GleanTable

We're going to gradually build a table module from scratch.

In general:

- Do the parts in order. **The evaluation criteria is: 1) Progress on the parts, with correct output & satisfying the noted constraints 2) Reasonably clean code.**
- Unless noted otherwise, run time complexity does not need to be perfectly optimal, but try to avoid doing things super inefficiently.
- Feel free to add more public/private functions if useful
- Assume input is well-formed
- Feel free to use the internet for any coding-related questions
- No third party libraries but you can use standard libraries
- **Do some testing to make sure your code works, but do not spend time writing full-on unit tests. Again, you want to make it through as many parts as possible.**

V0

We want to implement a table where the columns have names and a record is a given row in the table. The values are all integers. Initially, it should support the following operations:

```
class GleanTable:
  def __init__(self, column_names: list[str]):
    """Constructor, creates a table with the given columns"""

  def append_row(self, row: list[int]):
    """Adds a row to the end of the table. Row will have the same length as the
number of columns, and index i of the row corresponds to column i"""

  def print(self, columns: list[str]):
    """Prints to the console the desired column headers in the order given followed
by the values of the provided columns. If columns is an empty array, prints all
columns. Different rows must be separated by a new line and rows must be printed in
order of insertion."""
```

For the column names in `print()`, you can **assume that the columns exist**. You can also assume similarly for the rest of this problem.

Example output for `print(["A", "B"])`:

```
A B
1 3
2 6
```

V1

Support a slice function:

```
  def slice(self, columns: list[str]) -> ‘GleanTable’:
    """Returns a new GleanTable instance which only has the passed in columns."""
```

You should now be able to do `t1.slice([“A”]).print([])`

```
A
1
2
```

V2

Support functions for adding/deleting a new column to the table.

```
  def add_column(self, column: str) -> ‘GleanTable’:
    """Returns a new GleanTable instance with the additional new column to the table.
For existing rows, values for this column should be set to 0. You can assume the
column name doesn’t match any of the existing column names."""

  def delete_column(self, column: str) -> ‘GleanTable’:
    """Returns a new GleanTable instance which removes the given column. You can
assume the column always exists."""
```

V3

Make sure you discuss your solution for inner_join with your interviewer before you start writing code.

```
  def inner_join(self, table2: GleanTable, column: str) -> ‘GleanTable’:
    """Returns a new GleanTable instance with columns from this table and table2 for
whenever there are records with matching values for the given column. There is only
one shared column between the two tables, which is the given column. The ordering of
the columns in the new table should be {self.columns - column}{column}{table2.columns
- column}, i.e. the inner join column is in the middle. See examples on the next
page.
```

For inner_join() only:

- Order of the rows in the output table does not matter.
- The column values are **unique within each table** (i.e. there will only be at most one match for each row).
- **IMPORTANT: THE COLUMN VALUES ARE NOT HASHABLE** - meaning that we cannot use the values as keys to a HashMap/dictionary, and we can’t add the values to a HashSet/ set. They are comparable.
- This method will be used infrequently
- **We want the runtime complexity for this method to be better than O(M * N), where M, N are the number of rows in the tables and you can assume M, N are large and similar in magnitude.**

```
Example 1:
Current table:
A | B
------
1 | 3
2 | 6


Table2:
B | C
------
6 | 8
5 | 3

Then table.inner_join(table2, “B”) should return a table as follows since B = 6 in
both tables:
A | B | C
----------
2 | 6 | 8

Example 2 (larger):
Current table:
A | B
------
1 | 3
2 | 6
7 | 9
4 | 4

Table2:
B | C
------
6 | 8
5 | 3
4 | 2

Then table.inner_join(table2, “B”) should return a table as follows since B = 6 and B
= 4 matches in both tables:
A | B | C
----------
2 | 6 | 8
4 | 4 | 2
"""
```

V3 - continued

```
  def outer_join(self, table2: GleanTable, column: str) -> ‘GleanTable’:
```

"""Returns a new GleanTable instance with columns from this table and table2 for whenever there are records with matching values for a column, and for values that don’t match, adds a record with only values for the columns in the table it came from and 0 for all columns in the other table. There is only one shared column between the two tables, which is the given column. The ordering of the columns in the new table should be {self.columns - column}{column}{table2.columns - column}.

Order of the rows does not matter. Assume the input column exists in both tables. **The constraints from the inner_join() part do not apply, i.e. column values are not guaranteed to be unique within each table, column values are hashable, and runtime complexity does not need to be better than O(M * N).**

Example 1:

Current table:
```
A | B
______
1 | 3
2 | 6
```

Table2:
```
B | C
______
6 | 8
5 | 3
```

Then table.outer_join(table2, “B”) should return a table as follows:
```
A | B | C
__________
1 | 3 | 0
2 | 6 | 8
0 | 5 | 3
```

Example 2:

Current table:
```
A | B
______
1 | 3
2 | 6
7 | 4
9 | 4
```

Table2:
```
B | C
______
6 | 8
4 | 3
4 | 2
```

```
Then table.outer_join(table2, “B”) should return a table as follows:
A | B | C
----------
1 | 3 | 0
2 | 6 | 8
7 | 4 | 2
7 | 4 | 3
9 | 4 | 2
9 | 4 | 3
"""
```

V4
Make sure your solution for V3 works correctly before moving on to V4.

Suppose `append_row()` successfully appends a row 60% of the time and does nothing 40% of the time due to write failures. Add the following implemented function to GleanTable:

Python
```
  import random
   def append_row_noisy(self, row):
       if random.random() < 0.4:
         return
       self.append_row(row)
```
Java
```
    import java.lang.Math;
   public void append_row_noisy(List<int> row) {
       if (Math.random() < 0.4) return;
       append_row(row);
   }
```

We want to build the `RobustGleanTable`, which has the same functionality and requirements as GleanTable V0, but insertions succeed at least 90% of the time. We want to do so using multiple `GleanTables`, where a single `GleanTable’s` `append_row_noisy()` succeeds 60% of the time. We will not have knowledge/nor should we derive knowledge if a single table’s append succeeded. All rows inserted will be unique.

```
class RobustGleanTable:
  def __init__(self, column_names: string[]):
    """Constructor, creates a table with the given columns"""

  def append_row(self, row: int[]):
    """Adds a row to the end of the table. Row will have the same length as the
number of columns, and index i of the row corresponds to column i"""
```

```
  def print(self, columns: string[]):
"""Prints to the console the desired column headers in the order given followed by
the values of the provided columns. If columns is an empty array, prints all columns.
Different rows must be separated by a new line and rows must be printed in order of
insertion."""
```

V5

Support export & import functions (outside of the class)

```
 def export_table(filename: string, table: GleanTable, columns: string[]):
    """Write the table (only the selected columns) as a csv to the given filename."""

 def import_table(filename: string, columns: string[]):
    """Reads the selected columns from csv and returns a GleanTable."""
```