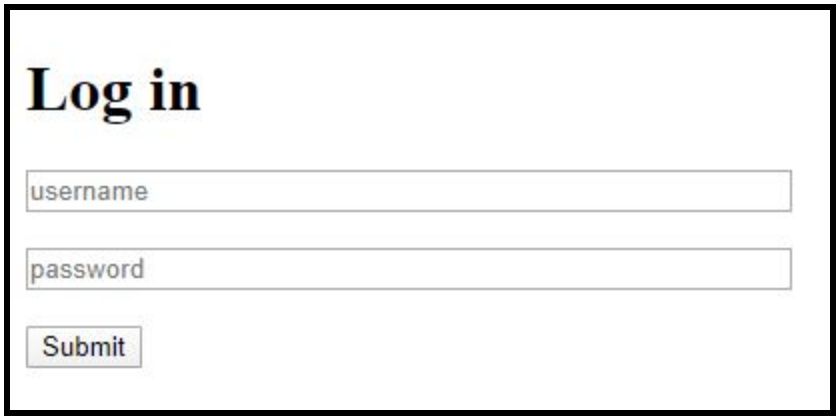
```python
from flask import Flask, request, redirect, render_template, session, url_for, flash

app = Flask(__name__)

# When the user goes onto the home page, they will be redirected to the login page.
@app.route('/')
def index():
    return redirect(url_for('login'))

# The login page.
@app.route('/login', methods=["GET", "POST"])
def login():


    # Removes the user id if there is one inside the session.
    # This way, if the user goes to the login page from any other page, they must login again.
    session.pop('user_id', None)

    if(request.method == "POST"):

        # Gets the username and password the user enters.
        username = request.form['username']
        password = request.form['password']

        # Compares the username and password the user entered to the correct one.
        if (username == "username" and password == "password"):

            session['user_id'] = username
            session['logged_in'] = True

            # Redirects the user to their home page.
            return redirect(url_for('home'))
        else:

            # If the user enters an invalid username password combination, then this error message will show up.
            flash('You have entered an invalid username and/or password. Please Try Again.')
    return render_template('index.html')
```

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

@app.route('/logout')
def logout():
    # Removes the user's session and redirects them to the login page.
    session.pop('user_id', None)
    return redirect(url_for('login'))

if (__name__ == "__main__"):
    # Creates a secret key.
    app.secret_key = b"secretkey"
    app.run(debug = True)
```

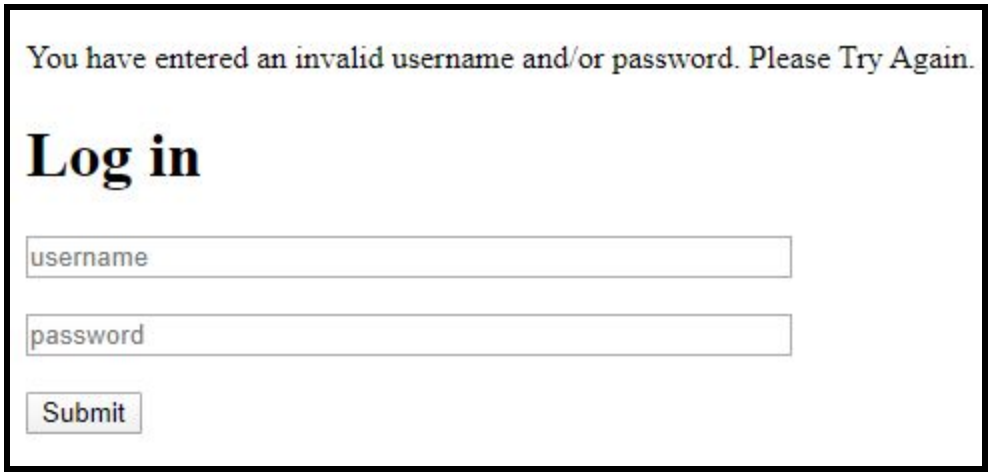
```html
<!DOCTYPE html>
<html>
<head>
    <title> Log In </title>
</head>
<body>
    <!-- This is index.html. -->
    <!-- Prints the error message. -->
    {% with error_message = get_flashed_messages() %}
        {% if error_message %}
            <!-- Loops through the error messages. -->
            {% for error in error_message %}
                <p id="error"> {{ error }} </p>
            {% endfor %}
        {% endif %}
    {% endwith %}

