**POSTGRE SQL**

| **S.No.** | **Code** | **Explanation/ Example** | **Flask\_SQLAlchemy example** |
| --- | --- | --- | --- |
|  | psql -U <username> | To start accessing Postgresql through the specified user after entering the password |  |
|  | CREATE DATABASE <name>; | To create a new database based on standard parameters |  |
|  | \c <database name> | To connect to a database |  |
|  | \d | To check the details of the existing database, relations etc |  |
|  | CREATE TABLE <name> (details of columns i.e., name and type separated by a ',' for different columns); | **Example**  CREATE TABLE flights (  id SERIAL PRIMARY KEY,  origin VARCHAR NOT NULL,  destination VARCHAR NOT NULL,  duration INTEGER NOT NULL  );  **Explanation**  SERIAL represents automatic generated numbers in a sequence (1,2,3,4,5….)  PRIMARY key represents the key that will be used to identify/ fetch the related data  VARCHAR represents variable character type where the number of characters are not known  NOT NULL represents that cannot be left blank for this specified column in the table  INTEGER represents number  CREATE TABLE passengers (  id SERIAL PRIMARY KEY,  name VARCHAR NOT NULL,  flight\_id INTEGER REFERENCES flights);  Here REFERENCES is used to make reference to data in another table. Primary key of other table is used for reference. | Create a class for creating a new table  class Flight(db.Model):      # Flight class to correspond with table name flights      \_\_tablename\_\_ = "flights"      id = db.Column(db.Integer, primary\_key=True)      origin = db.Column(db.String, nullable=False)      destination = db.Column(db.String, nullable=False)      duration = db.Column(db.Integer, nullable=False) |
|  | INSERT INTO <table name> (c1,c2,c3) VALUES (value against c1, value against c2, value against c3); | **Example**  INSERT INTO flights (origin, destination, duration) VALUES ('New York', 'London', 415); | *Importing data from csv file*  def main():      f = open("flights.csv")      reader = csv.reader(f)      for origin, destination, duration in reader:          flight = Flight(origin=origin, destination=destination, duration=duration)          db.session.add(flight)          print(f"Added flight from {origin} to {destination} lasting {duration} minutes.")      db.session.commit() |
|  | SELECT \* FROM <table name>; | This will fetch complete record of the table. | flights = Flight.query.all()      for flight in flights:          print(f"{flight.origin} to {flight.destination}, {flight.duration} minutes.") |
|  | SELECT <c1>, <c2> FROM <table name> | This will fetch the data from the specified columns and all rows within those columns  **Example**  SELECT origin, destination FROM flights | Flight.query.filter\_by(origin=’Paris’).all()  *if query based on id then (alternate):*  Flight.query.get(5) |
|  | SELECT \* FROM <table name> WHERE <c = value>;  Here you can also use >,< in place of =, where data is an integer/ serial  Apart from above Boolean values can also be used.  (AND, OR) | This will fetch data from a specified row  **Example:**  SELECT \* FROM flights WHERE id = 3  This will show data from row 3 (as id is auto generated therefore id # = row #)  SELECT \* FROM flights WHERE origin = ‘New York’  This will show all the rows that contain New York as origin  SELECT \* FROM flights WHERE duration >500;  SELECT \* From flights WHERE destination = ‘Paris’ AND duration >500;  SELECT \* From flights WHERE destination = ‘Paris’ OR duration >500; | *Boolean*  Flight.query.filter(Flight.origin != ’Paris’).all()  Flight.query.filter(and\_(Flight.origin == ‘Paris’, Flight.duration > 500)).all()  Flight.query.filter(or\_(Flight.origin == ‘Paris’, Flight.duration > 500)).all() |
|  | SELECT AVG(c) FROM <table name>;  here you can replace is other standard operations like SUM, MIN, MAX, COUNT etc  you can also merge this with conditions on table | Used for performing different calculations through SELECT query  **Example:**  SELECT MIN(duration) FROM flights;  SELECT AVG(duration) FROM flights WHERE destination = ‘Paris’ AND duration > 500;  SELECT COUNT(\*) FROM flights WHERE origin = ‘New York’;  This will count the number of instances where origin is New York |  |
|  | SELECT \* FROM <table name> WHERE <c> IN (value1, value2); | Defining range of possible values.  **Example:**  SELECT \* FROM flights WHERE origin IN (‘New York’, ‘Moscow’);  In the above example complete data (row) for flights where origin is either New York or Moscow will be displayed | Flight.query.filter(Flight.origin.in\_([‘Tokyo’,’Paris’])).all() |
