COMS W4111: Introduction to Databases
Spring 2024, Sections 002/V02

Homework 3

Introduction

- This notebook contains HW3. **Both Programming and Nonprogramming tracks should complete this homework.**
- You will submit **PDF and ZIP files** for this assignment. Gradescope will have two separate assignments for these.
- For the PDF:
    - The most reliable way to save as PDF is to go to your browser's menu bar and click `File -> Print`. Switch the orientation to landscape mode, and hit save.
    - **MAKE SURE ALL YOUR WORK (CODE AND SCREENSHOTS) IS VISIBLE ON THE PDF. YOU WILL NOT GET CREDIT IF ANYTHING IS CUT OFF.** Reach out for troubleshooting.
    - **MAKE SURE YOU DON'T SUBMIT A SINGLE PAGE PDF.** Your PDF should have multiple pages.
- For the ZIP:
    - Zip a folder containing this notebook and any screenshots.
    - You may delete any unnecessary files, such as caches.

Setup

In [ ]:

```
%load_ext sql
%sql mysql+pymysql://root:tzy123456@localhost
%sql SELECT 1
```

```
 * mysql+pymysql://root:***@localhost
1 rows affected.
```

Out[ ]:

1
1

In [ ]:

```
%%sql

drop schema if exists s24_hw3;
create schema s24_hw3;
use s24_hw3;
```

```
 * mysql+pymysql://root:***@localhost
4 rows affected.
1 rows affected.
0 rows affected.
```

Out[ ]:

```
[]
```

In [ ]:

```
import copy
import math

import pandas
import pymysql
from sqlalchemy import create_engine

sql_conn = pymysql.connect(
    user="root",
    password="tzy123456",
    host="localhost",
    port=3306,
    cursorclass=pymysql.cursors.DictCursor,
    autocommit=True
)
engine = create_engine("mysql+pymysql://root:tzy123456@localhost")

cur = sql_conn.cursor()
res = cur.execute("SELECT 1")
res = cur.fetchall()
res
```

```
Out[ ]: [{'1': 1}]
```

Written

- As usual, try to keep things short. Do not bloviate.
- You may use external resources, but you should cite your sources.

W1

Explain and list some differences between

- RAM
- Solid state drives
- Hard drives

`Answer`

- RAM
  - data that is being actively used or processed by the CPU.
  - Very fast access times compared to SSDs and HDDs.
  - loses all stored information when the device is turned off or restarted.
- Solid state drives
  - Non-volatile storage used for holding data more permanently.
  - Faster read/write speeds than HDDs because they use NAND, but slower than RAM.
  - retains data without power.
- Hard drives
  - Non-volatile storage used for storing large amounts of data relatively cheaply.
  - Slower than RAM and SSD.
  - retains data without power.

W2

With regards to disk drives, define

- Seek time
- Rotational latency time
- Transfer time/data transfer rate

`Answer`

- Seek time
    - the time it takes for the disk's read/write head to move to the correct track on the disk where the desired data is stored.
- Rotational latency time
    - the time it takes for the disk to rotate the correct sector of the disk under the read/write head.
- Transfer time/data transfer rate
    - The time it takes to read or write data once the read/write head is in the correct position and the data sector is under the head. This is after seek time and rotational latency time.

W3

Explain the concepts of

- Logical block addressing
- Cylinder-head-sector addressing

`Answer`

- Logical block addressing
    - Each block of data on the disk is assigned a unique address to specify. This is logical address instead of physical.
- Cylinder-head-sector addressing
    - An earlier method for specifying locations on a hard disk drive. The disk controller needs to know the cylinder (depth), the head (which platter surface), and the sector (the specific area of the track) to extract the targeted data.

W4

Define and list some benefits of

- Fixed-length records
- Variable-length records
- Row-oriented storage
- Column-oriented storage

`Answer`

- Fixed-length records
    - have a consistent size for all records.
    - Accessing a specific record is straightforward because the offset of any record can be easily calculated.
- Variable-length records
    - fields within a record to have varying lengths.
    - have flexibility, accommodating a wide range of data types and sizes.
- Row-oriented storage
    - store database information by rows, meaning that all data related to a specific record is stored together.
    - Easier to implement and understand, as data is stored in a way that closely reflects the structure of a typical table.
- Column-oriented storage
    - store data by columns rather than by rows.
    - Data in each column is often similar, allowing for more effective compression techniques.

W5

Explain and list some differences between

- RAID 0
- RAID 1
- RAID 5

`Answer`

- RAID 0
    - data is split into blocks and each block is written to a separate disk drive.
    - no redundancy. If one disk fails, all data in the array is lost.
- RAID 1
    - creates an exact copy of a set of data on two or more disks.
    - high redundancy due to the several copies.
- RAID 5
    - combines the techniques of RAID 0 and parity. Parity information is used to rebuild the data in case of a disk failure.
    - better read/write performance than RAID 1.Requires additional calculations for parity information.

SQL

Overview

- The `data` directory contains a file `People.csv`. The columns are
    - `nameFirst`
    - `nameLast`
    - `birthYear`
    - `birthCountry`
    - `deathYear`
    - `deathCountry`
- For Nonprogramming students, note that this `People.csv` differs from the one you loaded in HW2. Do not mix the two files.
- **There is no one right answer for this section.** You can come up with and document your own design (as long as they satisfy the requirements).

Create Table

- Create a table based on the structure of `People.csv`
    - **You must add an additional attribute, `personID`, which has type char(9)**
        - `personID` should be the primary key of your table
    - `nameFirst` and `nameLast` cannot be null. The other (non-PK) columns can be null.
    - You should choose reasonable data types for the attributes
        - Do not use the `year` data type for `birthYear` or `deathYear`. The range for year is too small.
    - Your table will be empty for the next few sections. We will insert data later.

In [ ]:

```
%%sql

create table People
(
    personID     char(9)     not null,
    nameFirst    varchar(32) not null,
    nameLast     varchar(32) not null,
    birthYear    int         null,
    birthCountry varchar(32) null,
    deathYear    int          null,
    deathCountry varchar(32) null,
    constraint People_pk
        primary key (personID)
);
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
```

```
Out[ ]: []
```

Person ID Function

- `personID` is formed using the following rules:

1. The ID consists of three sections: `[lastSubstr][firstSubstr][number]`
2. `lastSubstr` is formed by lowercasing `nameLast`, then taking the first 5 letters. If `nameLast` is less than 5 letters, use the entire `nameLast`.
3. `firstSubstr` is formed by lowercasing `nameFirst`, then taking the first 2 letters. If `nameFirst` is less than 2 letters, use the entire `nameFirst`.
4. For a specific combination of `[lastSubstr][firstSubstr]`, `number` starts from 1 and increments. `number` should be padded to have length 2.
5. `nameFirst` and `nameLast` may contain periods ".", hyphens "-", and spaces " ". You should remove these characters from `nameFirst` and `nameLast` **before** doing the above substring processing.