    <!-- Lets the user log in. -->
    <h1> Log in </h1>
    <form action="/login" method="POST">
      <input class="box_content" type="text" name="username" placeholder="username" size=50 required><br> <br>
      <input class="box_content" type="password" name="password" placeholder="password" size=50 required><br> <br>
      <input class="button" type="submit" value="Submit">
    </form>

</body>
</html>
```

```html
<!DOCTYPE html>
<html>
<head>
    <title> Home Page</title>
</head>
<body>
    <!-- This is home.html. -->
    <p> Welcome to your home page! </p>

    <!-- Creates a link that allows the user to log out. -->
    <!-- Note that url_for() takes in the logout function name in login.py. -->
    <a href="{{url_for('logout')}}"> Logout </a>
</body>
</html>
```

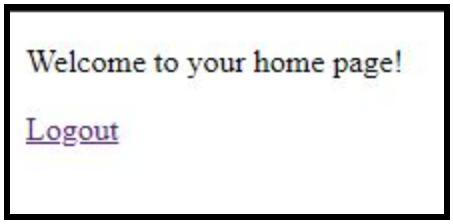
This is what the HTML page looks like:
This is the login screen.



If a user tries to enter an invalid username password combination, then the flash() error message will appear:

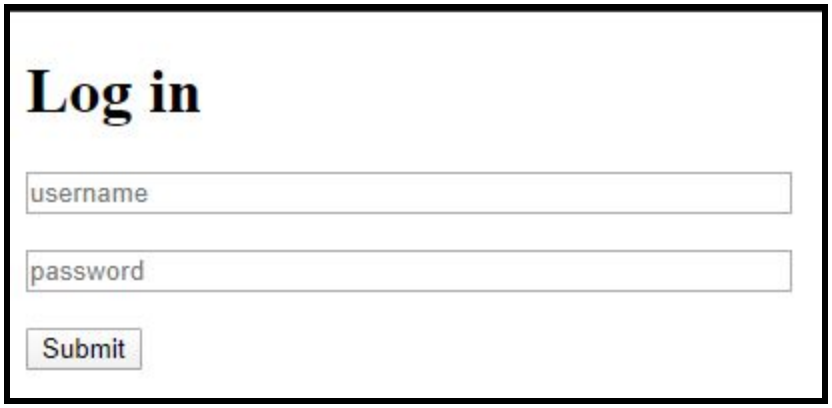


If the user does enter in a valid username password combination, then they will be redirected to the home page:



If the user clicks on the Logout link, they will be logged out and will return to the login

page.



SQLite:

- **Setting up the database:**

```
import sqlite3
from flask import Flask, render_template, request, g

# the database file we are going to communicate with
DATABASE = '/path/to/database.db'

# connects to the database
def get_db():
    # if there is a database, use it
    db = getattr(g, '_database', None)
    if db is None:
        # otherwise, create a database to use
        db = g._database = sqlite3.connect(DATABASE)
    return db

# converts the tuples from get_db() into dictionaries
def make_dicts(cursor, row):
    return dict((cursor.description[idx][0], value)
                for idx, value in enumerate(row))

# given a query, executes and returns the result
def query_db(query, args=(), one=False):

    # create a cursor called "cur" and execute the query "query" with arguments
"args"
    cur = get_db().execute(query, args)

    # get all the results (this function only works for SELECT statements)
    rv = cur.fetchall()

    # close the connection (do not leave an open connection)
    cur.close()
```

```
# return the results depending on if there are many or just one
return (rv[0] if rv else None) if one else rv
```

- Whenever we want to query information from the database, we need to set up a connection with the database which is done using the function **get_db()**. This function returns a database object of which we can apply queries to.
- The function of **make_dicts()** is to turn the data returned as tuples to dictionaries. By default, Flask returns database data as tuples which are an inconvenient data structure to use. This function utilizes the Factory design pattern to generate dictionaries for all the tuples normally returned by Flask. This function is technically not necessary but will make your life infinitely easier in the long run.
  E.g. I have the database called test.db and it contains this information:

Table: test

	FirstName	LastName	Address
	Filter	Filter	Filter
1	Rick	Lan	123 ABC Road
2	ABC	DEF	234 XYZ Drive

Now, consider these 2 functions:

```
@app.route('/')
def home():
    db = get_db()
    info = query_db('SELECT * FROM test', one=True)
    db.close()
    print("I am not using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''
```

