Connecting python to existing PostgreSQL database

```
import psycopg2 as pg2
# A library that allows Python to connect to an existing PostgreSQL
database to utilize SQL functionality.

conn = pg2.connect(database='Airline ',
user='postgres',password='password',host='localhost', port=5432)
# Create a connection with PostgreSQL

cur = conn.cursor() # Establish connection and start cursor to be
ready to query

cur.execute("SELECT * from my2007") # Pass in a PostgreSQL query as a
string
cur.fetchmany(5) # Return a tuple of the first 5 rows as Python
objects

[(datetime.time(12, 32),
  datetime.time(12, 25),
  datetime.time(13, 41),
  datetime.time(13, 40),
  'WN',
  '2891',
  'N351',
  69,
  75,
  1,
  7,
  'SMF',
  'ONT',
  datetime.datetime(2007, 1, 1, 12, 25)),
 (datetime.time(19, 18),
  datetime.time(19, 5),
  datetime.time(20, 43),
  datetime.time(20, 35),
  'WN',
  '462',
  'N370',
  85,
  90,
  8,
  13,
  'SMF',
  'PDX',
  datetime.datetime(2007, 1, 1, 19, 5)),
 (datetime.time(22, 6),
  datetime.time(21, 30),
  datetime.time(23, 34),
```

```
 datetime.time(23, 0),
 'WN',
 '1229',
 'N685',
 88,
 90,
 34,
 36,
 'SMF',
 'PDX',
 datetime.datetime(2007, 1, 1, 21, 30)),
(datetime.time(12, 30),
 datetime.time(12, 0),
 datetime.time(13, 56),
 datetime.time(13, 30),
 'WN',
 '1355',
 'N364',
 86,
 90,
 26,
 30,
 'SMF',
 'PDX',
 datetime.datetime(2007, 1, 1, 12, 0)),
(datetime.time(8, 31),
 datetime.time(8, 30),
 datetime.time(9, 57),
 datetime.time(10, 0),
 'WN',
 '2278',
 'N480',
 86,
 90,
 -3,
 1,
 'SMF',
 'PDX',
 datetime.datetime(2007, 1, 1, 8, 30))]
```

Note on using python and SQL

The code above is an example of **EMBEDDED SQL**. **Embedded SQL** can be understood as small SQL queries put into high-level languages to get meaningful outputs. With the help of the embedding of queries, database can be easily accessed without creating any bulky code. Embedded SQL can be used to create APIs which can easily fetch and feed data as and when required.

Leveraging Python's features for Intial Data Exploration

```
import pandas as pd
from sqlalchemy import create_engine
# connection string: driver://username:password@server/database
engine =
create_engine('postgresql+psycopg2://postgres:password@localhost/Airli
ne ')

def get_columns(tablename):
    return pd.read_sql('''select ordinal_position, column_name,
data_type
                                from information_schema.columns
                                where table_name = '{}'
                            ''' .format(tablename)
                            ,engine)
```

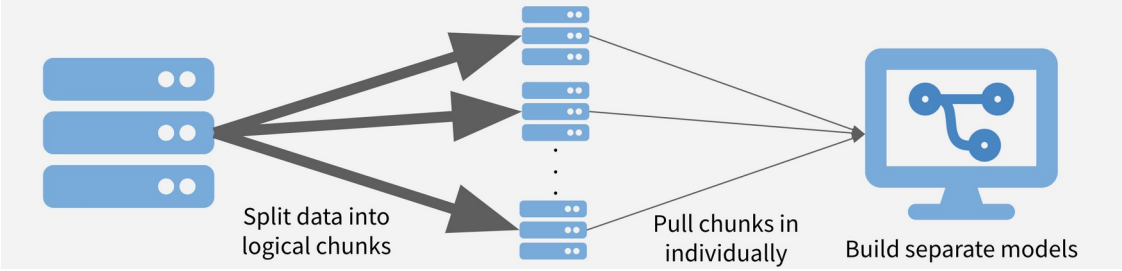
pandas.read_sql() is used to turn a SQL query into a DataFrame.

```
get_columns('my2007')
```