- As an example, starting from an empty table, below is what `personID` would be assigned to the following names (assuming they were inserted in the order that they are shown)

nameFirst	nameLast	personID
Donald	Ferguson	fergudo01
David	Aardsma	aardsda01
Doe	Fergue	fergudo02
J. J.	Park	parkjj01

- Write a SQL function that generates a person ID using the above rules
  - You should determine what parameters and return type are needed
  - This function will be called by triggers in the next section. **It is up to you which logic you put in the function and which logic you put in the triggers.**
    - That is, if you plan to place the bulk of your logic in your triggers, then your function could be a few lines.
  - You may define helper functions
  - You may add additional attributes to your table if it helps

```
In [ ]: %%sql

create function generatePersonID(nameFirst varchar(32), nameLast varchar(32)) returns char(9)
deterministic
begin
    declare first_cleaned varchar(32);
    declare first_initial varchar(32);
    declare last_cleaned varchar(32);
    declare last_initial varchar(32);
    declare id_count int;
    declare id_prefix varchar(32);
    declare result char(9);
    declare id_pattern varchar(32);

    set id_count = 0;

    set first_cleaned = replace(replace(replace(replace(nameFirst, ' ', ''), '-', ''), '.', ''), '\'', '');
    set last_cleaned = replace(replace(replace(replace(nameLast, ' ', ''), '-', ''), '.', ''), '\'', '');

    set first_initial = lower(substr(first_cleaned, 1, 2));
    set last_initial = lower(substr(last_cleaned, 1, 5));
    set id_prefix = concat(last_initial, first_initial);

    set id_pattern = concat(id_prefix, '%');

    select
        count(*) into id_count
    from People
    where personID like id_pattern;

    set result = concat(id_prefix, LPAD(cast((id_count+1) as char),2,'0'));

    return result;
end;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
```

Out[ ]: []

In [ ]:

```
%%sql

create function getOldPersonID(nameFirst varchar(32), nameLast varchar(32)) returns char(9)
deterministic
begin
    declare first_cleaned varchar(32);
    declare first_initial varchar(32);
    declare last_cleaned varchar(32);
    declare last_initial varchar(32);
    declare id_count int;
    declare id_prefix varchar(32);
    declare result char(9);
    declare id_pattern varchar(32);

    set id_count = 0;

    set first_cleaned = replace(replace(replace(replace(nameFirst, ' ', ''), '-', ''), '.', ''), '\'', '');
    set last_cleaned = replace(replace(replace(replace(nameLast, ' ', ''), '-', ''), '.', ''), '\'', '');

    set first_initial = lower(substr(first_cleaned, 1, 2));
    set last_initial = lower(substr(last_cleaned, 1, 5));

    set id_prefix = concat(last_initial, first_initial);

    set id_pattern = concat(id_prefix, '%');

    select
        count(*) into id_count
    from People
    where personID like id_pattern;

    set result = concat(id_prefix, LPAD(cast((id_count) as char),2,'0'));

    return result;
end;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
```

Out[ ]: []

Insert and Update Triggers

- We want to automatically generate `personID` using the function above whenever a row is inserted. The user should not need to manually specify it.

- Write a SQL trigger that runs every time a row is inserted

    - The trigger should generate a person ID for the row based on its `nameFirst` and `nameLast`; it should then set the `personID` for that row.
        - This should occur even if the user attempts to manually set `personID`. The user's value for `personID` is ignored.
        - You should call the function you wrote above
- Write another SQL trigger that runs every time a row is updated

    - There is no `immutable` keyword in MySQL; however, we can simulate immutability using a trigger. If the user attempts to modify `personID` directly, throw an exception.
    - If the user modifies `nameFirst` or `nameLast` such that the `personID` is no longer valid based on the rules in the previous section (specifically, if `[lastSubstr][firstSubstr]` is no longer the same as before), you should re-generate `personID` and re-set it.
        - You should call the function you wrote above
- **You are writing two SQL triggers for this section**

In [ ]:

```
%%sql

create trigger s24_hw3.set_id
    before insert
    on s24_hw3.People
    for each row
begin
    set new.personID = generatePersonID(new.nameFirst, new.nameLast);
end;

create trigger s24_hw3.update_id
    before update
    on s24_hw3.People
    for each row
begin
    if old.personID != new.personID then
        signal sqlstate 'HY000'
            set message_text = "You cannot change the PersonID.";
    end if;

    if old.nameFirst != new.nameFirst or old.nameLast != new.nameLast then
        set new.personID = generatePersonID(new.nameFirst, new.nameLast);
    end if;
end;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
0 rows affected.
```

Out[ ]:

```
[]
```

Create and Update Procedures

- You must implement two stored procedures

1. `createPerson(nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry, personID)`
    A. `personID` is an out parameter. It should be set to the ID generated for the person.
    B. All the other parameters are in paramaters
2. `updatePerson(personID, nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry, newPersonID)`
    A. `newPersonID` is an out parameter. It should be set to the ID of the person after the update (even if it didn't change).
    B. All the other parameters are in parameters.
        a. `personID` is used to identify the row that the user wants to update. The other in parameters are the values that the user wants to set.
        b. **Ignore null in parameters.** Only update an attribute if the in parameter is non-null.

- Depending on how you implemented your triggers, these procedures could be as simple as calling `insert`/`update` and setting the out parameters

In [ ]:

```
%%sql

CREATE PROCEDURE createPerson(
    IN nameFirst VARCHAR(32),
    IN nameLast VARCHAR(32),
    IN birthYear int,
    IN birthCountry VARCHAR(32),
    IN deathYear int,
    IN deathCountry VARCHAR(32),
    OUT personID char(9)
)
BEGIN
    SET personID = generatePersonID(nameFirst, nameLast);

    INSERT INTO People (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry)
    VALUES (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry);
END
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
```

Out[ ]: []

In [ ]:

```
%%sql

create
    definer = root@localhost procedure updatePerson(IN personID_1 char(9), IN nameFirst_1 varchar(32),
                                                    IN nameLast_1 varchar(32), IN birthYear_1 int,
                                                    IN birthCountry_1 varchar(32), IN deathYear_1 int,
                                                    IN deathCountry_1 varchar(32), OUT newPersonID char(9))
BEGIN
    declare nameFirst_old varchar(32);
    declare nameLast_old varchar(32);

    select
        p.nameFirst into nameFirst_old
    from People p
    where personID = personID_1;

    select
        p.nameLast into nameLast_old
    from People p
    where personID = personID_1;

    UPDATE s24_hw3.People
    SET nameFirst = COALESCE(nameFirst_1, nameFirst),
        nameLast = COALESCE(nameLast_1, nameLast),
        birthYear = COALESCE(birthYear_1, birthYear),
        birthCountry = COALESCE(birthCountry_1, birthCountry),
        deathYear = COALESCE(deathYear_1, deathYear),
        deathCountry = COALESCE(deathCountry_1, deathCountry)
    WHERE personID = personID_1;

    SET newPersonID = getOldPersonID(COALESCE(nameFirst_1, nameFirst_old), COALESCE(nameLast_1, nameLast_old));
END;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
```

Out[ ]: []

Security

- You must create a new user `general_user` and use security to allow it to perform only `select` and `execute` operations (i.e., no `insert`, `delete`, and `update` operations)

In [ ]:

```
%%sql

CREATE USER 'general_user'@'localhost' IDENTIFIED BY '123456';

GRANT SELECT, EXECUTE ON *.* TO 'general_user'@'localhost';

FLUSH PRIVILEGES;
```

```
 * mysql+pymysql://root:***@localhost
(pymysql.err.OperationalError) (1396, "Operation CREATE USER failed for 'general_user'@'localhost'")
[SQL: CREATE USER 'general_user'@'localhost' IDENTIFIED BY '123456';]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

Inheritance Using Views

- A person can be a player or manager
  - That is, a player is-a person, and a manager is-a person
- Describe how you could implement this inheritance relationship given that you already have your `people` table
  - No code is necessary

`Answer` We can create a view based on the people table, adding features for player inheriting his info in the people table.

Data Insertion Testing

- The cells below load data from `People.csv` to your database
  - No code is required on your part. Make sure everything runs without error.

In [ ]:

```
# Load People.csv into a dataframe.
# You may see NaNs in the non-null columns. This is fine.

