Data Cleansing Exercise

Introduction

Often times, data folk will need to deal with 'messy' data. This can mean it can have missing values, inconsistent formatting, malformed data or other outliers.

We'll be using two open source Python libraries to help us clean some example data - Pandas and NumPy.

Part 1 - Setting up

1. Create a virtual env inside the supplied `data-cleansing` directory.
2. Install the necessary packages from the `requirements.txt` file.
3. Run `jupyter notebook` to fire up a notebook instance.
4. Create a new `Python 3` notebook.
5. Give it a name by clicking `Untitled` near the top.

Part 2 - Dropping Columns

When cleaning datasets, one step to take is to remove columns that we don't care about. Retaining this unnecessary data means that we would be using more disk space and CPU cycles, as well as potentially slowing down our processing time. We will use `pandas` to help us. Have your notebook open and ready to use.

1. In the first cell of your notebook, run `import pandas as pd`.
2. In the second cell, run the below code. The first line will create what's called a Data Frame for us. The second line will provide us information about the headers and the first five rows (by default). You should see 15 column names with five rows of data.

   ```
   df = pd.read_csv('british-library-books.csv')
   df.head()
   ```

   Note: For `head()`, 5 is the default number of rows returned, but you can pass an integer value as an argument to return as many as you want, like this: `head(200)`. You can also pass a negative value to get all rows except the last *n* rows, for example: `head(-5)`. `tail()` also exists and is the inverse of `head()`.
3. We're only interested in certain columns, so let's drop the ones we don't need. Update the code in the second cell to the following. Open the `Cell` and hit `Run All` to rerun both cells. Notice now that some of the columns have been removed entirely.

```
df = pd.read_csv('british-library-books.csv')
to_drop = [
    'Edition Statement',
    'Corporate Author',
    'Corporate Contributors',
    'Former owner',
    'Engraver',
    'Contributors',
    'Issuance type',
    'Shelfmarks',
    'Flickr URL'
]

df.drop(columns=to_drop, inplace=True)
df.head()
```

The `drop()` function removes rows or columns from our data. The first argument takes a list of label names, assigned to either the `columns` named arg, or the `index` named arg. Choosing one of these named args specifies which axis we want to operate on. The second argument, `inplace`, is a boolean that specifies whether the returned data will be copied (`false`) or will update the current object it's working on (`true`).

Part 3 - Changing the default index

You may have noticed that when you run `head()`, the first column is an unnamed index (primary key) column, starting from zero and incrementing by one for every row. This isn't necessarily a problem, but imagine someone wants to *search* for a record, such as a librarian. All books have unique identifiers, which also happens to be in our data set, so let's use that instead.

We can verify that the column of data is unique by running the below snippet in a **new** cell. It will return `True` if it holds that each field in the column is unique.

```
df['Identifier'].is_unique
```

To update the index in the dataframe, we can now run the below code, and output the result.

```
df.set_index('Identifier', inplace=True)
df.head()
```

Unlike primary keys in SQL, a Pandas index doesn't make any guarantee of being unique, although many indexing and merging operations will notice a speedup in runtime if it is.

Tidying up field data

When we read data into a dataframe, the type of our data for each column is called `dtype`, which can hold any Python object, and is somewhat analogous to Python's `str`. It's used to encapsulate any field that doesn't fit as numerical or categorical data. This is useful to us as some of our data is 'messy'.

Some columns of data are not consistent throughout. For example, `Date of Publication` contains a variety of formats. This would make it hard to query our data and get back accurate results. Run the below code snippet and look at the output. This will return our dataframe data for just the `Date of Publication`, *starting* from the index of 206. Try another identifier and see how the data changes.

```
df.loc[206:, 'Date of Publication'].head(10)
```

`loc[]` (not intuitively named) allows us to do label-based indexing, which is the labelling of a record without regard to its position. For instance, `loc[206]` would be the first label of the index. If you want positional-based indexing (like Python lists), use `iloc[]`.

The `:` after 206 denotes that we want to get back every record after it. We then chop that result to only give us back the first 10.

Notice the differences in the data: `1879 [1878]`, `1868`, `[1789?]` etc. We should clean this data in the following ways:

1. Completely remove the dates we are not certain about, change to `NaN`. For example: `[1897?]` or `[1885.]`.
2. If a value has been read in with a type of `float`, change to `NaN`.
3. If a value is missing a matching bracket (has `[` but no `]` for instance), change to `NaN`.
4. Remove the extra dates in square brackets, wherever present. For example: `1879 [1878] --> 1879`
5. Convert date ranges to their start date, wherever present. For example: `1860-1863 --> 1860`

Create a new cell and try to implement a solution that will meet the above criteria. Here is some code to help start you off. Note, `np.nan` is a special type of value provided from the `numpy` library to allow us to have a `NaN` type value which will work with numbers.

```
col = 'Date of Publication'

for i in range(len(df[col])):
    current = df[col].iloc[i]

    # insert your code here

    # example - get date after [ bracket
    if current.startswith('['):
        df[col].iloc[i] = current[1:5]
```

When you think you have completed the above, you can apply this code snippet to verify it works:

```
df[col] = pd.to_numeric(df[col])
df[col].dtype
>>> dtype('float64')
```

`to_numeric` will try and convert all of the values in the `Date of Publication` column, which are all currently strings, into a number type. If you get an error when running this, it likely means you haven't completed all of the necessary cleaning steps.

We can also check how many of our dates couldn't be converted. The below snippet tells us that roughly every 1 in 10 values are `NaN`.

```
df['Date of Publication'].isnull().sum() / len(df)
>>> 0.0905687718874532
```

You've reached the end of the exercise. Congrats!