Generating_Fake_Data_in_Python

November 27, 2020

1 Generating Fake Data in Python

1.0.1 Import the Relevant Libraries

[1]:
```python
from numpy.random import randint
from faker import Faker
```

1.0.2 Create an Instance of Faker

[2]:
```python
fake = Faker()
```

1.0.3 Define Classes for Each MySQL Table

We will be generating 10,000 fake users whose Twitter handles are their names concatenated. We assume that every name will be unique, and thus every handle will be unique. The email addresses will be the handles @ a certain random email provider, with a bias towards @gmail.com.

Each MySQL table will have its own unique class below. The **init** method will initialize each instance of the class with each of the features of the MySQL table. We will then iterate over a list of these objects to generate a list of tuples that will then be converted into a string that will be part of the MySQL multi-line insert query for each table.

Note: While experimenting with the number of possible unique hashtags, I discovered that the number of unique words generated from fake.text() did not exceed 960. This is probably a limitation of the Faker library.

[3]:
```python
class User:
    def __init__(self):
        # Generate a fake name for the user with Faker
        self.full_name = fake.name()

        # Use the fake name to create user_handle, first_name, last_name␣
 ↪attributes
        self.user_handle = "".join(self.full_name.split())
        self.first_name = self.full_name.split()[0]
        self.last_name = self.full_name.split()[1]

        # Create email_address attribute
        n = randint(1,5)
        if n==1 or n==2:
            self.email_address = self.user_handle+'@gmail.com'
        elif n==3:
            self.email_address = self.user_handle+'@yahoo.com'
        else:
            self.email_address = self.user_handle+'@hotmail.com'

        # Create phone_number attribute
        self.phone_number = str(randint(2000000000,9999999999))

        # Create birthday attribute
        self.birthday = str(fake.date_of_birth(minimum_age=13,maximum_age=50))

class Follower:
    def __init__(self):
        # A user cannot follow his/herself, thus the validation while loop␣
 ↪below.
        # This could also be validated on the client side.
        while True:
            x = randint(1,10001)
            y = randint(1,10001)
            if x==y:
                pass
            else:
                break

        self.follower_id = x
        self.following_id = y

class Tweet:
    def __init__(self):
        self.user_id = randint(1, 10001)

        random_tweet = fake.text()
        if len(random_tweet) > 280:
            random_tweet = random_tweet[0,279]
        self.tweet_text = random_tweet

class BaseComment:
    def __init__(self):
        self.op_tweet_id = randint(1, 100001)
        self.commenter_id = randint(1, 10001)

        random_text = fake.text()
        if len(random_text) > 280:
            random_text = random_text[0,279]
        self.comment_text = random_text
```

```
class SubComment:
    def __init__(self):
        self.parent_comment_id = randint(1, 100001)
        self.op_tweet_id = randint(1, 100001)
        self.commenter_id = randint(1, 10001)

        random_text = fake.text()
        if len(random_text) > 280:
            random_text = random_text[0,279]
        self.comment_text = random_text

class Retweet:
    def __init__(self):
        self.op_tweet_id = randint(1,100001)
        self.retweeter_id = randint(1, 10001)

class RetweetedComment:
    def __init__(self):
        self.comment_id = randint(1, 300001)
        self.op_tweet_id = randint(1,100001)
        self.retweeter_id = randint(1, 10001)

class TweetLike:
    def __init__(self):
        self.user_id = randint(1,10001)
        self.tweet_id = randint(1,100001)

class CommentLike:
    def __init__(self):
        self.comment_id = randint(1,300001)
        self.commenter_id = randint(1,10001)
```

1.0.4 Generate the Random Data

We will use nested list comprehensions to generate the fake data. The MySQL syntax for inserting data is INSERT INTO table VALUES (data,data,data);

In order to create the data in this format, we first instantiate a list of objects from each class above via the following: [Class() for j in range(number)]

We then iterate over this list of objects to retrieve the attributes for each object, organized in tuples, so: our_list = [(attribute1, attribute2) for i in [Class() for j in range(number)]].

Please note that this code may take several minutes to run, as it is generating 925,000 values.

[11]:
```
# Generate a list of 10,000 Users for the users table:
user_data = [(i.user_handle, i.first_name, i.last_name, i.email_address, i.
 ↪phone_number, i.birthday) for i in [User() for j in range(10000)]]
```

```python
# Generate a list of unique following interactions for the followers table, to be compliant with the MySQL primary key restrictions.
followers_data = set([(i.follower_id, i.following_id) for i in [Follower() for j in range(100000)]])

# Generate a list of 100,000 random tweets by random users:
tweets = [(i.user_id, i.tweet_text) for i in [Tweet() for j in range(100000)]]

# Generate a list of 100,000 random comments by random users:
base_comments = [(i.op_tweet_id, i.commenter_id, i.comment_text) for i in [BaseComment() for j in range(100000)]]

# Generate a list of 200,000 random subcomments by random users:
sub_comments = [(i.parent_comment_id, i.op_tweet_id, i.commenter_id, i.comment_text) for i in [SubComment() for j in range(200000)]]

# Generate a list of 10,000 retweets
retweets = [(i.op_tweet_id, i.retweeter_id) for i in [Retweet() for j in range(10000)]]

# Generate a list of 5,000 retweeted comments (can be retweeted subcomment)
retweeted_comments = [(i.op_tweet_id, i.comment_id, i.retweeter_id) for i in [RetweetedComment() for j in range(50000)]]

# Generate a set of unique tweet likes to satisfy the MySQL primary key requirements.
# Note - This is because a user cannot like a tweet more than once
tweet_likes = set([(i.user_id, i.tweet_id) for i in [TweetLike() for j in range(200000)]])

# Generate a set of unique comment likes to satisfy the MySQL primary key requirements (includes likes on subcomments)
comment_likes = set([(i.comment_id, i.commenter_id) for i in [CommentLike() for j in range(200000)]])
```

