SI 564 Final Project: Trails in U.S. National Parks

Haley Johnson
April 18, 2022

This document contains information about how I designed nat_parks.db and inserted data into the database. The original data was scraped from AllTrails in 2019 and is hosted on Kaggle. All the code and materials involved in this project are also available at this GitHub repository.

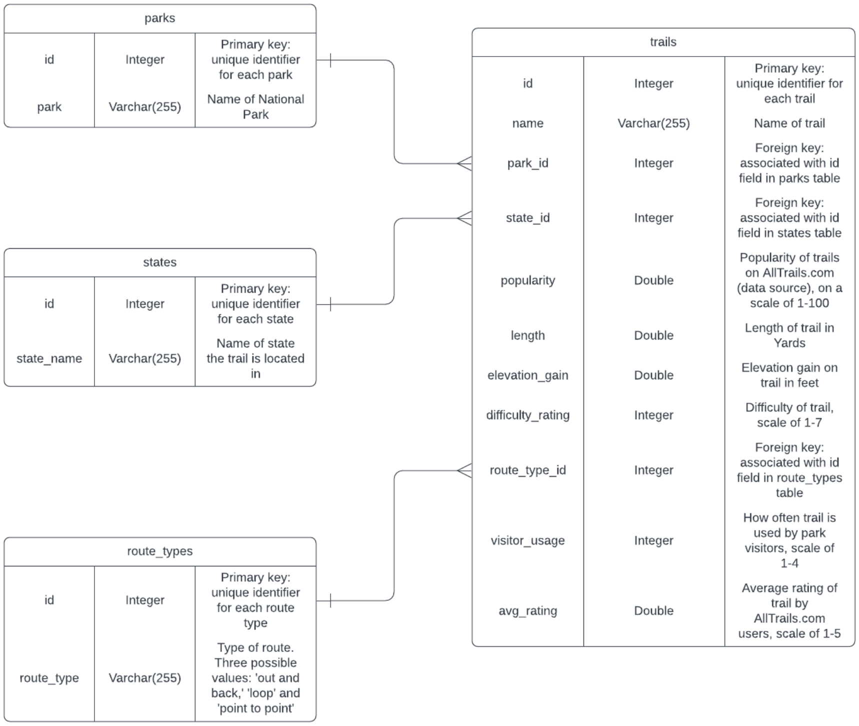
Schema

Before I began manipulating the data, I designed the schema for the final database. The original dataset is split out into four tables: parks, states, routes_types, and trails. The first three tables remove duplicate string data from trails, which is the main table in the dataset.

The original dataset included two columns called 'features' and 'activities,' which were lists of different attributes associated with each trail. For instance, 'features' might mention that a trail is dog-friendly and passed by water, while 'activities' might note that there are running trails or birding areas nearby. Since each trail was associated with multiple activities and features, properly normalizing this would've required me to create linking tables for each column that connected trail id's to feature id's or activity id's. This would've significantly expanded the scope of the project, so I opted to drop both of these columns. Retrospectively, it would've been worthwhile to keep at least one of these columns, since they allow users to ask more interesting questions about the database.

There were a few columns with location data about the trails. These included the country the trail was located in, the nearest city, and the exact latitude and longitude of the trailhead. These columns didn't add any information that was relevant for my purposes to the dataset, so they were removed.

This is the final database schema:



parks		
id	Integer	Primary key: unique identifier for each park
park	Varchar(255)	Name of National Park

states		
id	Integer	Primary key: unique identifier for each state
state_name	Varchar(255)	Name of state the trail is located in

route_types		
id	Integer	Primary key: unique identifier for each route type
route_type	Varchar(255)	Type of route. Three possible values: 'out and back,' 'loop' and 'point to point'

trails		
id	Integer	Primary key: unique identifier for each trail
name	Varchar(255)	Name of trail
park_id	Integer	Foreign key: associated with id field in parks table
state_id	Integer	Foreign key: associated with id field in states table
popularity	Double	Popularity of trails on AllTrails.com (data source), on a scale of 1-100
length	Double	Length of trail in Yards
elevation_gain	Double	Elevation gain on trail in feet
difficulty_rating	Integer	Difficulty of trail, scale of 1-7
route_type_id	Integer	Foreign key: associated with id field in route_types table
visitor_usage	Integer	How often trail is used by park visitors, scale of 1-4
avg_rating	Double	Average rating of trail by AllTrails.com users, scale of 1-5

