Lecture 6:

- Structured Data (cont)
- Overview of HW-2

Announcements:

- HW-2 out, due Tues Sept. 26
- Quiz 3 on Thurs (python classes, dataset basics)
Often need to **combine** multiple datasets for data analytics

- typically we combine datasets by matching rows on their “objects”
- in database systems, combining tables is called a **join**
- *note*: database systems are *really* good at performing joins

<table>
<thead>
<tr>
<th>SaleId</th>
<th>EmployeeId</th>
<th>CarName</th>
<th>ModelYear</th>
<th>Amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>555</td>
<td>12</td>
<td>ford pinto</td>
<td>75</td>
<td>3076</td>
</tr>
<tr>
<td>556</td>
<td>12</td>
<td>toyota corolla</td>
<td>75</td>
<td>2611</td>
</tr>
<tr>
<td>998</td>
<td>13</td>
<td>toyota corolla</td>
<td>75</td>
<td>2800</td>
</tr>
<tr>
<td>999</td>
<td>12</td>
<td>toyota corolla</td>
<td>76</td>
<td>2989</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CarName</th>
<th>ModelYear</th>
<th>MSRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ford pinto</td>
<td>75</td>
<td>2769</td>
</tr>
<tr>
<td>toyota corolla</td>
<td>75</td>
<td>2711</td>
</tr>
<tr>
<td>ford pinto</td>
<td>76</td>
<td>3025</td>
</tr>
<tr>
<td>toyota corolla</td>
<td>76</td>
<td>2789</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

A simple (but inefficient) approach: ... aka nested-loops join

---

**Algorithm:** Combine (inefficient)

**Input:** Tables $R_1$, $R_2$ and attributes $A_1$, $A_2$ to match on, respectively

**Result:** New combined table $R_3$.

1 begin
2      $R_3 :=$ new empty table
3      for row $r_1$ in $R_1$ do
4          for row $r_2$ in $R_2$ do
5          if $A_1$ values for $r_1 == A_2$ values for $r_2$ then
6              add new row $r_1 + r_2$ to $R_3$
In previous version of combine, only the matches are returned

- sometimes we may want to keep non-matches (by “null” padding)
- where a “null” value means the value is missing / doesn’t exist
- we will use '' (empty string) to mean “null” (NA below)

Example of including the non-matches: ... aka a full “outer” join

<table>
<thead>
<tr>
<th>SaleId</th>
<th>Employeeld</th>
<th>CarName</th>
<th>ModelYear</th>
<th>Amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>555</td>
<td>12</td>
<td>ford pinto</td>
<td>75</td>
<td>3076</td>
</tr>
<tr>
<td>556</td>
<td>12</td>
<td>toyota corolla</td>
<td>75</td>
<td>2611</td>
</tr>
<tr>
<td>998</td>
<td>13</td>
<td>toyota corolla</td>
<td>75</td>
<td>2800</td>
</tr>
<tr>
<td>999</td>
<td>12</td>
<td>toyota corolla</td>
<td>76</td>
<td>2989</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CarName</th>
<th>ModelYear</th>
<th>MSRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ford pinto</td>
<td>75</td>
<td>2769</td>
</tr>
<tr>
<td>toyota corolla</td>
<td>75</td>
<td>2711</td>
</tr>
<tr>
<td>toyota corolla</td>
<td>77</td>
<td>2789</td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>SaleId</th>
<th>Employeeld</th>
<th>CarName</th>
<th>ModelYear</th>
<th>Amt</th>
<th>MSRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>555</td>
<td>12</td>
<td>ford pinto</td>
<td>75</td>
<td>3076</td>
<td>2769</td>
</tr>
<tr>
<td>556</td>
<td>12</td>
<td>toyota corolla</td>
<td>75</td>
<td>2611</td>
<td>2711</td>
</tr>
<tr>
<td>998</td>
<td>13</td>
<td>toyota corolla</td>
<td>75</td>
<td>2800</td>
<td>2711</td>
</tr>
<tr>
<td>999</td>
<td>12</td>
<td>toyota corolla</td>
<td>76</td>
<td>2989</td>
<td>NA</td>
</tr>
<tr>
<td>NA</td>
<td>NA</td>
<td>toyota corolla</td>
<td>77</td>
<td>NA</td>
<td>2789</td>
</tr>
</tbody>
</table>

Q: Why might we want to include the non-matches?

- so we don’t lose data in the tables just because they don’t have a match!
- often, we will deal with missing values after we combine (as needed)
- e.g., might want to try and “infer” what the missing values are
Q: How would we combine these two tables? What is different?

<table>
<thead>
<tr>
<th>MPG</th>
<th>Cyls</th>
<th>Displacement</th>
<th>Hrspwr</th>
<th>Wght</th>
<th>Accel</th>
<th>ModelYear</th>
<th>Origin</th>
<th>CarName</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.0</td>
<td>4</td>
<td>140.0</td>
<td>83.0</td>
<td>2639</td>
<td>17.0</td>
<td>75</td>
<td>1</td>
<td>ford pinto</td>
</tr>
<tr>
<td>29.0</td>
<td>4</td>
<td>97.0</td>
<td>75.0</td>
<td>2171</td>
<td>16.0</td>
<td>75</td>
<td>3</td>
<td>toyota corolla</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CarName</th>
<th>ModelYear</th>
<th>MSRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ford pinto</td>
<td>75</td>
<td>2769</td>
</tr>
<tr>
<td>toyota corolla</td>
<td>75</td>
<td>2711</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- combine on object attributes \{CarName, ModelYear\} even though out of order

Q: What are better ways to combine than “nested loops” approach?