1.0.5 Create the hashtag_list and hashtag_instances data

Due to Python syntax where tuples with only one entry must be written as (entry,) we can't use the same list comprehension methodology as before but instead must use the following for loop to generate the hasthag_list. The hashtag_instances data can be created with the comprehensions like the others, though, because they contain two entries in the tuples. But the hashtag_instances class must be defined following the hashtag_list because its **init** method depends on the length of hashtag_list.

```
[5]: class Hashtag:
         def __init__(self):
             self.hashtag_name = fake.text().split()[1] # Choose index 1 because the␣
      ↪word at index 0 is always capitalized

     # Generate the list of unique hashtags in hashtag_list
     hashtag_list = list(set([(j.hashtag_name) for j in [Hashtag() for k in␣
      ↪range(4000)]]))

     # Iterate over the list to create the string query for MySQL
     query_hashtag_list = "INSERT INTO hashtag_list (hashtag_name) VALUES "
     for x in hashtag_list:
         query_hashtag_list += f"('{x}'), "

     # We need to remove the last comma before placing the ; delimiter
     query_hashtag_list = query_hashtag_list[:-2] + "; \n\n"

     class HashtagInstance:
         def __init__(self):
             self.hashtag_id = randint(1,len(hashtag_list)+1)
             self.tweet_id = randint(1,100001)

     # Generate a list of 20,000 hashtag instances
     hashtag_instances = [(i.hashtag_id, i.tweet_id) for i in [HashtagInstance() for␣
      ↪j in range(20000)]]
```

1.0.6 Create the MySQL INSERT Queries

For each list we created (except for the hashtag_list, whose query is already written), we now convert each list of tuples into a string and slice it to remove the brackets: str(our_list)[1:-1]. This gives us a string containing only the tupled values we want for the MySQL insert. This is concatenated with the relevant INSERT INTO query syntax. All of the INSERT INTO queries for each table are then concatenated to form one large string, which is then written to a text file for future reference. We are writing the data to text files to store them permanently because this Python script will generate a new list of random data each time we run it.

```
[12]: query_users = "INSERT INTO users (user_handle, first_name, last_name,␣
       ↪email_address, phone_number, birthday) VALUES " + str(user_data)[1:-1] + ";␣
       ↪\n\n"
      query_followers = "INSERT INTO followers (follower_id, following_id) VALUES " +␣
       ↪str(followers_data)[1:-1] + "; \n"
      query_tweets = "INSERT INTO tweets (user_id, tweet_text) VALUES " +␣
       ↪str(tweets)[1:-1] + "; \n\n"
      query_base_comments = "INSERT INTO comments (op_tweet_id, commenter_id,␣
       ↪comment_text) VALUES " + str(base_comments)[1:-1] + "; \n\n"
      query_sub_comments = "INSERT INTO comments (parent_comment_id, op_tweet_id,␣
       ↪commenter_id, comment_text) VALUES " + str(sub_comments)[1:-1] + "; \n\n"
```

```
query_retweets = "INSERT INTO retweets (op_tweet_id, retweeter_id) VALUES " +␣
↪str(retweets)[1:-1] + "; \n\n"
query_retweeted_comments = "INSERT INTO retweets (op_tweet_id, comment_id,␣
↪retweeter_id) VALUES " + str(retweeted_comments)[1:-1] + "; \n\n"
query_hashtag_instances = "INSERT INTO hashtag_instances (hashtag_id, tweet_id)␣
↪VALUES " + str(hashtag_instances)[1:-1] + "; \n\n"
query_tweet_likes = "INSERT INTO tweet_likes (user_id, tweet_id) VALUES " +␣
↪str(tweet_likes)[1:-1] + "; \n\n"
query_comment_likes = "INSERT INTO comment_likes (comment_id, commenter_id)␣
↪VALUES " + str(comment_likes)[1:-1] + "; \n\n"

total_query = query_users + query_followers + query_tweets +␣
↪query_base_comments + query_sub_comments + query_retweets +␣
↪query_retweeted_comments + query_hashtag_list + query_hashtag_instances +␣
↪query_tweet_likes + query_comment_likes
```

1.0.7 Write the MySQL query to a text file for storage

The following lines of code will generate separate text files for the data to be inputted into each table, as well as all data compiled into a single file called ALL_DATA.txt. You can then inspect the data for yourself in each individual file. When ready, you can convert them into .sql files and run them directly in MySQL, as long as you start each file with the query USE mock_twitter_db; followed by the data.

[13]:
```
file1 = open("users.txt","w")
file1.write(query_users)
file1.close()

file2 = open("followers.txt","w")
file2.write(query_followers)
file2.close()

file3 = open("tweets.txt","w")
file3.write(query_tweets)
file3.close()

file4 = open("base_comments.txt","w")
file4.write(query_base_comments)
file4.close()

file5 = open("sub_comments.txt","w")
file5.write(query_sub_comments)
file5.close()

file6 = open("retweets.txt","w")
file6.write(query_retweets)
file6.close()
```

```
file7 = open("retweeted_comments.txt","w")
file7.write(query_retweeted_comments)
file7.close()

file8 = open("hashtag_list.txt","w")
file8.write(query_hashtag_list)
file8.close()

file9 = open("hashstag_instances.txt","w")
file9.write(query_hashtag_instances)
file9.close()

file10 = open("tweet_likes.txt","w")
file10.write(query_tweet_likes)
file10.close()

file11 = open("comment_likes.txt","w")
file11.write(query_comment_likes)
file11.close()

total_query_file = open("ALL_DATA.txt","w")
total_query_file.write(total_query)
total_query_file.close()
```