people_df = pandas.read_csv("data/People.csv")
people_df.head(10)
```

Out[ ]:

	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry
0	Ed	White	1926.0	USA	1982.0	USA
1	Sparky	Adams	1894.0	USA	1989.0	USA
2	Bob	Johnson	1959.0	USA	NaN	NaN
3	Johnny	Ryan	1853.0	USA	1902.0	USA
4	Jose	Alvarez	1956.0	USA	NaN	NaN
5	Andrew	Brown	1981.0	USA	NaN	NaN
6	Chris	Johnson	1984.0	USA	NaN	NaN
7	Johnny	Johnson	1914.0	USA	1991.0	USA
8	Albert	Williams	1954.0	Nicaragua	NaN	NaN
9	Ed	Brown	NaN	USA	NaN	NaN

In [ ]:

```
def add_person(p):
    """
    p is a dictionary containing the column values for either a student or an employee.
    """
    cur = sql_conn.cursor()

    # This function changes the data, converting nan to None.
    # So, we make a copy and change the copy.
    p_dict = copy.copy(p)
    for k, v in p_dict.items():
        if isinstance(v, float) and math.isnan(v):
            p_dict[k] = None

    # This provides a hint for what your stored procedure will look like.
    res = cur.callproc("s24_hw3.createPerson",
                       # The following are in parameters
                        (p_dict['nameFirst'],
                        p_dict['nameLast'],
                        p_dict['birthYear'],
                        p_dict['birthCountry'],
                        p_dict['deathYear'],
                        p_dict['deathCountry'],
                        # The following are out parameters for personID.
                        None))

    # After the procedure executes, the following query will select the out values.
    res = cur.execute("""SELECT @_s24_hw3.createPerson_6""")
    result = cur.fetchall()

    sql_conn.commit()
    cur.close()
    return result[0]["@_s24_hw3.createPerson_6"]  # Return personID
```

- Below is the main data insertion logic
    - `add_person` calls your `createPerson` procedure
    - The `data` directory also contains a file `People_Ids.csv`, which is the expected `personID` for each row after it is inserted. We'll use this to check your `createPerson` implementation.

In [ ]:
```
%sql truncate table s24_hw3.people

expected_ids_df = pandas.read_csv("data/People-Ids.csv", header=None)
expected_ids = [e[0] for e in expected_ids_df.values.tolist()]

for i, (p, e_id) in enumerate(zip(people_df.to_dict(orient="records"), expected_ids)):
    p_id = add_person(p)
    assert p_id == e_id, \
    f"Row {i}: Expected {e_id}, but got {p_id} for {p['nameFirst']} {p['nameLast']}"

print("Successfully inserted all data")
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
Successfully inserted all data
```

Data Updating Testing

- The following cells test your update trigger and `updatePerson` implementation
    - No code is required on your part. Make sure everything runs as expected.
    - The tests assume you just finished the Data Insertion Testing section. You may run into issues if you run the Data Updating Testing section multiple times without reseting your data.

In [ ]:
```
# Switch back to root
%sql mysql+pymysql://root:tzy123456@localhost/s24_hw3
```

```
def transform(d):
    # %sql returns dict of attributes to one-tuples.
    # This function extracts the values from the one-tuples.
    return {k: v[0] for k, v in d.items()}

def is_subset(d1, d2):
    # Checks if d1 is a subset of a d2
    for k, v in d1.items():
        if k not in d2 or str(d2[k]) != str(v):
            return False
    return True
```

In [ ]:

```
# Create new person to test on

%sql call createPerson("Babe", "Ruth", null, null, null, null, @ruthID)
res1 = %sql select * from people p where p.personID = @ruthID
res1_d = transform(res1.dict())
expected_d = dict(
    personID="ruthba01",
    nameFirst="Babe",
    nameLast="Ruth",
    birthYear=None,
    birthCountry=None,
    deathYear=None,
    deathCountry=None
)

print(res1)

assert is_subset(expected_d, res1_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res1_d}"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| ruthba01 |    Babe   |   Ruth   |    None   |     None     |    None   |     None     |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

In [ ]:

```
# Update birth country and year
%sql call updatePerson(@ruthID, null, null, 1895, "USA", 1948, "USA", @ruthID)
res2 = %sql select * from people p where p.personID = @ruthID
res2_d = transform(res2.dict())
expected_d = dict(
    personID="ruthba01",
    nameFirst="Babe",
    nameLast="Ruth",
    birthYear=1895,
    birthCountry="USA",
    deathYear=1948,
    deathCountry="USA"
)

print(res2)

assert is_subset(expected_d, res2_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res2_d}"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| ruthba01 |    Babe   |   Ruth   |    1895   |     USA      |    1948   |     USA      |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

In [ ]:

```
# Checking that null is a noop
%sql call updatePerson(@ruthID, null, null, null, null, null, null, @ruthID)
res3 = %sql select * from people p where p.personID = @ruthID
res3_d = transform(res3.dict())

print(res3)

assert is_subset(expected_d, res3_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res3_d}"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
| ruthba01 |    Babe   |   Ruth   |    1895   |     USA      |    1948   |     USA      |
+----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

In [ ]:

```
# Try to manually set personID
# Note: You should get an OperationalError. If you get an AssertionError, then
# your trigger is not doing its job.

res4 = %sql update people set personID = "dff9" where personID = "ruthba01"

assert res4 is None, "Your trigger should throw an exception"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1644, 'You cannot change the PersonID.')
[SQL: update people set personID = "dff9" where personID = "ruthba01"]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
Success
```

In [ ]:

```
# Check that update trigger updates personID if name changes

%sql call updatePerson(@ruthID, "George", "Herman", 1920, "USA", 2005, "USA", @ruthID)
res5 = %sql select * from people p where p.personID = @ruthID
res5_d = transform(res5.dict())
expected_d = dict(
    personID="hermage01",
    nameFirst="George",
    nameLast="Herman",
    birthYear=1920,
    birthCountry="USA",
    deathYear=2005,
    deathCountry="USA"
)

print(res5)

assert is_subset(expected_d, res5_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res5_d}"

print("Success")
```

```
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
|  personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
| hermage01 |   George  |  Herman  |    1920   |     USA      |    2005   |     USA      |
+-----------+-----------+----------+-----------+--------------+-----------+--------------+
Success
```

Security Testing

- Write and execute statements below to show that you set up the permissions for `general_user` correctly
  - You should show that `select` and `execute` work, but `insert`, `update`, and `delete` don't

In [ ]:

```
# Connect to database as general_user
%sql mysql+pymysql://general_user:123456@localhost/s24_hw3
```

In [ ]:

```
# select
%sql select * from People limit 5;
```

```
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
5 rows affected.
```

Out[ ]:

personID	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry
aberntte01	Ted	Abernathy	1921	USA	2001	USA
aberntte02	Ted	Abernathy	1933	USA	2004	USA
abreujo01	Jose	Abreu	1987	Cuba	None	None
abreujo02	Joe	Abreu	1913	USA	1993	USA
adamsau01	Austin	Adams	1986	USA	None	None

In [ ]:

```
# execute
%sql call createPerson("Jerry", "Tan", 2000, "CN", null, null, @ID)
```

```
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
```

```
Out[ ]: []
```

```
In [ ]: # insert
        %% sql

        INSERT INTO People (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry)
        VALUES ('Zoe', 'Feng', 2000, 'CN', null, null);
```

```
  Cell In[24], line 4
    INSERT INTO People (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry)
           ^
SyntaxError: invalid syntax
```

```
In [ ]: # update
        %%sql

        UPDATE People
        SET birthYear = 2000
        WHERE personID = 'abernte01';
```

```
  Cell In[25], line 4
    UPDATE People
           ^
SyntaxError: invalid syntax
```

```
In [ ]: # delete
        %%sql

        DELETE FROM People
        WHERE personID = 'abernte01';
```

```
  Cell In[26], line 4
    DELETE FROM People
           ^
SyntaxError: invalid syntax
```

GoT Data Visualization

Data Loading

- Run the cell below to create and insert data into GoT-related tables

```
In [ ]: %sql mysql+pymysql://root:tzy123456@localhost/s24_hw3

        for filename in [
            "episodes_basics", "episodes_characters", "episodes_scenes"
        ]:
            df = pandas.read_json(f"data/{filename}.json")
            df.to_sql(name=filename, schema="s24_hw3", con=engine, index=False, if_exists="replace")

        print("Success")
```

```
Success
```

Overview

- In this section, you'll be combining SQL and Dataframes to create data visualizations

    - You may find this notebook helpful
    - You may also find the Pandas docs helpful
- **For all questions, you need to show the SQL output and the visualization generated from it.** See DV0 for an example.