	ordinal_position	column_name	data_type
0	1	year	text
1	2	month	text
2	3	dayofmonth	text
3	4	dayofweek	text
4	5	deptime	text
5	6	crsdeptime	text
6	7	arrtime	text
7	8	crsarrtime	text
8	9	uniquecarrier	text
9	10	flightnum	text
10	11	tailnum	text
11	12	actualelapsedtime	text
12	13	crselapsedtime	text
13	14	airtime	text
14	15	arrdelay	text
15	16	depdelay	text
16	17	origin	text
17	18	dest	text
18	19	distance	text
19	20	taxiin	text
20	21	taxiout	text
21	22	cancelled	text
22	23	cancellationcode	text
23	24	diverted	text
24	25	carrierdelay	text
25	26	weatherdelay	text
26	27	nasdelay	text
27	28	securitydelay	text
28	29	lateaircraftdelay	text

```
def process_sql_using_pandas():
    engine = create_engine(
        'postgresql+psycopg2://postgres:password@localhost/Airline '
    )
    conn = engine.connect().execution_options(
        stream_results=True)
    for chunk_dataframe in pd.read_sql(
            "SELECT * FROM my2007", conn, chunksize=1000):
        print(f"Got dataframe w/{len(chunk_dataframe)} rows")

#process_sql_using_pandas()
```



If ***pd.read_sql()*** is used, all rows in the database are loaded **4 times**.

1. dbapi_cursor.fetchall() retrieves all the rows.
2. SQLAlchemy does some sort of additional manipulation involving the rows.
3. Pandas converts data into tuples.
4. Pandas converts some data (the tuples?) into arrays.

A function ***process_sql_using_pandas()*** is defined above that tell SQLAlchemy to use server-side cursors, aka streaming. Instead of loading all rows into memory, it will only load rows from the database when they're requested by the user. This batch processing method saves memory by a huge factor.

Note on exception handling

Even though the above commands was successful, to protect the integrity of the databases during Insert, Update or Deletion operations, future embedded SQL commands should include exception handling. **Rollback()** on transaction will be performed if an error is encountered, if there are none then **commit()** the changes.

The code below shows the template for future SQL queries.

```
try:
     cur.execute('''SELECT * FROM my2000''')
except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()
```

```
relation "my2000" does not exist
LINE 1: SELECT * FROM my2000
                      ^
```

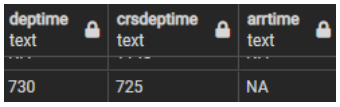
A transaction is a unit of work that is performed against a database. If no exception handling is handled when an incorrect query is made to the database, the database will throw an exception and the connection to the database will be terminated.

Data Cleaning Part 1: Dropping NULL and NA values

```
cur.execute('''Select count(*) from my2007 WHERE deptime ilike 'NA'
''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE depdelay ilike 'NA'
''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE arrtime ilike 'NA'
''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE arrdelay ilike 'NA'
''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE arrdelay = '0' ''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE depdelay = '0' ''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE cancellationcode IS
NOT NULL''')
print(cur.fetchall())
```

```
[(160748,)]
[(160748,)]
[(177927,)]
[(177927,)]
[(203365,)]
[(626239,)]
[(160749,)]
```

There is a mixture of NULL and 'NA' values present in the tables. It is clear that for cancelled flights (where rows have a non-null value for cancellationcode), no information about the dep, arr and delay times are recorded. Why is it then, that there are more NA values for arrival times(177927) then departure times(160748)?