```
@app.route('/Test')
def test():
    db = get_db()
    db.row_factory = make_dicts
    info = query_db('SELECT * FROM test', one=True)
    db.close()
    print("I am using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''
```

This is what you see in the terminal:

```
 * Detected change in '/mnt/c/Users/rick/Desktop/app.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
I am not using make_dicts here.
('Rick', 'Lan', '123 ABC Road')
127.0.0.1 - - [06/Apr/2020 02:15:45] "GET / HTTP/1.1" 200 -
I am using make_dicts here.
{'FirstName': 'Rick', 'LastName': 'Lan', 'Address': '123 ABC Road'}
127.0.0.1 - - [06/Apr/2020 02:15:47] "GET /Test HTTP/1.1" 200 -
```

Notice that the line after “I am not using make_dicts here” in the terminal contains the data in a tuple while the line after “I am using make_dicts here” contains the data in a dictionary with the name of the column being the key.

Note: The part of “one=True” means that you are only getting the first row that satisfies the query. In my database above, there are 2 rows, but only the first got printed. If you do not put the “one=True” part, then you will get all the row(s) that satisfies the query. Furthermore, the data will be returned in either an array of tuples, if you do not use make_dicts, or an array of dictionaries, if you do use make_dicts. Lastly, it doesn’t matter how many rows are getting returned, the data will always be inside an array.

If I refresh my two web pages and remove the “one=True” parts, as shown below,

```
@app.route('/')
def home():
    db = get_db()
    info = query_db('SELECT * FROM test')
    db.close()
    print("I am not using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''

@app.route('/Test')
def test():
    db = get_db()
    db.row_factory = make_dicts
    info = query_db('SELECT * FROM test')
    db.close()
    print("I am using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''
```

I get the following output in the terminal:

```
 * Detected change in '/mnt/c/Users/rick/Desktop/app.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
I am not using make_dicts here.
[('Rick', 'Lan', '123 ABC Road'), ('ABC', 'DEF', None)]
127.0.0.1 - - [06/Apr/2020 02:27:59] "GET / HTTP/1.1" 200 -
I am using make_dicts here.
[{'FirstName': 'Rick', 'LastName': 'Lan', 'Address': '123 ABC Road'}, {'FirstName': 'ABC', 'LastName': 'DEF', 'Address': None}]
127.0.0.1 - - [06/Apr/2020 02:28:02] "GET /Test HTTP/1.1" 200 -
```

Notice that now, all the returned data is in an array.

Now, if I change the query such that none of the rows will satisfy it, like such,

```
@app.route('/')
def home():
    db = get_db()
    info = query_db('SELECT * FROM test where FirstName=?', ["ooo"])
    db.close()
    print("I am not using make_dicts here.")
    print(info)
    return ''' <h1> Test </h1>'''
```

then the output changes to

```
 * Detected change in '/mnt/c/Users/rick/Desktop/app.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
I am not using make_dicts here.
[]
127.0.0.1 - - [06/Apr/2020 02:30:43] "GET / HTTP/1.1" 200 -
```

Notice how that even though none of the rows matched, I still got back an array.

- The function of **query_db()** is to query the database based on a given query. Given a query in the form of a string, it executes and returns the result of the query. In conjunction with make_dicts(), this return will be in the form of a dictionary. query_db() uses a database construct called a cursor to select the data based upon the query and then packages the result to finally return. query_db() is a simplification of something called a cursor action in the database. A **cursor** in a database is sort of like a cursor on a computer (i.e. the mouse pointer). It sort of "hovers" over the database and performs the queries in the database. When you run queries in SQLite browser or on the command-line, these programs create a cursor for you that you don't see and perform what is happening in this function.
- **Database teardown:**
- Whenever you close the Flask application, you need to close any open connections you have to the database. Otherwise, the next time you access the database, you may be blocked by a connection that was still left open. To do this, we create a function that will be called on application teardown that will destroy any currently open connections.
- You can use the function below to close the database.