DV0

- This question is an example of what is required from you
- Create a bar graph showing the amount of time each season ran for (in seconds)
- You should use the `episodes_scenes` table
- Note: `season_running_time <<` in the following cell saves the output of the SQL query into a local Python variable `season_running_time`

In [ ]:

```
%%sql

season_running_time <<

with one as (
    select seasonNum, episodeNum, sceneNum, sceneEnd, time_to_sec(sceneEnd) as sceneEndSeconds,
           sceneStart,  time_to_sec(sceneStart) as sceneStartSeconds,
            time_to_sec(sceneEnd)-time_to_sec(sceneStart) as sceneLengthSeconds
    from episodes_scenes
),
two as (
    select seasonNum, episodeNum, max(sceneEnd) as episodeEnd, max(sceneEndSeconds) as episodeEndSeconds
    from one
    group by seasonNum, episodeNum
),
    three as (
        select seasonNum, cast(sum(episodeEndSeconds) as unsigned) as totalSeasonSeconds,
               sec_to_time(sum(episodeEndSeconds)) as totalRunningTime
        from two
        group by seasonNum
    )
select * from three;
```

```
   mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
8 rows affected.
Returning data to local variable season_running_time
```

In [ ]:

```
# You must show the SQL output

season_running_time = season_running_time.DataFrame()
season_running_time
```

Out[ ]:

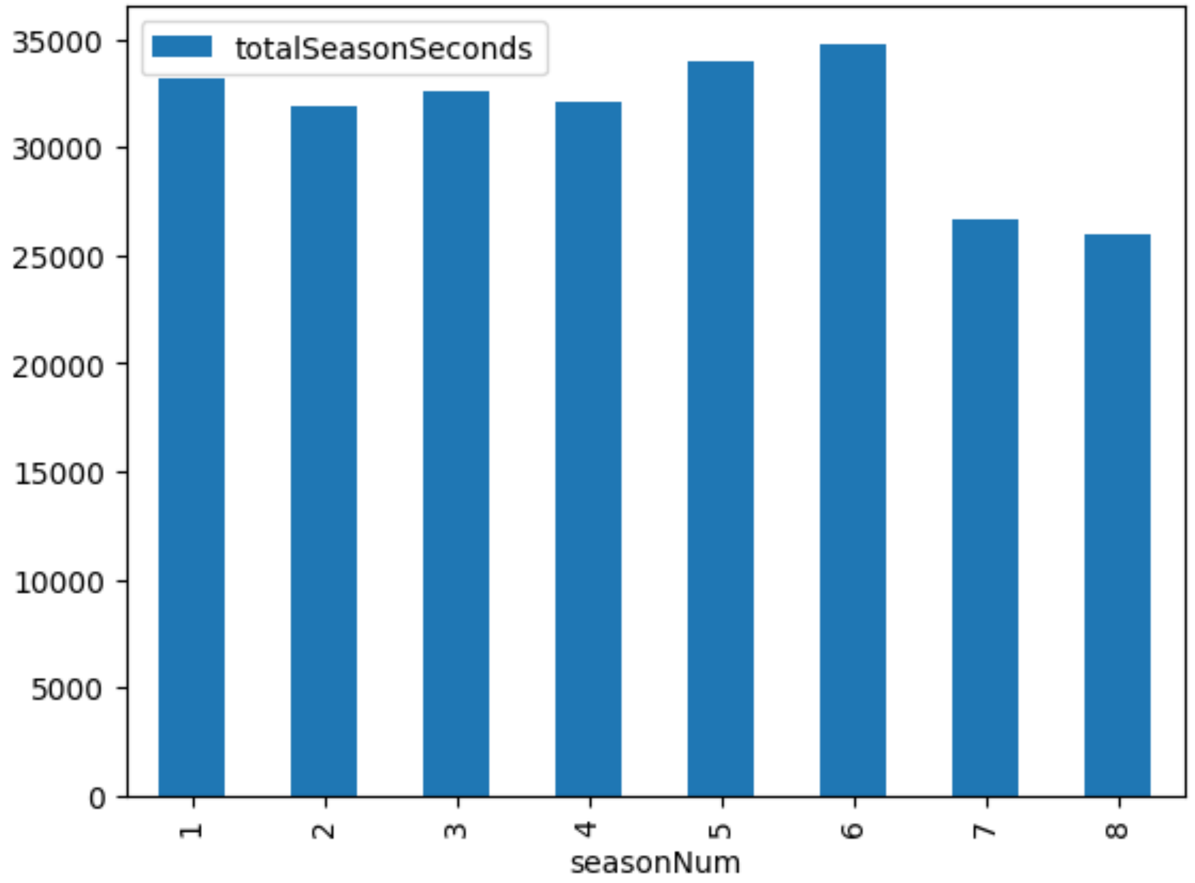
	seasonNum	totalSeasonSeconds	totalRunningTime
0	1	33143	0 days 09:12:23
1	2	31863	0 days 08:51:03
2	3	32541	0 days 09:02:21
3	4	32100	0 days 08:55:00
4	5	34003	0 days 09:26:43
5	6	34775	0 days 09:39:35
6	7	26675	0 days 07:24:35
7	8	25922	0 days 07:12:02

In [ ]:

```
# You must show the visualization

season_running_time[['seasonNum', 'totalSeasonSeconds']].plot.bar(x='seasonNum', y='totalSeasonSeconds')
```

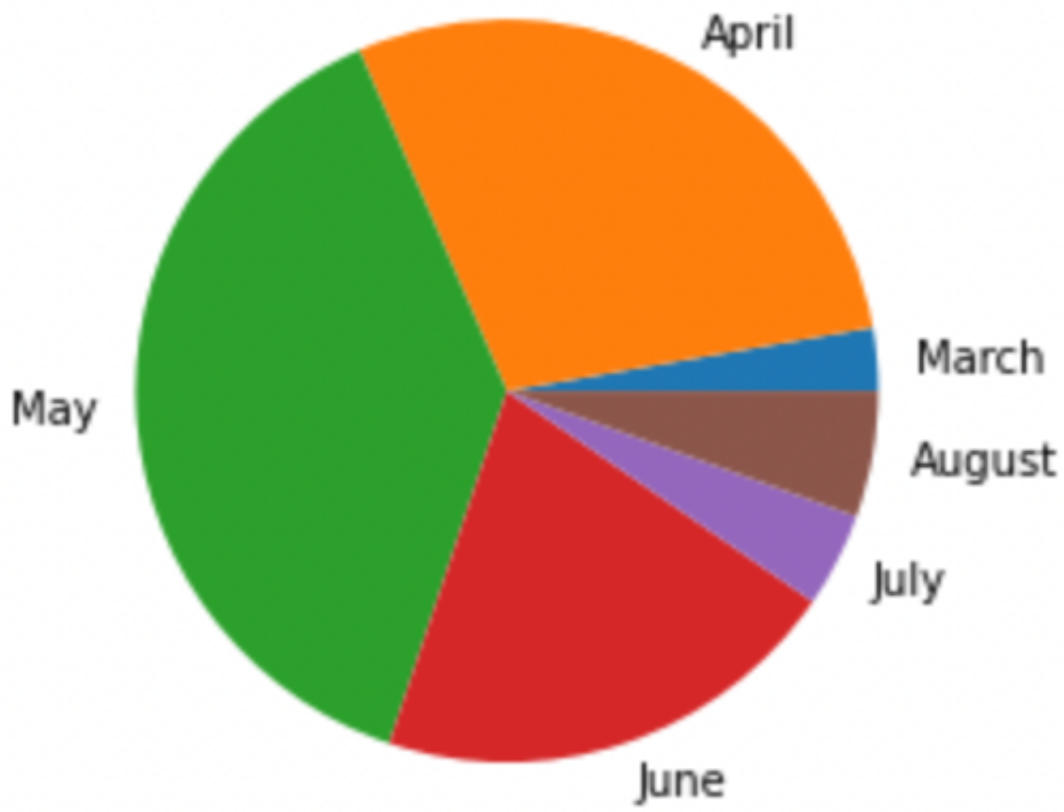
Out[ ]:

```
<Axes: xlabel='seasonNum'>
```