deptime text	crsdeptime text	arrtime text
730	725	NA

Ghost Flights

Ghost flights are flights that are flown with no to minimal (less then 10%) passenger capacity. They either are flown to refuel, or to keep a hold of the flight slots that are allocated to a airline for each season. Thus, arrival destinations and arrtime are ommitted. Either that or the plane has crashed. These rows will be removed from the dataset.

```
try:
     for table in ['my1987','my2006', 'my2007']:
      cur.execute('''DELETE FROM {} WHERE arrtime ILIKE 'NA'
'''.format(table))
except Exception as e:
      conn.rollback()
      print(e)
else:
     conn.commit()
```

```
cur.execute('''Select count(*) from my2007 WHERE deptime ilike 'NA'
''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE depdelay ilike 'NA'
''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE arrtime ilike 'NA'
''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE arrdelay ilike 'NA'
''')
print(cur.fetchall())
cur.execute('''Select count(*) from my2007 WHERE cancellationcode IS
NOT NULL''')
print(cur.fetchall())
```

```
[(0,)]
[(0,)]
[(0,)]
[(0,)]
[(1,)]
```

ALL the 'NA' values have been removed.

```
try:
      cur.execute('''DELETE FROM plane WHERE year ILIKE 'None' ''')
      cur.execute('''DELETE FROM plane WHERE year is NULL''')
      cur.execute('''DELETE FROM plane WHERE year LIKE '0%' ''')
except Exception as e:
      conn.rollback()
      print(e)
else:
     conn.commit()
```

The year that the airplane started working is the most important variable for question 2, analysis of whether the age of the plane will affect delay. Thus, all null values of the year row are deleted.

Data Cleaning Part 2: Dropping unnecessary columns

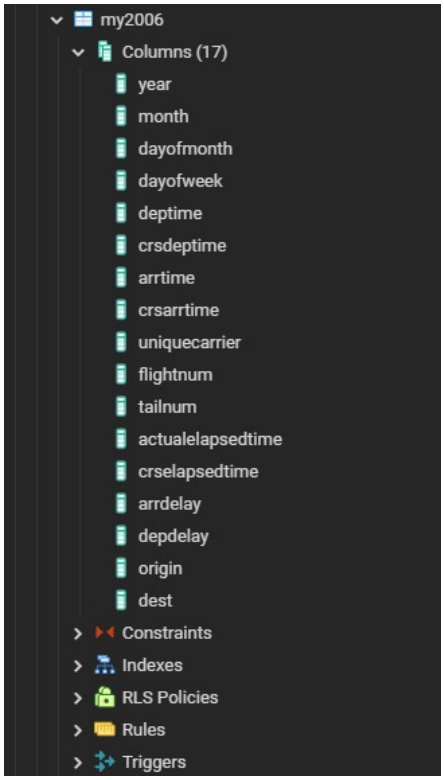
We shall drop any variables which are not relevant in our analysis of aircraft delays.

```
to_drop = ['airtime',
'distance','taxiin','taxiout','cancelled','cancellationcode','diverted
',
           'carrierdelay','weatherdelay','nasdelay','securitydelay',
'lateaircraftdelay']

try:
    for table in ['my1987','my2006', 'my2007']:
       drop_command = 'ALTER TABLE ' + table + ' '
       for column in to_drop:
        drop_command = drop_command + ' DROP ' + column + ","
       drop_command = drop_command[:-1]
       cur.execute(drop_command)
except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()

get_columns('my2007')
```

	ordinal_position	column_name	data_type
0	1	year	text
1	2	month	text
2	3	dayofmonth	text
3	4	dayofweek	text
4	5	deptime	text
5	6	crsdeptime	text
6	7	arrtime	text
7	8	crsarrtime	text
8	9	uniquecarrier	text
9	10	flightnum	text
10	11	tailnum	text
11	12	actualelapsedtime	text
12	13	crselapsedtime	text
13	15	arrdelay	text
14	16	depdelay	text
15	17	origin	text
16	18	dest	text



By calling the ***get_columns()*** function that we defined earlier and checking our pgAdmin GUI, we can see that the alter command worked successfully.

```
to_drop = ['index', 'type', 'manufacturer', 'issue_date', 'model',
'status', 'aircraft_type', 'engine_type']
try:
       drop_command = 'ALTER TABLE plane '
       for column in to_drop:
        drop_command = drop_command + ' DROP ' + column + ","
       drop_command = drop_command[:-1]
       cur.execute(drop_command)
except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()
```