- sort join attributes, then “match” rows like merge in mergesort
- hash the join attributes of \( T_2 \), iterate \( T_1 \) rows and probe hash table (*)

(*) note in python can use a dictionary as the hash table
Intro to Data Preparation

The early steps of getting data ready to process / analyze …

- formatting data (e.g., getting into proper tabular form)
- combining data sets (e.g., “joining” or “concatenating”)
- dealing with incorrect values (e.g., misspelled names, values out of range)
- dealing with missing values
- transforming values (e.g., to similar units, to categorical values)
- and so on
HW-1

- goal is to develop classes for representing tabular data
- will use these over the rest of the course
- (very) loosely based on numpy arrays and pandas data frames

DataRow ... represents a row (instance) of data

```python
def __init__(self, cols=[], vals=[])  # eg: r = DataRow(['a','b'], [10,20])
def __repr__(self)  # returns formatted string
def __getitem__(self, column)  # eg: v = r['a']
def __setitem__(self, column, value)  # eg: r['b'] = 30
def __delitem__(self, column)  # eg: del r['a']
def __eq__(self, other)  # eg: r1 == r2
def __add__(self, other)  # r1 + r2
def columns(self)  # r.columns() == ['a', 'b']
def values(self, columns=None)  # r.values() == [10, 20]
def select(self, columns=None)  # r_new = r.select(['a'])
def copy(self)  # r_new = r.copy()
```

Some notes:
- add() rows must have different columns
- values() can take subset of columns to return values for
- select() can take subset of columns, returns new row
- copy() just calls select() without columns
- this class is “done” for you ...
DataTable

... represents a table of rows with same columns

def __init__(self, columns=[]) # eg: t = DataTable(['a', 'b', 'c'])
def __repr__(self) # returns formatted string
def __getitem__(self, row_index) # eg: r = t[0] ... copy of 1st row
def __delitem__(self, row_index) # eg: del t[3] ... delete 4th row
def column_count(self) # gives number of columns
def row_count(self) # gives number of rows
def columns() # eg: t.columns() == ['a', 'b', 'c']
def append(self, row_values) # eg: t.append([10, 20])
def rows(self, row_indexes) # eg: new_t = t.rows([10, 15, 80])
def copy(self) # t_new = t.copy()
def update(self, row_index, col, val) # eg: t.update(0, 'b', 30)
def load(self, filename, delimiter=',')
def save(self, filename, delimiter=',')

@staticmethod
def combine(table1, table2, columns=[], non_matches=False)

@staticmethod
def convert_numeric(value)

Some notes:

- load() / save() loads/saves rows to/from text file
- combine() combines two tables on the given columns
- not the most efficient approach!
- provides minimal support for data tables
To format rows and tables, we will use the `tabulate` package:

```
conda install -y tabulate
```

For load and save, we’ll use `csv` package: ... comma-separated values

- should already be installed

**Unit testing using `pytest`**

- you are given a suite of unit tests that you need to pass
- in future assignments you will be asked to write your own tests
- can run all the tests (`hw2_tests.py`) from command line:

  ```
  pytest -v hw2_tests.py  
  ...
  ```

- test functions use: `assert test-result == desired-result`
- exception tests use: `with pytest.raises(SomeException) as e`:

**You will create another simple jupyter notebook ...**

- to show that your `DataTable` implementation works correctly
- focus on loading, selecting (e.g., 10 rows), displaying, and saving data tables
Additional “hints” for HW-2 ... Be sure to:

- look at the existing code provided
- look at (and understand) the unit tests
- read the function comments
- ask questions if unsure about what function does (piazza)
- leave yourself enough time to work on combine (most involved by far)

Notes on the load function:

```python
def load(self, filename, delimiter=',',):
    with open(filename, newline='') as csvfile:
        reader = csv.reader(csvfile, delimiter=delimiter)
        num_cols = len(self.columns())
        for row in reader:
            row_cols = len(row)
            if num_cols != row_cols:
                raise ValueError(f'expecting {num_cols}, found {row_cols}')
            converted_row = []
            for value in row:
                converted_row.append(DataTable.convert_numeric(value.strip()))
            self.__row_data.append(DataRow(self.columns(), converted_row))
```

- `with` is similar to `try-except` but automatically handles “cleanup"
- does this via a special `__exit__(...)__` function on `with` object
- note here we are opening the file via to read from (default for `open`)
- `newline` parameter here returns the newline to the reader for each line
Notes on the save function:

```python
def save(self, filename, delimiter=','):
    with open(filename, 'w', newline='') as csvfile:
        writer = csv.writer(csvfile, delimiter=delimiter, quotechar='''',
                            quoting=csv.QUOTE_NONNUMERIC)
        for row in self.__row_data:
            writer.writerow(row.values())
```

- here we are opening the file for writing
- for writing non-numeric values, we use double quotes