DV1

- Create a pie chart showing the proportion of episodes aired in each month (regardless of year)
- You should use the `episodes_basics` table
- As an example, your pie chart may look like this:



```
In [ ]: %%sql

        episodes_per_month <<

        select
            mth,
            count(*) as epi_each_month
        from(
        select
            *,
            month(episodeAirDate) as mth
        from episodes_basics
        ) a
        group by mth
```

```
  mysql+pymysql://general_user:***@localhost/s24_hw3
  mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
6 rows affected.
Returning data to local variable episodes_per_month
```

```
In [ ]: # SQL output

        episodes_per_month = episodes_per_month.DataFrame()
        episodes_per_month
```

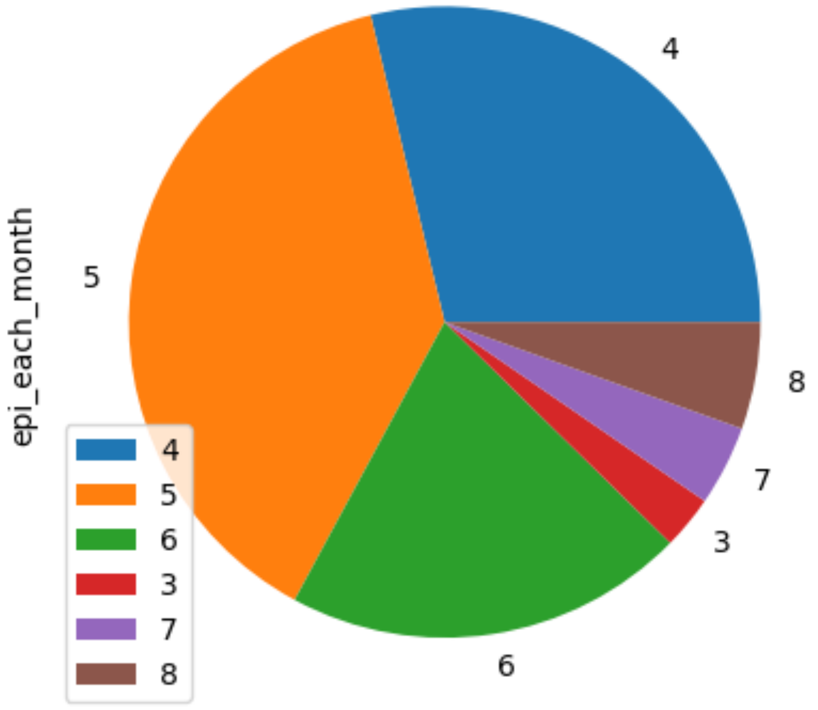
Out[ ]:

	mth	epi_each_month
0	4	21
1	5	28
2	6	15
3	3	2
4	7	3
5	8	4

```
In [ ]: # TODO: visualization
```

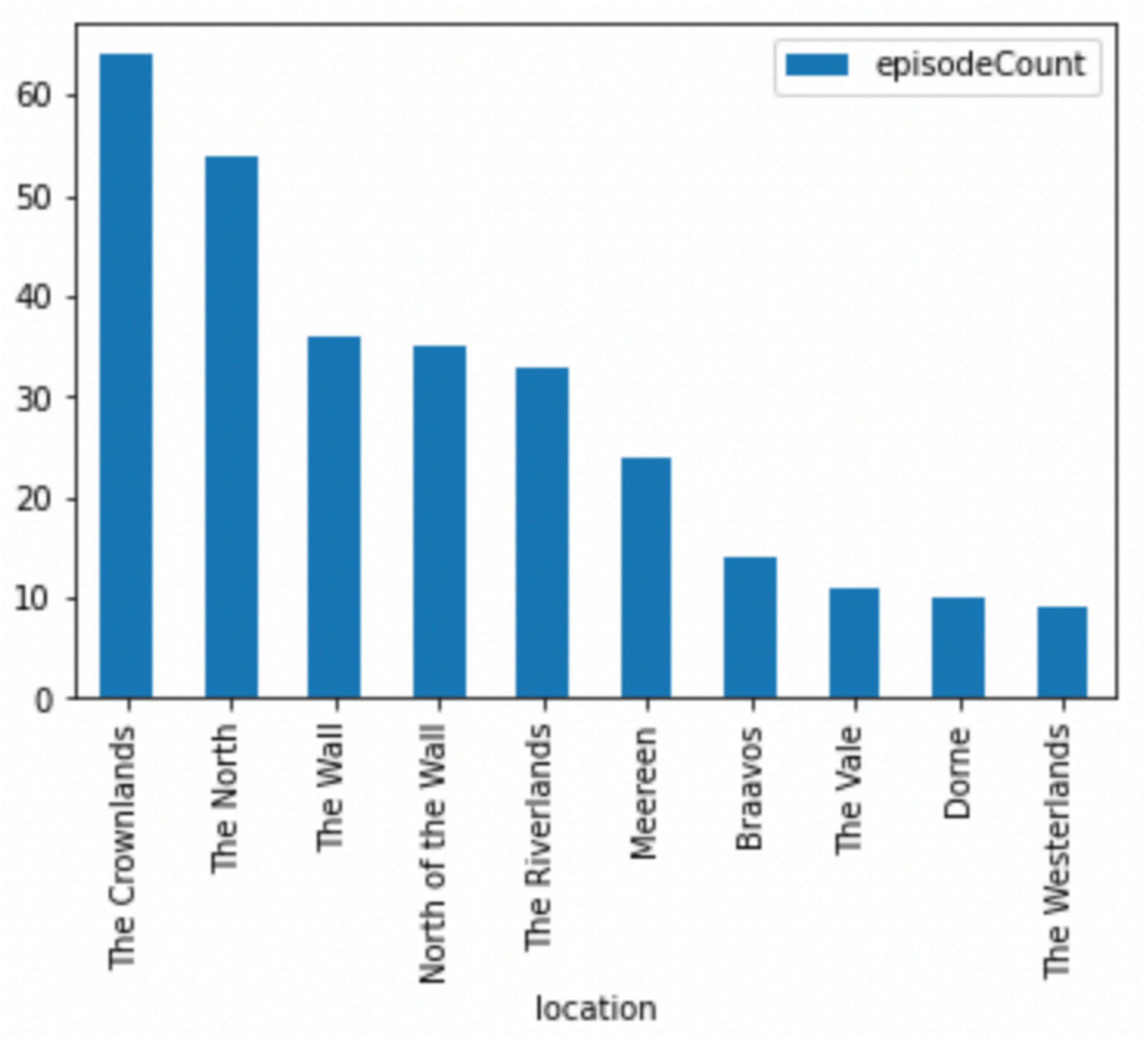
```
# You must show the visualization

episodes_per_month['epi_each_month'] = episodes_per_month['epi_each_month'].astype('float')
episodes_per_month.set_index('mth', inplace=True)
episodes_per_month.plot.pie(y='epi_each_month')
```

```
Out[ ]: <Axes: ylabel='epi_each_month'>
```