Data Cleaning Part 3

Data type casting and transformation

The data type for every column in all tables are of type text. Does this make sense? PostgreSQL offers a variety of data types for date and time, including *DATE*, *DATETIME*,

TIMESTAMP and *YEAR*. To allow better operations on date and time variables, these text variables shall be converted into their appropiate variable type.

```
try:
    for table in ['my2006', 'my2007', 'my1987']:
      cur.execute('''ALTER TABLE {} ALTER COLUMN arrdelay TYPE integer
USING arrdelay::integer,
                                    ALTER COLUMN depdelay TYPE integer
USING depdelay::integer,
                                    ALTER COLUMN actualelapsedtime
TYPE integer USING actualelapsedtime::integer,
                                    ALTER COLUMN crselapsedtime TYPE
integer USING crselapsedtime::integer
                                    '''.format(table))
except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()
```

Time duration variables such as arrdelay, depdelay, actualelapsedtime and crselapsedtime are converted from text to integer. This is so we can peform arithemetic operations on them easily, such as ***sum()*** and ***avg()***.

```
try:
    for table in ['my1987','my2006', 'my2007']:
      cur.execute('''UPDATE {} SET arrtime = LPAD(arrtime, 4, '0'),
                                    crsarrtime = LPAD(crsarrtime, 4,
'0'),
                                    deptime = LPAD(deptime, 4, '0'),
                                    crsdeptime = LPAD(crsdeptime, 4,
'0')'''.format(table))

except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()


try:
     for table in ['my1987','my2006', 'my2007']:
       cur.execute('''DELETE FROM {} WHERE arrtime::integer >2400 OR
deptime::integer>2400 '''.format(table))
       cur.execute('''ALTER TABLE {} ALTER COLUMN arrtime TYPE time
USING (arrtime::time),
                                     ALTER COLUMN deptime TYPE time
USING (deptime::time),
                                     ALTER COLUMN crsarrtime TYPE time
USING (crsarrtime::time),
```

```
                                        ALTER COLUMN crsdeptime TYPE time
USING (crsdeptime::time)'''.format(table))
except Exception as e:
       conn.rollback()
       print(e)
else:
      conn.commit()
```

Arrival,Departure times and both the actual and scheduled times are converted from text to time variables.

```
try:
    for table in ['my1987','my2006', 'my2007']:
      cur.execute('''ALTER TABLE {} ADD COLUMN departure_date
TIMESTAMP '''.format(table))
      cur.execute('''UPDATE {} SET departure_date =
make_date(year::int, month::int, dayofmonth::int) + crsdeptime
'''.format(table))
except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()
```

Instead of having 4 different columns (year, month, dayofmonth and crsdeptime) represent the datetime of the scheduled departure of the plane, these 4 text variables are concantenated into one time stamp variable called scheduled_departure.

Why is the same not performed for scheduled arrival? It is unclear what is the date of the scheduled arrival.

```
to_drop = ['year', 'month', 'dayofmonth', 'dayofweek']

try:
    for table in ['my1987','my2006', 'my2007']:
       drop_command = 'ALTER TABLE ' + table + ' '
       for column in to_drop:
        drop_command = drop_command + ' DROP ' + column + ","
       drop_command = drop_command[:-1]
       cur.execute(drop_command)
except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()
```

These date variables can now be dropped. For future analysis using R, many functions to extract these variables are easily available from the lubridate library, such as ***year()***, ***month()*** and ***wday()***.