Foreign Keys

The database is signed so that trails is the main table in the database. Trails contains the bulk of the information and columns in the database. The other three tables (parks, states, and route_types) can be seen as 'supporting tables.' These are designed to remove duplicate string data from trails. As a result, each of these tables contain just two columns: a primary key and a varchar(255) field.

These are the connections between tables:

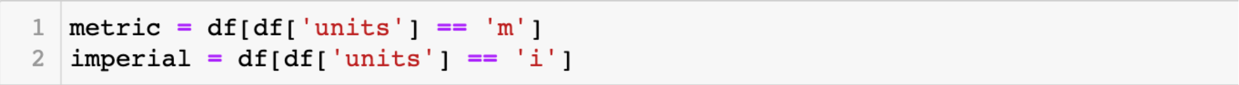
- The **trails** table links to the **parks** table on **trails.park_id = parks.id**
- The **trails** table links to the **states** table on **trails.state_id = states.id**
- The **trails** table links to the **route_types** table on **trails.route_type_id = route_types.id**

Manipulation

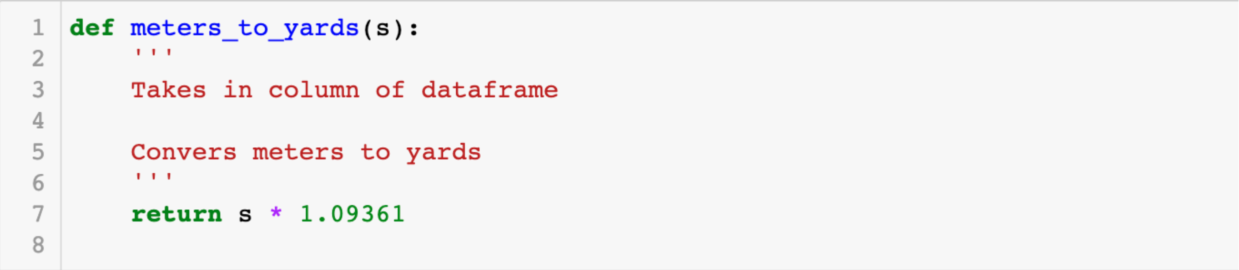
Before the data was ready to be split into normalized tables, I needed to make a few modifications to the dataset. I did all of my data manipulation and normalization in Jupyter Notebooks using Python.

First, there were a few trails in the original dataset that did not actually belong to National Parks. For instance, a trail in the 1996 Atlanta Olympic village was in the dataset. These entries were removed.

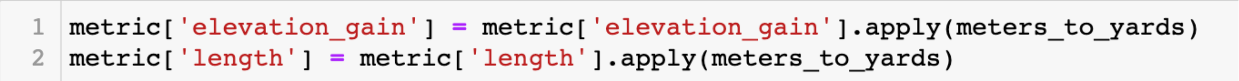
Similarly, a handful of trails had their length and elevation gain reported in metric units. To keep measurements consistent throughout the database, I converted all metric columns to imperial units.



```
metric = df[df['units'] == 'm']
imperial = df[df['units'] == 'i']
```



```
def meters_to_yards(s):
    '''
    Takes in column of dataframe

    Convers meters to yards
    '''
    return s * 1.09361