```
# this function must come after the instantiation of the variable app
# (i.e. this comes after the line app = Flask(__name__))
@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        # close the database if we are connected to it
        db.close()
```

- **Querying Data:**
- Here are the generic steps you can use to query data:
    1. Create a database instance using get_db().
    2. Apply make_dicts to convert the tuples to dictionaries.

3. Get any request parameters (if there are any).
4. Call and execute the database query while storing it somewhere.
5. Close the database.
6. Return the results.

- **Querying without request parameters:**
  Consider the code snippets and database below:

```python
@app.route('/')
def root():
    db = get_db()
    db.row_factory = make_dicts

    # Querying without using request parameters.
    student = query_db('select * from Students')
    db.close()
    # Sending the result of the query to index.html.
    return render_template('index.html', students=student)
```

```html
<!DOCTYPE html>
<html>
<head>
    <title> Showing Querying Databases </title>
</head>
<body>

    <!-- This is how you iterate through an array of dictionaries using Flask. -->
    {% for student in students %}

        <!-- Each "student" is a dictionary. This is how we iterate through dictionaries using Flask. -->
        {% for key, value in student.items() %}

            <!-- This is how you extract the relevant information from the dictionary. -->
            <p> {{key}}: {{value}} </p>
        {% endfor %}
        <br>
    {% endfor %}
</body>
</html>
```

Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	1
2	ABC	DEF	2
3	GHI	JKL	3

The HTML output looks like this:

FirstName: Rick

LastName: Lan

StudentNumber: 1

FirstName: ABC

LastName: DEF

StudentNumber: 2

FirstName: GHI

LastName: JKL

StudentNumber: 3

- **Querying with request parameters:**

Consider the code snippers and database below:

```python
@app.route('/<name>')
def getstudent(name):
    db = get_db()
    db.row_factory = make_dicts

    # Querying using request parameters.
    student = query_db('select * from Students where FirstName=?', [name], one=True)
    db.close()
    # Sending the result of the query to index.html.
    return render_template('student.html', students=student)
```

```
<!DOCTYPE html>
<html>
<head>
    <title> Showing Querying Databases </title>
</head>
<body>

    <!-- Each "student" is a dictionary. This is how we iterate through dictionaries using Flask. -->
    {% for key, value in students.items() %}

        <!-- This is how you extract the relevant information from the dictionary. -->
        <p> {{key}} {{value}} </p>
    {% endfor %}

</body>
</html>
```

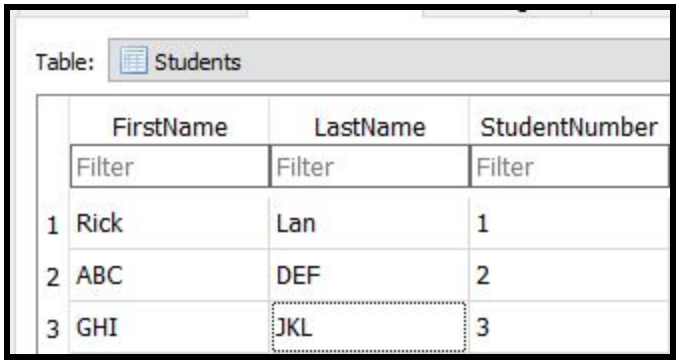
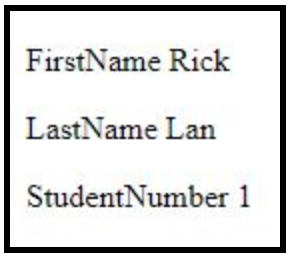


Table: Students

	FirstName	LastName	StudentNumber
	Filter	Filter	Filter
1	Rick	Lan	1
2	ABC	DEF	2
3	GHI	JKL	3

This is what the HTML output looks like:



FirstName Rick

LastName Lan

StudentNumber 1

Note: After every query, when we are completely done with the connection, you must close the connection. If you do not close the connection, the next time you access the database, you may get locked out.

Note: We replace the request parameter using a ? instead of something like **query_db('select * from items where name = {name}'.format(name=name))**. This is because using the .format() method to replace parameters leaves the code vulnerable to **SQL injection**. An SQL injection is when a malicious user writes SQL code in the text box which could cause very bad things to happen. Using the ? sanitizes the input and makes it safe to execute.

- **The general process of executing any sort of SQL query is a 3-step process:**
    1. Create a cursor.
    2. Run the query using **.execute()**. Also commit the changes if you modified data.
    3. Close the cursor.

Note: The reason why you didn't have to create a cursor or use .execute() when you were querying is because the query_db() function did it for you. You just had to call it.

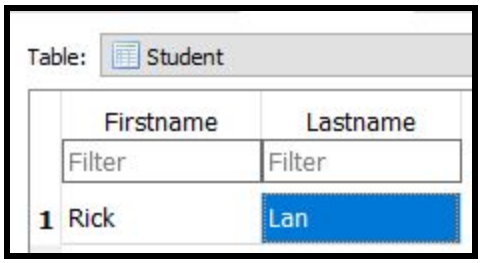
Note: For a SELECT query, the cursor does not modify any data in the database so you will immediately get back the result of the query. For a query that modifies data in the database (such as an INSERT or an UPDATE), the database does not automatically apply the query when it is executed. Instead, it "stages" the query and waits until you "commit" it to apply the changes to the database.

- **Inserting into the database:**

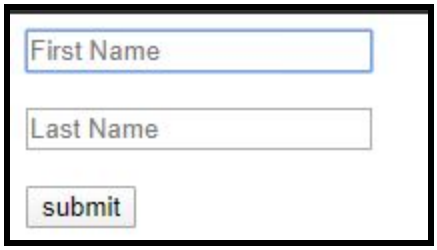
Consider the code snippets and database below:

```python
@app.route('/', methods=["GET", "POST"])
def home():
  if(request.method == "POST"):

    db = get_db()
    db.row_factory = make_dicts
    # make a new cursor from the database connection
    cur = db.cursor()

    # get the post body
    data = request.form

    # Inserts the data into the database
    cur.execute('insert into Student (Firstname, Lastname) values (?, ?)', [
      data['Firstname'],
      data['Lastname']
    ])
    # commit the change to the database
    db.commit()
    # close the cursor
    cur.close()
```

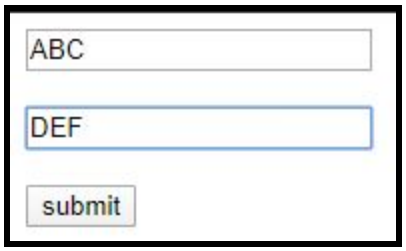
```html
<!DOCTYPE html>
<html>
<head>
    <title> Inserting into Database </title>
</head>
<body>

    <form action="/" method="POST">
        <input type="text" name="Firstname" placeholder="First Name" required><br><br>
        <input type="text" name="Lastname" placeholder="Last Name" required><br><br>
        <input type="submit" value="submit">
    </form>

</body>
</html>
```



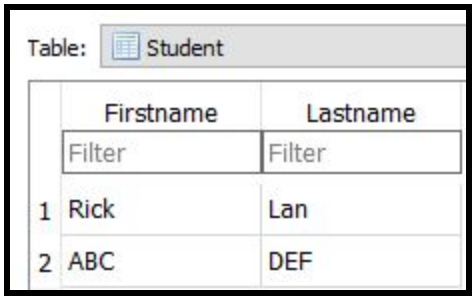
Here is what the HTML page looks like:



If I enter ABC for first name and DEF as last name and click submit:



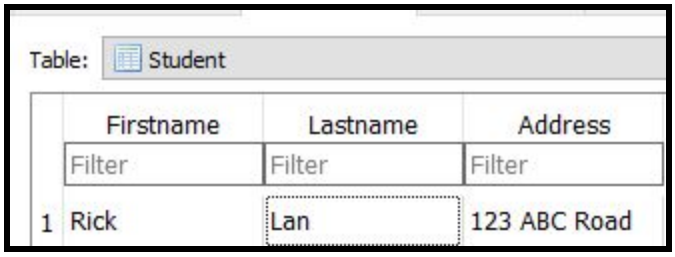
We see the changes in the database:



- **Updating information in the database:**
  Consider the code snippets and database below:

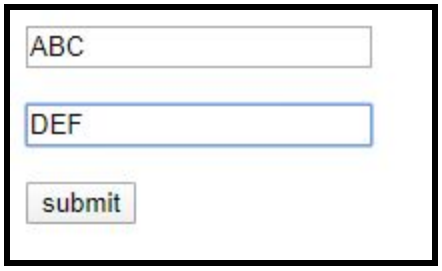
```python
@app.route('/', methods=["GET", "POST"])
def home():
  if(request.method == "POST"):

    db = get_db()
    db.row_factory = make_dicts
    # make a new cursor from the database connection
    cur = db.cursor()