Data Cleaning Part 4: Readability of Data

```
try:
    for table in ['my1987','my2006', 'my2007']:
        cur.execute('''ALTER TABLE {} RENAME COLUMN crsdeptime TO
scheduled_departure_time'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN actualelapsedtime
TO travel_time'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN crselapsedtime TO
scheduled_travel_time'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN arrdelay TO
arrival_delay'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN depdelay TO
departure_delay'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN crsarrtime TO
scheduled_arrival'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN deptime TO
departure_time'''.format(table))
        #cur.execute('''ALTER TABLE {} RENAME COLUMN departure_date TO
scheduled_departure'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN uniquecarrier TO
airline'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN origin TO
origin_airport'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN dest TO
dest_airport'''.format(table))
        cur.execute('''ALTER TABLE {} RENAME COLUMN arrtime TO
arrival_time'''.format(table))

except Exception as e:
        conn.rollback()
        print(e)
else:
        conn.commit()
```

Columns are renamed for better readability.

```
try:
        cur.execute('''CREATE TABLE df
                       AS
                       SELECT * FROM my2006
                       UNION
                       SELECT * FROM my2007;''')
except Exception as e:
        conn.rollback()
        print(e)
else:
        conn.commit()
```

The two cleaned tables of data from 2006 and 2007 are then combined into a table called df.

Preparation for question 1

The season variable is added. This will help greatly for analysis of question 1, which asks "When is the best time of day, day of the week, and time of year to fly to minimise delays?". In this case, time of year is meant to mean season.

```
try:
      cur.execute('''ALTER TABLE df ADD COLUMN season TEXT
'''.format(table))
      cur.execute('''UPDATE df SET season = CASE
      WHEN (EXTRACT(MONTH FROM "scheduled_departure") IN (10, 11, 12))
THEN 'Fall'
      WHEN (EXTRACT(MONTH FROM "scheduled_departure") IN (1, 2, 3))
THEN 'Winter'
      WHEN (EXTRACT(MONTH FROM "scheduled_departure") IN (4, 5, 6))
THEN 'Spring'
      ELSE 'Summer'
      END''')
except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()
```

Removing outliers from the database

Why remove outliers from the database instead of plotting boxplots and removing outlier data in R or python? The same issue arises: Our dataset is too big to pull into R's memory. SQL also offers stastical functions such as ***STDDEV_SAMP()*** for caculating standard deviation.

```
try:
     cur.execute(''' DELETE FROM df WHERE departure_delay NOT BETWEEN
     (SELECT AVG(departure_delay) - STDDEV_SAMP(departure_delay) * 2
FROM df )
     AND (SELECT AVG(departure_delay) + STDDEV_SAMP(departure_delay) *
2 FROM df )
                      ;  ''')
except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()
```

Preparation for question 2

For question 2 which asks "Do older planes suffer more delays?", we only need the delay times, the year of the plane and the scheduled departure.

```
try:

     cur.execute(''' CREATE TABLE dff as
                     SELECT departure_delay, arrival_delay,
scheduled_departure, plane.year
                     FROM
                     df INNER JOIN
                     plane
                     ON df.tailnum = plane.tailnum''')

     cur.execute(''' DELETE
                     FROM dff
                     where EXTRACT(year FROM dff.scheduled_departure)
< CAST("year" AS INTEGER) ''')

except Exception as e:
     conn.rollback()
     print(e)
else:
     conn.commit()
```

Preparation for question 3

For question 3 which asks "How does the number of people flying between different locations change over time?", Only the departure date and the coordinates of the destination are needed.

```
try:
     cur.execute(''' CREATE TABLE dfff as
                     SELECT scheduled_departure as Date, lat, long
                     FROM
                     my1987 INNER JOIN
                     airports
                     ON my1987.dest_airport = airports.iata''')

     cur.execute('''ALTER TABLE dfff ALTER COLUMN lat TYPE double
precision USING (lat::double precision),
                                     ALTER COLUMN long TYPE double
precision USING (long::double precision),
                                     ALTER COLUMN Date TYPE date USING
(Date::date)''')

except Exception as e:
     conn.rollback()
```

```
        print(e)
else:
        conn.commit()
```

SUMMARY

In this notebook, the dataset was cleaned and explored. Null and Na values were removed, and the data types were change into their more appropiate form. Different tables were joined in preparation for questions 1 - 4.