Data Cleaning and Preparation

Part 2

Data Transformation

Part 1

Removing Duplicates

- Duplicate rows may be found in a DataFrame for any number of reasons.
- Here is an example:

```
In [40]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
                              'k2': [1, 1, 2, 3, 3, 4, 4]})

In [41]: data
Out[41]:
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

- The DataFrame method `duplicated` returns a boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not:

```
In [42]: data.duplicated()

Out[42]: 0    False
         1    False
         2    False
         3    False
         4    False
         5    False
         6     True
         dtype: bool
```

- Relatedly, `drop_duplicates` returns a DataFrame where the `duplicated` array is `False`:

```
In [43]: data.drop_duplicates()
Out[43]:
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

- Both of these methods by default consider all of the columns; alternatively, you can specify any subset of them to detect duplicates.
- Suppose we had an additional column of values and wanted to filter duplicates only based on the `'k1'` column:

```
In [44]: data['v1'] = range(7)

In [45]: data
Out[45]:
```

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

```
In [46]: data.drop_duplicates(['k1'])
Out[46]:
```

	k1	k2	v1
0	one	1	0
1	two	1	1

- `duplicated` and `drop_duplicates` by default keep the first observed value combination.
- Passing `keep='last'` will return the last one:

```
In [47]: data
Out[47]:
```

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

```
In [48]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[48]:
```

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
6	two	4	6

Transforming Data Using a Function or Mapping

- For many datasets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame.

- Consider the following hypothetical data collected about various kinds of meat:

```
In [49]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
                                       'Pastrami', 'corned beef', 'Bacon',
                                       'pastrami', 'honey ham', 'nova lox'],
                              'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [50]: data
Out[50]:
```

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

- Suppose you wanted to add a column indicating the type of animal that each food came from.
- Let’s write down a mapping of each distinct meat type to the kind of animal:

```
In [51]: meat_to_animal = {
           'bacon': 'pig',
           'pulled pork': 'pig',
           'pastrami': 'cow',
           'corned beef': 'cow',
           'honey ham': 'pig',
           'nova lox': 'salmon'
         }
```

- The `map` method on a Series accepts a function or dict-like object containing a mapping, but here we have a small problem in that some of the meats are capitalized and others are not.
- Thus, we need to convert each value to lowercase using the `str.lower` Series method:

```
In [52]: lowercased = data['food'].str.lower()

In [53]: lowercased
Out[53]: 0          bacon
         1    pulled pork
         2          bacon
         3       pastrami
         4    corned beef
         5          bacon
         6       pastrami
         7      honey ham
         8       nova lox
         Name: food, dtype: object
```

```
In [54]: data['animal'] = lowercased.map(meat_to_animal)
```

```
In [55]: data
```

Out[55]:

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

- We could also have passed a function that does all the work:

```
In [56]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[56]: 0       pig
         1       pig
         2       pig
         3       cow
         4       cow
         5       pig
         6       cow
         7       pig
         8    salmon
         Name: food, dtype: object
```

Replacing Values

- Filling in missing data with the `fillna` method is a special case of more general value replacement.
- As you’ve already seen, `map` can be used to modify a subset of values in an object but `replace` provides a simpler and more flexible way to do so.

```
In [57]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [58]: data
Out[58]: 0       1.0
         1    -999.0
         2       2.0
         3    -999.0
         4   -1000.0
         5       3.0
         dtype: float64
```

- The `−999` values might be sentinel values for missing data.
- To replace these with NA values that pandas understands, we can use `replace`, producing a new Series (unless you pass `inplace=True`):

```
In [59]: data.replace(-999, np.nan)
Out[59]: 0       1.0
         1       NaN
         2       2.0
         3       NaN
         4   -1000.0
         5       3.0
         dtype: float64
```

- If you want to replace multiple values at once, you instead pass a list and then the substitute value:

```
In [60]: data.replace([-999, -1000], np.nan)
Out[60]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         4    NaN
         5    3.0
         dtype: float64
```

- To use a different replacement for each value, pass a list of substitutes:

```
In [61]: data.replace([-999, -1000], [np.nan, 0])
Out[61]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         4    0.0
         5    3.0
         dtype: float64
```

- The argument passed can also be a dict:

```
In [62]: data.replace({-999: np.nan, -1000: 0})
Out[62]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         4    0.0
         5    3.0
         dtype: float64
```

Renaming Axis Indexes

- Like values in a Series, axis labels can be similarly transformed by a function or mapping of some form to produce new, differently labeled objects.
- You can also modify the axes in-place without creating a new data structure.

```
In [63]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
                             index=['Ohio', 'Colorado', 'New York'],
                             columns=['one', 'two', 'three', 'four'])
```

- Like a Series, the axis indexes have a `map` method:

```
In [64]: data
Out[64]:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

```
In [65]: transform = lambda x: x[:4].upper()

In [66]: data.index.map(transform)
Out[66]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

- You can assign to index, modifying the DataFrame in-place:

```
In [67]: data.index = data.index.map(transform)

In [68]: data
Out[68]:
```

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

- If you want to create a transformed version of a dataset without modifying the original, a useful method is `rename`:

```
In [69]: data.rename(index=str.title, columns=str.upper)
Out[69]:
```

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3
Colo	4	5	6	7
New	8	9	10	11

- Notably, `rename` can be used in conjunction with a dict-like object providing new values for a subset of the axis labels:

In [70]:
```
data.rename(index={'OHIO': 'INDIANA'},
            columns={'three': 'peekaboo'})
```

Out[70]:

	one	two	peekaboo	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

- `rename` saves you from the chore of copying the DataFrame manually and assigning to its `index` and `columns` attributes.
- Should you wish to modify a dataset in-place, pass `inplace=True`:

```
In [71]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [72]: data
Out[72]:
```

	one	two	three	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11