    # get the post body
    data = request.form

    # Updates the data in the db.
    cur.execute('UPDATE Student SET Firstname=?, Lastname=? WHERE Address="123 ABC Road"', [
      data['Firstname'],
      data['Lastname']
    ])
    # commit the change to the database
    db.commit()
    # close the cursor
    cur.close()
```

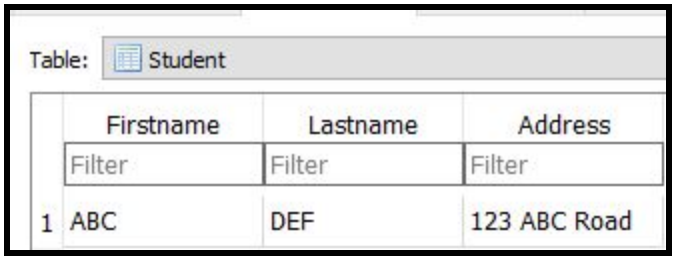
```html
<!DOCTYPE html>
<html>
<head>
    <title> Updating into Database </title>
</head>
<body>

    <form action="/" method="POST">
        <input type="text" name="Firstname" placeholder="First Name" required><br><br>
        <input type="text" name="Lastname" placeholder="Last Name" required><br><br>
        <input type="submit" value="submit">
    </form>

</body>
</html>
```



This is what the HTML page looks like:

If I type ABC for first name and DEF for last name and click submit, the database changes to:





- **Doing a natural join:**
  Consider the code snippets and database below:

```python
@app.route('/')
def home():

    db = get_db()
    db.row_factory = make_dicts

    # Does a natural join.
    natural_join = query_db('SELECT * FROM Student NATURAL JOIN Marks')
    # close the cursor
    db.close()

    return render_template('index.html', info=natural_join)
```

```html
<!DOCTYPE html>
<html>
<head>
    <title> Natural Join </title>
</head>
<body>

    {% for student in info%}
        {% for key, value in student.items() %}
            <p> {{key}}: {{value}} </p>
        {% endfor %}
        <br>
    {% endfor %}

</body>
</html>
```

Table: Marks

	StudentNumber	Mark
	Filter	Filter
1	1	80
2	2	77
3	3	85
4	4	55

Table: Student

	Firstname	Lastname	StudentNumber
	Filter	Filter	Filter
1	ABC	DEF	1
2	GHI	JKL	2
3	MNO	PQR	3
4	STU	VWXYZ	4

If I run the command “select * from Student natural join Marks;” in the database, I get this:

	Firstname	Lastname	StudentNumber	Mark
1	ABC	DEF	1	80
2	GHI	JKL	2	77
3	MNO	PQR	3	85
4	STU	VWXYZ	4	55

When I run app.py, this is what the HTML page looks like:

Firstname: ABC

Lastname: DEF

StudentNumber: 1

Mark: 80

Firstname: GHI

Lastname: JKL

StudentNumber: 2

Mark: 77

Firstname: MNO

Lastname: PQR

StudentNumber: 3

Mark: 85

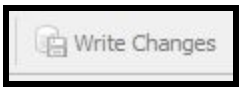
Firstname: STU

Lastname: VWXYZ

StudentNumber: 4

Mark: 55

- **Some common errors:**
  - When you manually update the database, please make sure that you click on the “Write Changes” button. If you do not and your Flask code tries to access the database, you may get errors. When the “Write Changes” button is clicked, it will grey out, as shown here

    This is what the “Write Changes” button looks like when you make change(s) to the database but don’t click it.

    The error looks like this:

  - Do not have spaces in the names of your tables or your columns. When you try to access tables/columns with names that have spaces, it might get mistaken as 2 names rather than 1.

HTTP Requests:

- Whenever we transfer data between the frontend (i.e. the HTML) and the backend (i.e. Flask), we do so using something called an HTTP request. There are four main HTTP request methods, shown below:

Method name	**Description**
GET	Used to retrieve data from the server
POST	Used to send data to the server

PUT	Used to update pre-existing data on the server
DELETE	Used to delete pre-existing data on the server

- Each of the HTTP request methods is denoted by the method= attribute in the <form> tag in your HTML. If you do not denote a method, it defaults to a GET request. To get the data, passed through the GET request, in Flask, you would need to use **request.args.get()**. Note that by standard, we are not allowed to pass data alongside a GET request; we can only add "data" that can be read by the server through the URL itself.
- If we want to send data to the server, we want to do so using a POST request instead of a GET request. To do this, we slightly modify our <form> in the HTML code by adding the attribute method="POST" to account for this. This attribute will tell Flask that we are intending to send it a POST request to the route denoted at action=. In order to catch the data, we also need to modify our Flask route by doing the following:

```
@app.route('/', methods=["GET", "POST"])
def home():
    if (request.method == "POST"):
    ...
```

Inside the app.route(), we need to specify to Flask that we intend to catch a request of type POST (not putting anything defaults to GET) and inside the function, you need to specify that the method you want is POST by doing if(request.method == "POST").
Note: You must put both GET and POST in app.route. If you don't, you may get some error message saying that "Method not allowed."

Unlike the GET request, the POST request does not send the data from the form in the URL but instead puts it in something called the POST body which is a section of the request meant to hold data. There are two main reasons why POST requests don't put data in the URL:

1. Adding a large amount of data will clutter the URL. URLs actually have a limit of around 2000 characters.
2. GET requests are not allowed to carry data whereas POST requests are. They do not have the same limitations as GET requests

In order to access the data, we access it through a variable called **request.form** which represents the POST body. This variable is in a form called JavaScript Object Notation or JSON for short. Flask sees this data type as a regular Python dictionary where the keys are equal to the name attribute in your HTML form and the values are whatever were entered into them upon submission.

- Recall that the GET request will show the query in the URL while the POST request hides it. Furthermore, the GET request will cache its data while the POST request won't. For these reasons, and some more, it's best to use the POST request when dealing with sensitive information, such as usernames, passwords, etc.
- **Note:** Do NOT mix the GET request with the POST request. If you do

```
<form action='/' method="POST">
…
</form>
```

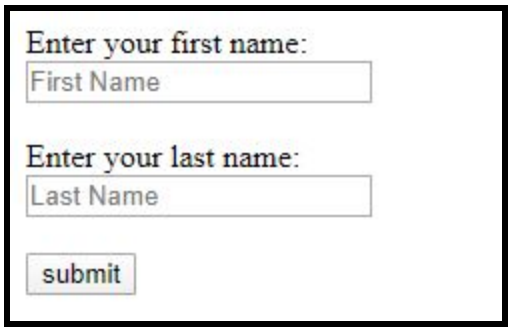
in HTML, you must use request.form in Flask. Using request.get.args will cause an error.

- **Note:** You must include the name attribute in all the places where you want the user to enter information. If you leave out the name attribute, then the data will not be submitted to the backend (Flask).
- **Note:** Say that in your HTML file, you have a <form> element and inside that <form> element, you have an <input> element with the name FirstName. Then, in Flask, you can do **request.form['FirstName']** to get the value associated with FirstName. If you just do **request.form**, it will show you all the values submitted. This is because request.form returns a dictionary with the key being the value of the name attribute. Recall that in Python, to get the value associated with a specific key in a dictionary, you do Dict['key_name'].
- **Note:** If you want to get a specific input from a GET request, use **request.args.get()**, but if you want to get all the data from a GET request, use **request.args**.
- **Note:** The action="..." specifies the endpoint of where this data should go to. When using Flask, it should match the endpoint in app.route(). For example, if you have app.route('/'), then you need to have <form action='/' ...)>.
- Here are some examples of HTTP GET and REQUEST methods:
  E.g. 1. HTTP POST Method
  Consider the pieces of code below:

```
1   from flask import Flask, request, render_template
2
3   app = Flask(__name__)
4
5   @app.route('/', methods=["GET", "POST"])
6   def home():
7       if(request.method == "POST"):
8           print(request.form)
9       return render_template('HTTP_Request_Method.html')
10
11  if (__name__ == "__main__"):
12      app.run(debug = True)
```

```html
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <form action="/" method="POST">

        <label for=FirstName> Enter your first name:</label> <br>
        <input type="text" name="FirstName" placeholder="First Name"> <br>

        <br>

        <label for=LastName> Enter your last name:</label> <br>
        <input type="text" name="LastName" placeholder="Last Name"> <br>

        <br>

        <input type="submit" value="submit">
    </form>
</body>
</html>
```

If the user goes to http://127.0.0.1:5000/, they will see the below picture.



If I enter Rick for First Name and Lan for Last Name and click on the submit button, you will see