DV2

- Create a bar chart showing the number of episodes that every location (not sublocation) appeared in
  - You are counting the number of episodes, not scenes. If a location appeared in multiple scenes in a single episode, that should increment your count only by one.
  - You should order your chart on the number of episodes descending, and you should only show the top 10 locations
- You should use the `episodes_scenes` table
- As an example, your bar chart may look like this:



In [ ]:

```
%%sql

location_episode_count <<

select
    sceneLocation,
    count(distinct seasonNum, episodeNum) as episodeCount
from episodes_scenes
group by sceneLocation
order by episodeCount desc
limit 10;
```

```
   mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
10 rows affected.
Returning data to local variable location_episode_count
```

In [ ]:

```
# SQL output

location_episode_count = location_episode_count.DataFrame()
location_episode_count
```

Out[ ]:

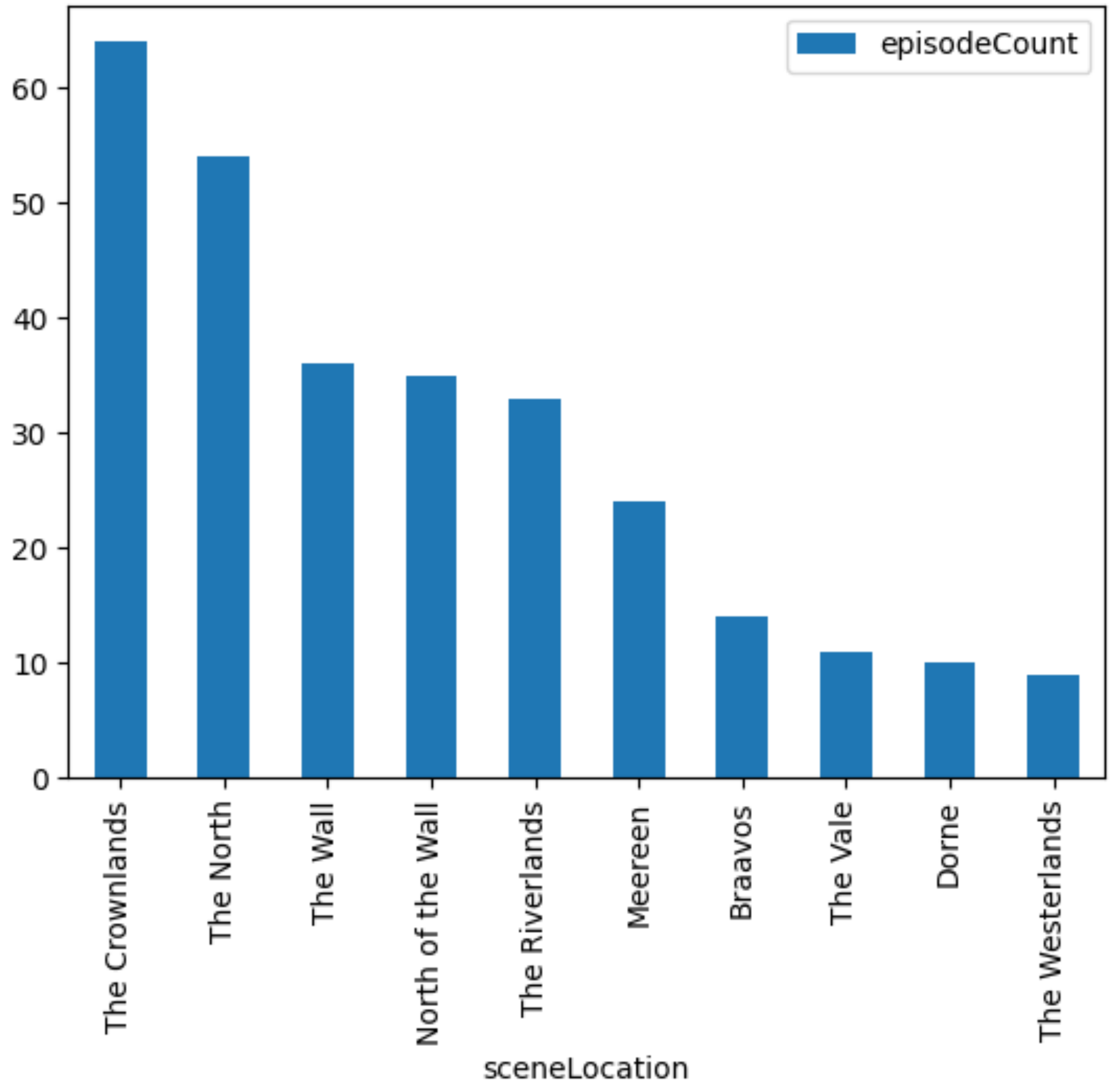
	sceneLocation	episodeCount
0	The Crownlands	64
1	The North	54
2	The Wall	36
3	North of the Wall	35
4	The Riverlands	33
5	Meereen	24
6	Braavos	14
7	The Vale	11
8	Dorne	10
9	The Westerlands	9

In [ ]:

```python
# TODO: visualization

location_episode_count.plot.bar(x='sceneLocation', y='episodeCount')
```

Out[ ]:

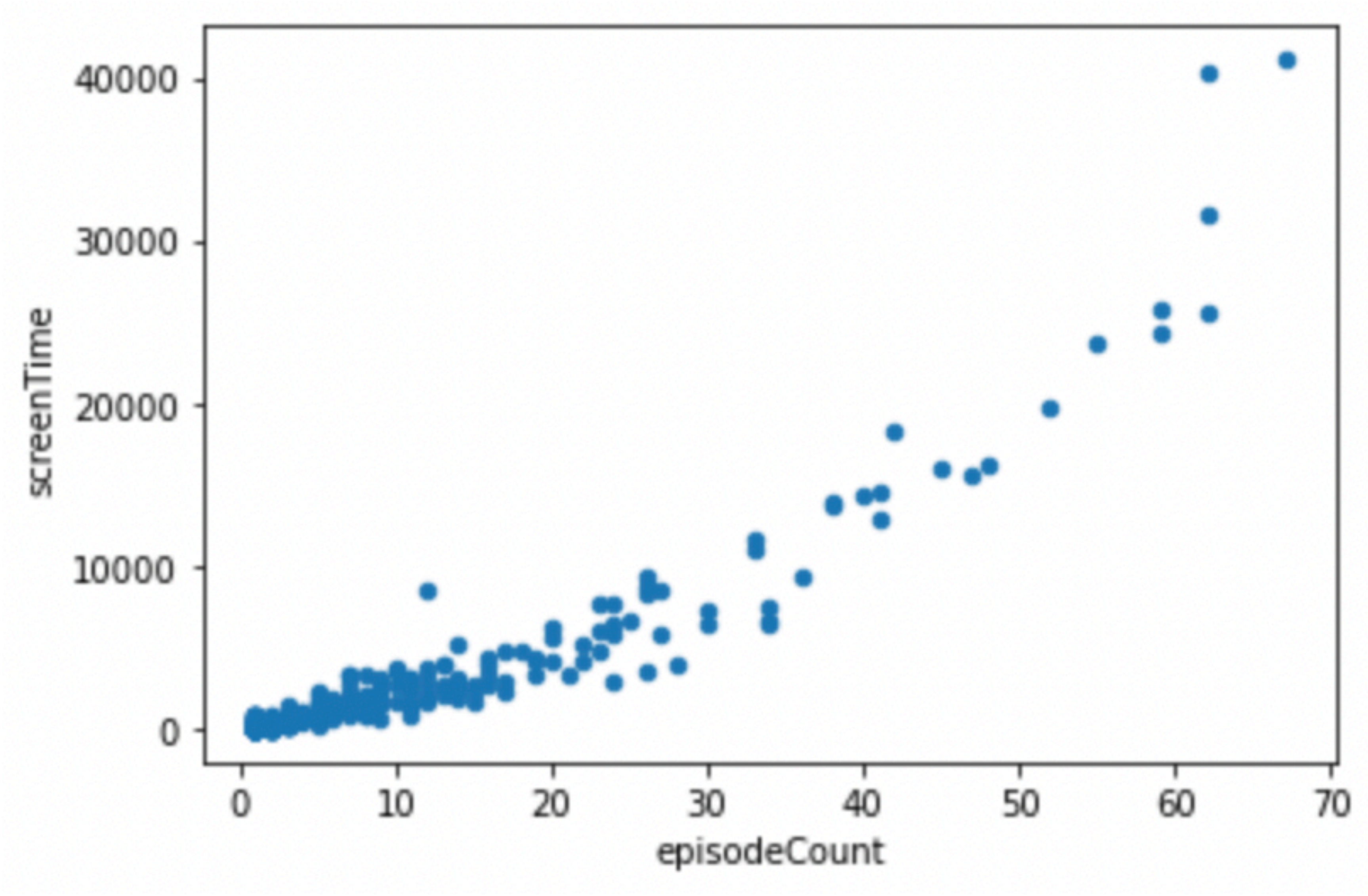
```
<Axes: xlabel='sceneLocation'>
```



DV3

- Create a scatter plot showing the relationship between the number of episodes (not scenes) a character appears in and their screen time (in seconds)
    - A character's screen time is the sum of the time lengths of all the scenes that the character appears in

- You should use the `episodes_characters` and `episodes_scenes` tables
- As an example, your scatter plot may look like this:



In [ ]:

```
%%sql

episode_count_screen_time <<

select
    characterName,
    count(distinct seasonNum, episodeNum) as episodeCount,
    sum(time_to_sec(sceneEnd) - time_to_sec(sceneStart)) as sceneTime
from(
select
    characterName,
    es.episodeNum as episodeNum,
    es.seasonNum as seasonNum,
    sceneStart,
    sceneEnd
from episodes_characters ec
left join episodes_scenes es
    on ec.episodeNum = es.episodeNum and ec.seasonNum = es.seasonNum and ec.sceneNum = es.sceneNum
) a
group by characterName;
```

```
   mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
577 rows affected.
Returning data to local variable episode_count_screen_time
```

In [ ]:

```
# SQL output
# Output is big, so just show first 10 rows
```

```
episode_count_screen_time = episode_count_screen_time.DataFrame()
episode_count_screen_time.head(10)
```

Out[ ]:

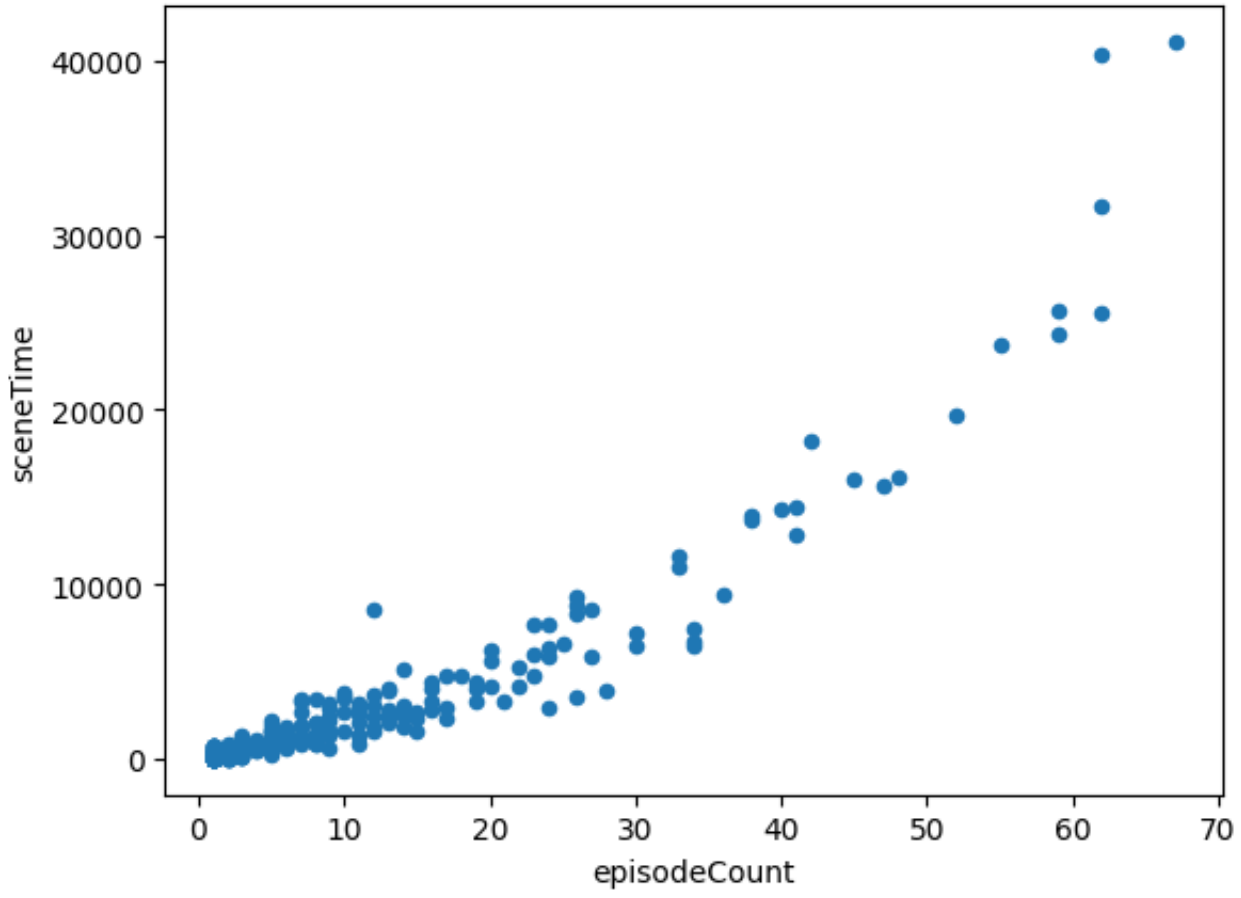
	characterName	episodeCount	sceneTime
0	Addam Marbrand	1	76
1	Adele Smyth-Kennedy	1	36
2	Adrack Humble	1	190
3	Aeron Greyjoy	2	601
4	Aerys Targaryen	1	23
5	Aggo	1	185
6	Akho	3	545
7	Alliser Thorne	19	4070
8	Alton Lannister	3	600
9	Alys Karstark	5	1050

In [ ]:

```
# TODO: visualization

episode_count_screen_time.plot.scatter(x='episodeCount', y='sceneTime')
```

Out[ ]:

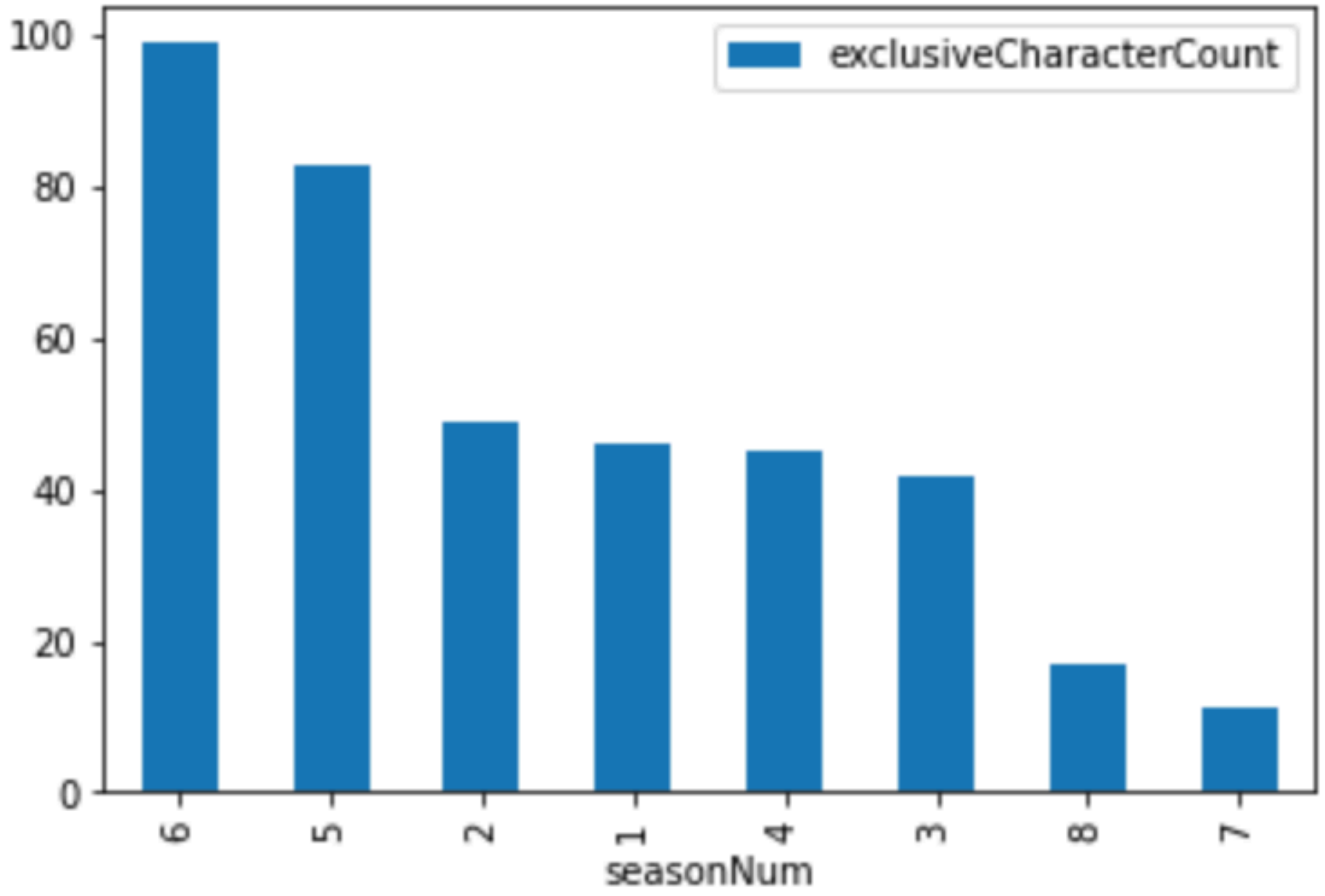
```
<Axes: xlabel='episodeCount', ylabel='sceneTime'>
```



DV4

- Create a bar chart showing the number of exclusive characters in each season
  - An exclusive character is a character that appeared in only that season, no other season
  - You should order your chart on the number of exclusive characters descending
- You should use the `episodes_characters` table

- You can assume `characterName` is unique across all characters. That is, a single name is one unique character.
- As an example, your bar chart may look like this:



```
In [ ]: %%sql

        season_exclusive_characters <<

        with one as (
        select
            characterName,
            count(distinct seasonNum) as appearCount
        from episodes_characters
        group by characterName
        )

        select
            seasonNum,
            count(distinct o.characterName) as exclusiveCharacterCount
        from episodes_characters ec
        left join one o
            on ec.characterName = o.characterName
        where appearCount = 1
        group by seasonNum
        order by exclusiveCharacterCount desc;
```

```
   mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
8 rows affected.
Returning data to local variable season_exclusive_characters
```

```
In [ ]: # SQL output
```

```
season_exclusive_characters = season_exclusive_characters.DataFrame()
season_exclusive_characters
```

Out[ ]:

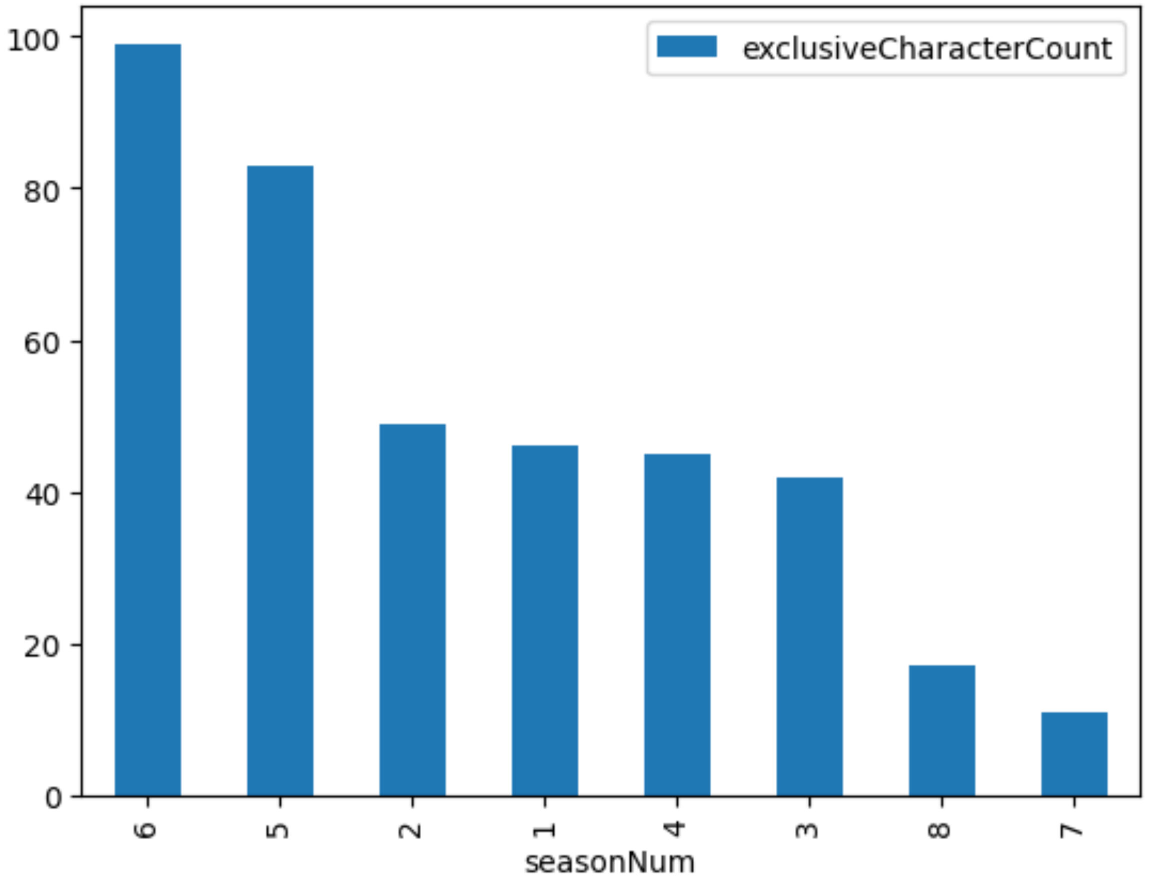
	seasonNum	exclusiveCharacterCount
0	6	99
1	5	83
2	2	49
3	1	46
4	4	45
5	3	42
6	8	17
7	7	11

In [ ]:

```
# TODO: visualization

season_exclusive_characters.plot.bar(y='exclusiveCharacterCount', x='seasonNum')
```

Out[ ]:

```
<Axes: xlabel='seasonNum'>
```


exclusiveCharacterCount
seasonNum