| . | SELECT \* FROM <table name> WHERE <c> LIKE ‘%<single or multiple alphabets>%’; | This will fetch the results which match the criteria provided based on data inside % sign, where the search will autocomplete.  **Example:**  SELECT \* FROM flights WHERE origin LIKE ‘%a%’;  Here all rows will be fetched where the data in origin column contains a. | Flight.query.filter(Flight.origin.like(%a%)).all() |
|  | UPDATE <table name> SET <c> = <new value> WHERE <other c> = <corresp. value> AND <another c> = <corresp. value>  [for updating multiple values]  UPDATE <table name> SET <c> = <new value>, <c>=<new value> WHERE <c> = <value> | Update a specific data which matches the selection.  **Example:**  UPDATE flights SET duration =430 WHERE origin = ‘New York’ AND destination = ‘London’;  For updating multiple values based on a parameter:  UPDATE flights SET duration = 425, origin = ‘Karachi’ WHERE id = 2;  This will update duration to 425 and origin to Karachi where id is 2 | flight = Flight.query.get(4)  flight.duration = 280  db.session.commit() |
|  | DELETE FROM <table name> WHERE <c> = <value> | Delete a row based on the defined parameters  **Example:**  DELETE FROM flights WHERE origin = ‘Karachi’; | flight = Flight.query.get(6)  db.session.delete(flight) |
|  | SELECT \* FROM <table name> LIMIT <number>; | This will fetch all the data from the top n (number) from the table  **Example:**  SELECT \* FROM flights LIMIT 2;  This will display top 2 rows from the table only. | Flight.query.filter\_by(origin = ‘Paris’).first() |
|  | SELECT \* FROM <table name> ORDER BY <c> ASC; | This will display data in an ascending order, based on the defined parameter.  **Example:**  SELECT \* FROM flights ORDER BY duration ASC;  This query will display results in an ascending order based on duration of each flight  SELECT \* FROM flights ORDER BY duration DESC;  This query will display results in an descending order based on duration of each flight  SELECT \* FROM flights ORDER BY duration ASC LIMIT 2;  This query will display 2 flights with the shortest duration | Flight.query.order\_by(Flight.origin).all()  Flight.query.order\_by(Flight.origin.desc()).all() |
|  | SELECT <c>, COUNT(\*) FROM <table name> GROUP BY <c> | Fetching details as to how common a particular data is  **Example:**  SELECT origin, COUNT(\*) FROM flights GROUP BY origin;  This query, based on origin will count flights and group them by origin. |  |
|  | SELECT <c>, COUNT(\*) FROM <table name> GROUP BY <c> HAVING COUNT(\*) >1; | **Example:**  SELECT origin, COUNT(\*) FROM flights GROUP BY origin HAVING COUNT(\*)>1;  This is fetch data where flights has more than 1 similar origin, say 2 flights from Karachi and 1 flight from Lahore and Islamabad, then this query will show Karachi |  |
|  | SELECT <c>,<c>,<c> FROM <table name1> JOIN <table name2> ON <table name1>.<c>=<table name2>.<c> | **Foreign Keys**  Referencing keys from different tables i.e., creating a connection between tables.  Here c after SELECT represents column from 2 or more different tables which are somehow linked (foreign keys) and the c after ON represents the column based on which the two tables are linked.  When we type only ‘JOIN’ this means inner Join, i.e., fetch the things that match.  **Example:**  SELECT origin, destination, name FROM flights JOIN passengers ON passengers.flight\_id = flights.id;  Here the query will fetch data regarding all the passengers and display their origin and destination of flights. [***Background*** a separate table of passengers is created where only names and flight id (as per flights table) is inserted in the table and based on the given query psql fetched data from both the tables and produce meaningful results]  SELECT origin, destination, name FROM flights JOIN passengers ON passengers.flight\_id=flights.id WHERE name = ‘Adnan’;  This will display origin and destination of flight for passenger Adnan.  SELECT origin, destination, name FROM flights LEFT JOIN passengers ON passengers.flight\_id = flights.id;  This query will result in data where there are flights with passengers / no passengers (all flights). Matching data will be displayed first.  We can also use RIGHT JOIN for matching data from right side. | db.session.query(Flight, Passenger).filter(Flight.id == Passenger.flight\_id).all() |
|  | CREATE INDEX <index\_name> ON <table\_name> | Index can be created on a specific column where a column will be widely used for referring data from that table and this will make it easy / speed up the process of selection.  After creation of an index, if data is updated/ new data is inserted, index also needs to be updated.  Indexing will occupy space and take time also. |  |
|  |  | Nested queries:  **Example:**  SELECT \* FROM flights WHERE id IN (SELECT flight\_id FROM passengers GROUP BY flight\_id HAVING COUNT(\*) >1); |  |