```
 * Detected change in '/mnt/c/Users/rick/Desktop/CSCB20 Code/HTTP Request Method/HTTP_Request_Method.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
ImmutableMultiDict([('FirstName', 'Rick'), ('LastName', 'Lan')])
127.0.0.1 - - [06/Apr/2020 17:46:04] "POST / HTTP/1.1" 200 -
```

In the terminal. Furthermore, you will not see the query in the URL.

E.g. 2. HTTP POST Method but with no name attribute.

If I remove the name attribute from First Name, as shown below,

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <form action="/" method="POST">

        <label for=FirstName> Enter your first name:</label> <br>
        <input type="text" placeholder="First Name"> <br>

        <br>

        <label for=LastName> Enter your last name:</label> <br>
        <input type="text" name="LastName" placeholder="Last Name"> <br>

        <br>

        <input type="submit" value="submit">
    </form>
</body>
</html>
```

and I enter Rick for First Name and Lan for Last Name, this is what you see in the terminal:

```
127.0.0.1 - - [06/Apr/2020 17:56:40] "POST / HTTP/1.1" 200 -
 * Detected change in '/mnt/c/Users/rick/Desktop/CSCB20 Code/HTTP Request Method/HTTP_Request_Method.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
ImmutableMultiDict([('LastName', 'Lan')])
127.0.0.1 - - [06/Apr/2020 17:57:16] "POST / HTTP/1.1" 200 -
```

Notice that this time, FirstName isn't shown in the output. This is because if you don't put the name attribute, whatever the user inputs for that textbox/radiobutton/etc will not be submitted to the backend (Flask).

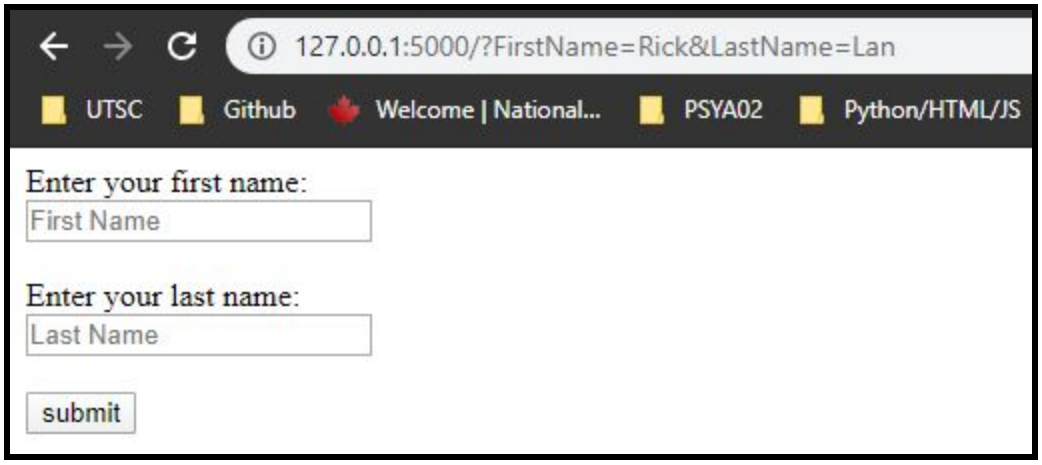
E.g. 3. HTTP GET Request
Consider the following pieces of code below:

```python
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/')
def home():

    # Will print all the data sent via a GET request.
    print(request.args)
    return render_template('HTTP_GET_Method.html')

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

```html
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <form action="/" method="GET">

        <!-- Note: The for attribute of the label must match the name attribute of the input/radiobutton/etc. -->
        <label for=FirstName> Enter your first name:</label> <br>
        <input type="text" name="FirstName" placeholder="First Name"> <br>

        <br>

        <label for=LastName> Enter your last name:</label> <br>
        <input type="text" name="LastName" placeholder="Last Name"> <br>

        <br>

        <input type="submit" value="submit">
    </form>
</body>
</html>
```

The HTML page looks like this:

If I were to type Rick for First Name and Lan for Last Name, I would see this in the URL:



Notice that the query is shown in the URL. Furthermore, this is what I see in the terminal:

```
 * Detected change in '/mnt/c/Users/rick/Desktop/CSCB20 Code/HTTP Get Method/HTTP_GET_METHOD.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 275-882-184
ImmutableMultiDict([('FirstName', 'Rick'), ('LastName', 'Lan')])
127.0.0.1 - - [06/Apr/2020 18:39:09] "GET /?FirstName=Rick&LastName=Lan HTTP/1.1" 200 -
```