Cleaning data in pandas

For cleaning jobs of any size, specialized tools like OpenRefine are still your best bet -- a typical workflow is to clean your data in OpenRefine, export as a CSV, then load into pandas.

But in many cases, you can use some of pandas' built-in tools to whip your data into shape. This is especially useful for data processing tasks that you plan to repeat as the data are updated.

Let's import pandas, then we'll run through some scenarios.

In [ ]:
```
import pandas as pd
```

How dirty is your data?

In Excel, running a pivot table (with counts) for each column will show you misspellings, external white space, inconsistent casing and other problems that keep your data from grouping correctly.

In SQL, you might do the same thing with The Golden Query™:

```
SELECT column, COUNT(*)
FROM table
GROUP BY column
ORDER BY 2 DESC
```

To do the equivalent operation in pandas, you can just call the `value_counts()` method on a column. Let's look at some Congressional junkets data as an example:

In [ ]:
```
junkets = pd.read_csv('../data/congress_junkets.csv')
```

In [ ]:
```
junkets.head()
```

Let's run `value_counts()` on the *Destination* columnn:

In [ ]:
```
junkets['Destination'].value_counts()
```

The default sort order is by count descending, but it can also be helpful in finding typos to sort by the name -- the "index" of what `value_counts()` returns. To do that, tack on `sort_index()` :

In [ ]:
```
junkets['Destination'].value_counts().sort_index()
```

... and now we start to see some common data problems in our 838 unique destinations -- whitespace, inconsistent values for the same thing ("Accra" and "Accra, Ghana") -- and can start fixing them.

Fixing whitespace, casing and other "string" problems

If part of our analysis hinged on having a pristine "Destination" column, then we've got some work ahead of us. First thing I'd do: Strip whitespace and upcase the text.

You can do a lot of basic cleanup like this by applying Python's built-in string methods to the `str` attribute of a column.

👉 For more information on Python string methods, check out this notebook.

To start with, let's create a new column, `destination_clean`, with a stripped/uppercase version of the destination data.

Note: Outside of pandas, you can use "method chaining" to apply multiple transformations to a string, like this: `'  My String'.upper().strip()`.

When you're chaining string methods on the `str` attribute of a pandas column series, though, it doesn't work like that -- you have to call `str` after each method call. In other words:

```
# this will throw an error
junkets['destination_clean'] = junkets['Destination'].str.upper().strip()

# this will work
junkets['destination_clean'] = junkets['Destination'].str.upper().str.strip()
```

In [ ]:
```
junkets['destination_clean'] = junkets['Destination'].str.upper().str.strip()
```

In [ ]:
```
junkets.head()
```

Now let's run `value_counts()` again to see if that helped at all.

In [ ]:
```
junkets['destination_clean'].value_counts().sort_index()
```

That eliminated a handful of problems. Now comes the tedious work of identifying entries to find and replace.

Bulk-replacing values with other values

If we were at this point in Excel, we'd scroll through the list of unique names and start making notes of what we need to change. Same story here.

Let's loop over a sorted list of `unique()` destinations and `print()` each one.

👉 For a refresher on *for loops*, see this notebook.

In [ ]:
```
for destination in sorted(junkets.destination_clean.unique()):
    print(destination)
```

And here is where we're going to start encoding our editorial choices. "Ames, IA" or "Ames, Iowa"? "Baku, Azerjaijan," or "Baku, Republic of Azerbaijan"? Etc.

There are several ways we could structure this data, but a dictionary sounds like it'd be the most fun, so let's do that. Each key will be a string that we'd like to replace; each value will be the string we'd like to replace it with. To get us started:

In [ ]:
```
typo_fixes = {
    'BAKU, AZERBIJAN': 'BAKU, AZERBAIJAN',
    'BAKU, REPUBLIC OF AZERBAIJAN': 'BAKU, AZERBAIJAN',
    'ADDIS, ETHIOPIA': 'ADDIS ABABA, ETHIOPIA',
    'ANKEY, IA': 'ANKENY, IA'
}
```

... and so on. (This is tedious work, and -- again -- tools like OpenRefine make this process somewhat less tedious. But if you have a long-term project that involves data that will be updated regularly, and it's worth putting in the time to make sure the data are cleaned the same way each time, you can do it all in pandas.)

👉 For more information on dictionaries, check out this notebook.

Here's how we might *apply* our bulk find-and-replace dictionary:

In [ ]:
```
def find_replace_destination(row):
    '''Given a row of data, see if the value is a typo to be replaced'''

    # get the clean destination value
    dest = row['destination_clean']

    # try to look it up in the `typo_fixes` dictionary
    # the `get()` method will return None if it's not there
    typo = typo_fixes.get(dest)

    # then we can test to see if `get()` got an item out of the dictionary (True)
    # or if it returned None (False)
    if typo:
        # if it found an entry in our dictionary,
        # return the value from that key/value pair
        return typo_fixes[dest]
    # otherwise
    else:
        # return the original destination string
        return dest

# apply the function and overwrite our working "clean' column"
junkets['destination_clean'] = junkets.apply(find_replace_destination, axis=1)
```

In [ ]:
```
junkets.head()
```

👉 For more information on writing your own functions, check out this notebook.