```

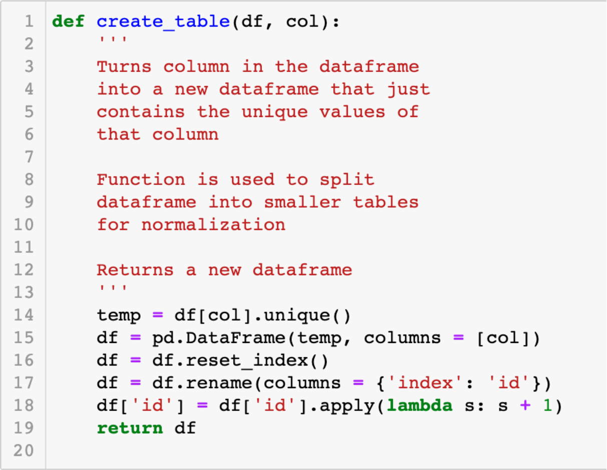


```
metric['elevation_gain'] = metric['elevation_gain'].apply(meters_to_yards)
metric['length'] = metric['length'].apply(meters_to_yards)
```

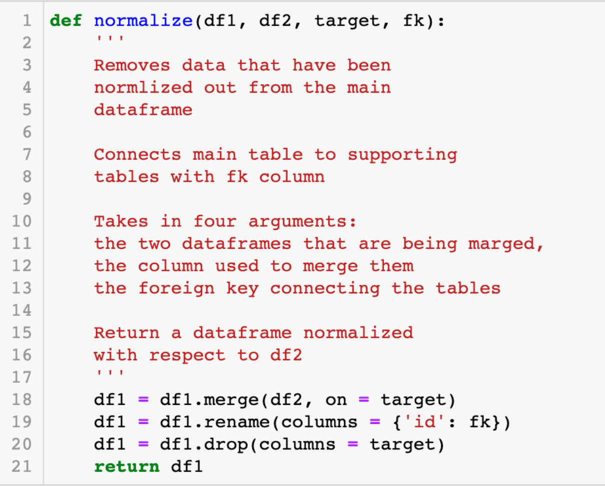
Normalization

I used two functions to normalize my dataset. The function ‘create_table’ separates the full dataset into tables. The function ‘normalize’ removes duplicate string data from a table and adds in the foreign key associated with the string data.

The functions and their docstrings are shown below:



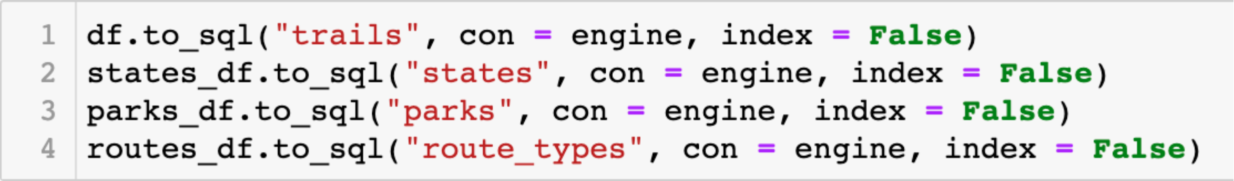
```
def create_table(df, col):
    '''
    Turns column in the dataframe
    into a new dataframe that just
    contains the unique values of
    that column

    Function is used to split
    dataframe into smaller tables
    for normalization

    Returns a new dataframe
    '''
    temp = df[col].unique()
    df = pd.DataFrame(temp, columns = [col])
    df = df.reset_index()
    df = df.rename(columns = {'index': 'id'})
    df['id'] = df['id'].apply(lambda s: s + 1)
    return df

```



```
def normalize(df1, df2, target, fk):
    '''
    Removes data that have been
    normlized out from the main
    dataframe

    Connects main table to supporting
    tables with fk column

    Takes in four arguments:
    the two dataframes that are being marged,
    the column used to merge them
    the foreign key connecting the tables

    Return a dataframe normalized
    with respect to df2
    '''
    df1 = df1.merge(df2, on = target)
    df1 = df1.rename(columns = {'id': fk})
    df1 = df1.drop(columns = target)
    return df1
```

Insertion

Tables were directly imported into DataGrip from my Jupyter Notebook.



```
df.to_sql("trails", con = engine, index = False)
states_df.to_sql("states", con = engine, index = False)
parks_df.to_sql("parks", con = engine, index = False)
routes_df.to_sql("route_types", con = engine, index = False)
```

Once the tables were in DataGrip, it was just a matter of manually adding in primary keys, foreign keys, and verifying that columns were the correct data types.