👉 For more information on applying functions to a pandas data frame, check out this notebook.

Nonstandard values to represent null entries

Data creators may express null values in a variety of ways -- `''`, `'n/a'`, `NA`, `.`, etc. But for your purposes, you want pandas to read them all as `NaN`, so you can take advantage of methods like `isnull()` in your analysis.

If you've already done some exploratory analysis, you can specify the `na_values` argument when you read in the data -- you can supply a *single* value that should be interpreted as null, or you can hand off a list `[]` of values.

As an example, let's take a look at some EPA air quality data from Ohio:

In [ ]:
```
air_quality = pd.read_excel('../data/epa.xlsx')
```

In [ ]:
```
air_quality.head()
```

After conferring with the source of this data, the dots `.` represent "no observation" -- a null value. Let's

try reading that in again, this time specifying `na_values`:

In [ ]:
```
air_quality = pd.read_excel('../data/epa.xlsx',
                            na_values='.')
```

In [ ]:
```
air_quality.head()
```

You want to replace null values with 0, or something else

Use the `fillna()` method on a data frame or column series to fill null values with some other value.

Let's say our reporting had shown that the dots in the air quality data weren't, in fact null. Let's say they were actually supposed to be zeroes. Here's how we'd fix that:

In [ ]:
```
air_quality.fillna(0)
```

You have duplicate rows

If your data have rows that are incorrectly duplicated, you use the data frame method `drop_duplicates()` to delete the duplicates.

(This assumes, of course, that you've done sufficient reporting to feel confident that the duplicated rows aren't in there legitimately.)

Let's look at some fake data to show how this'd work:

In [ ]:
```
fake_data = [
    {'id': 12345, 'name': 'Sally', 'position': 'Editor', 'org': 'Some News Organization'},
    {'id': 54321, 'name': 'George', 'position': 'Reporter', 'org': 'Some Other News Organi
    {'id': 12345, 'name': 'Sally', 'position': 'Editor', 'org': 'Some News Organization'},
    {'id': 49382, 'name': 'Sally', 'position': 'Editor', 'org': 'Some News Organization'},
    {'id': 39331, 'name': 'Pat', 'position': 'Producer', 'org': 'Some Other Other News Org
]

fake_df = pd.DataFrame(fake_data)
```

In [ ]:
```
fake_df.head()
```

Before you drop anything, you'd probably want to check for duplicate rows. You can do that by filtering your data using the `duplicated()` method.

👉 For more details on filtering data in pandas, see this notebook.

In [ ]:
```
fake_df[fake_df.duplicated()]
```

This is showing us a row where every value in every column matches exactly the values in at least one other row. So we've done the reporting to show that we need to cut the duplicates here.

The `drop_duplicates()` method gives you control over *how* this happens:

- You can drop *all* duplicate rows, or keep just the first instance (this is the default behavior), or the last instance

- You can drop rows where just the values in certain columns are duplicated

Here are a few examples:

In [ ]:
```
# default behavior -- duplicate rows must match exactly
fake_df.drop_duplicates()
```

In [ ]:
```
# drop rows where the values in name, org and position are identical
fake_df.drop_duplicates(subset=['name', 'org', 'position'])
```

Our original data frame is unchanged:

In [ ]:
```
fake_df
```

That's because we didn't specify `in_place=True` as an argument.

You can take one of two approaches here. You could alter your original dataframe -- the code would look like this:

```
# drop rows where the values in name, org and position are identical
fake_df.drop_duplicates(subset=['name', 'org', 'position'], inplace=True)
```

-- or you could "save" the resulting deduplicated data frame as a new variable, like this:

```
# drop rows where the values in name, org and position are identical
deduped = fake_df.drop_duplicates(subset=['name', 'org', 'position'])
```

I prefer the latter approach because I like to leave the original data as untouched as possible, working up to successively cleaner and more analyze-able data frames as I go. I also find this approach is easier to follow when I come back to it after a few weeks or months of inaction.

You have empty rows

To drop rows or columns whose values are all `NA`, use `dropna()`.

Specifying `axis=1` will drop empty *columns*; `axis=0` will drop empty *rows*.

In [ ]:
```
import numpy as np

fake_df = fake_df.append({'id': np.nan, 'name': np.nan, 'position': np.nan, 'org': np.nan}
```

In [ ]:
```
fake_df
```

In [ ]:
```
fake_df.dropna(axis=0)
```

Further reading

This just scratches the surface of what you can do in pandas. Here are some other resources to check out:

- Pythonic Data Cleaning With NumPy and Pandas
- pandas official list of tutorials
- Karrie Kehoe's guide to cleaning data in pandas
- Data cleaning with Python