Chapter 3. Data Wrangling

3.0 Introduction

Data wrangling is a broad term used, often informally, to describe the process of transforming raw data into a clean, organized format ready for use. For us, data wrangling is only one step in preprocessing our data, but it is an important step.

The most common data structure used to "wrangle" data is the dataframe, which can be both intuitive and incredibly versatile. Dataframes are tabular, meaning that they are based on rows and columns like you would see in a spreadsheet. Here is a dataframe created from data about passengers on the *Titanic*:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data as a dataframe
dataframe = pd.read_csv(url)

# Show first five rows
dataframe.head(5)
```

	Name	PClass	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	1st	29.00	female	1
1	Allison, Miss Helen Loraine	1st	2.00	female	0
2	Allison, Mr Hudson Joshua Creighton	1st	30.00	male	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.00	female	0
4	Allison,	1st	0.92	male	1

Master
Hudson
Trevor

There are three important things to notice in this dataframe.

First, in a dataframe each row corresponds to one observation (e.g., a passenger) and each column corresponds to one feature (gender, age, etc.). For example, by looking at the first observation we can see that Miss Elisabeth Walton Allen stayed in first class, was 29 years old, was female, and survived the disaster.

Second, each column contains a name (e.g., `Name`, `PClass`, `Age`) and each row contains an index number (e.g., `0` for the lucky Miss Elisabeth Walton Allen). We will use these to select and manipulate observations and features.

Third, two columns, `Sex` and `SexCode`, contain the same information in different formats. In `Sex`, a woman is indicated by the string `female`, while in `SexCode`, a woman is indicated by using the integer `1`. We will want all our features to be unique, and therefore we will need to remove one of these columns.

In this chapter, we will cover a wide variety of techniques to manipulate dataframes using the pandas library with the goal of creating a clean, well-structured set of observations for further preprocessing.

3.1 Creating a Dataframe

Problem

You want to create a new dataframe.

Solution

pandas has many methods for creating a new DataFrame object. One easy method is to instantiate a `DataFrame` using a Python dictionary. In the dictionary, each key is a column name and the value is a list, where each item corresponds to a row:

```
# Load library
import pandas as pd

# Create a dictionary
dictionary = {
  "Name": ['Jacky Jackson', 'Steven Stevenson'],
  "Age": [38, 25],
  "Driver": [True, False]
}

# Create DataFrame
```

```
dataframe = pd.DataFrame(dictionary)

# Show DataFrame
dataframe
```

	Name	Age	Driver
0	Jacky Jackson	38	True
1	Steven Stevenson	25	False

It's easy to add new columns to any dataframe using a list of values:

```
# Add a column for eye color
dataframe["Eyes"] = ["Brown", "Blue"]

# Show DataFrame
dataframe
```

	Name	Age	Driver	Eyes
0	Jacky Jackson	38	True	Brown
1	Steven Stevenson	25	False	Blue

Discussion

pandas offers what can feel like an infinite number of ways to create a DataFrame. In the real world, creating an empty DataFrame and then populating it will almost never happen. Instead, our DataFrames will be created from real data we have loaded from other sources (e.g., a CSV file or database).

3.2 Getting Information about the Data

Problem

You want to view some characteristics of a DataFrame.

Solution

One of the easiest things we can do after loading the data is view the first few rows using `head`:

```
# Load library
import pandas as pd
```

```
# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Show two rows
dataframe.head(2)
```

	Name	PClass	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0

We can also take a look at the number of rows and columns:

```
# Show dimensions
dataframe.shape
(1313, 6)
```

We can get descriptive statistics for any numeric columns using `describe`:

```
# Show statistics
dataframe.describe()
```

	Age	Survived	SexCode
count	756.000000	1313.000000	1313.000000
mean	30.397989	0.342727	0.351866
std	14.259049	0.474802	0.477734
min	0.170000	0.000000	0.000000
25%	21.000000	0.000000	0.000000
50%	28.000000	0.000000	0.000000
75%	39.000000	1.000000	1.000000
max	71.000000	1.000000	1.000000

Additionally, the `info` method can show some helpful information:

```
# Show info
dataframe.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1313 entries, 0 to 1312
Data columns (total 6 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Name      1313 non-null   object
 1   PClass    1313 non-null   object
 2   Age       756 non-null    float64
 3   Sex       1313 non-null   object
 4   Survived  1313 non-null   int64
 5   SexCode   1313 non-null   int64
dtypes: float64(1), int64(2), object(3)
memory usage: 61.7+ KB
```

Discussion

After we load some data, it's a good idea to understand how it's structured and what kind of information it contains. Ideally, we would view the full data directly. But with most real-world cases, the data could have thousands to hundreds of thousands to millions of rows and columns. Instead, we have to rely on pulling samples to view small slices and calculating summary statistics of the data.

In our solution, we are using a toy dataset of the passengers of the *Titanic*. Using `head`, we can look at the first few rows (five by default) of the data. Alternatively, we can use `tail` to view the last few rows. With `shape` we can see how many rows and columns our DataFrame contains. With `describe` we can see some basic descriptive statistics for any numerical column. And, finally, `info` displays a number of helpful data points about the DataFrame, including index and column data types, non-null values, and memory usage.

It is worth noting that summary statistics do not always tell the full story. For example, pandas treats the columns `Survived` and `SexCode` as numeric columns because they contain 1s and 0s. However, in this case the numerical values represent categories. For example, if `Survived` equals 1, it indicates that the passenger survived the disaster. For this reason, some of the summary statistics provided don't make sense, such as the standard deviation of the `SexCode` column (an indicator of the passenger's gender).

3.3 Slicing DataFrames

Problem

You need to select a specific subset data or slices of a DataFrame.

Solution

Use `loc` or `iloc` to select one or more rows or values:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Select first row
dataframe.iloc[0]
```

```
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
Sex                               female
Survived                               1
SexCode                                1
Name: 0, dtype: object
```

We can use `:` to define the slice of rows we want, such as selecting the second, third, and fourth rows:

```
# Select three rows
dataframe.iloc[1:4]
```

	Name	PClass	Age	Sex	Survive
1	Allison, Miss Helen Loraine	1st	2.0	female	0
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.0	female	0

We can even use it to get all rows up to a point, such as all rows up to and including the fourth row:

```
# Select four rows
dataframe.iloc[:4]
```

	Name	PClass	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.0	female	0

DataFrames do not need to be numerically indexed. We can set the index of a DataFrame to any value where the value is unique to each row. For example, we can set the index to be passenger names and then select rows using a name:

```
# Set index
dataframe = dataframe.set_index(dataframe['Name'])

# Show row
dataframe.loc['Allen, Miss Elisabeth Walton']
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
Sex                               female
Survived                               1
SexCode                                1
Name: Allen, Miss Elisabeth Walton, dtype: object
```

Discussion

All rows in a pandas DataFrame have a unique index value. By default, this index is an integer indicating the row position in the DataFrame; however, it does not have to be. DataFrame indexes can be set to be unique alphanumeric strings or customer numbers. To select individual rows and slices of rows, pandas provides two methods:

- `loc` is useful when the index of the DataFrame is a label (e.g., a string).
- `iloc` works by looking for the position in the DataFrame. For example, `iloc[0]` will return the first row regardless of whether the index is an integer or a label.

It is useful to be comfortable with both `loc` and `iloc` since they will come up a lot during data cleaning.

3.4 Selecting Rows Based on Conditionals

Problem

You want to select DataFrame rows based on some condition.

Solution

This can be done easily in pandas. For example, if we wanted to select all the women on the *Titanic*:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Show top two rows where column 'sex' is 'female'
dataframe[dataframe['Sex'] == 'female'].head(2)
```

	Name	PClass	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0

Take a moment to look at the format of this solution. Our conditional statement is `dataframe['Sex'] == 'female'`; by wrapping that in `dataframe[]` we are telling pandas to "select all the rows in the DataFrame where the value of `dataframe['Sex']` is `'female'`." These conditions result in a pandas series of booleans.

Multiple conditions are easy as well. For example, here we select all the rows where the passenger is a female 65 or older:

```
# Filter rows
dataframe[(dataframe['Sex'] == 'female') & (dataframe['Age'] >= 65)]
```

	Name	PClass	Age	Sex	Survive
73	Crosby, Mrs Edward Gifford (Catherine Elizabet...	1st	69.0	female	1

Discussion

Conditionally selecting and filtering data is one of the most common tasks in data wrangling. You rarely want all the raw data from the source; instead, you are interested in only some subset of it. For example, you might only be interested in stores in certain states or the records of patients over a certain age.

3.5 Sorting Values

Problem

You need to sort a dataframe by the values in a column.

Solution

Use the pandas `sort_values` function:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Sort the dataframe by age, show two rows
dataframe.sort_values(by=["Age"]).head(2)
```

	Name	PClass	Age	Sex	Survive
763	Dean, Miss Elizabeth Gladys (Millvena)	3rd	0.17	female	1
751	Danbom, Master Gilbert	3rd	0.33	male	0

	Sigvard Emanuel

Discussion

During data analysis and exploration, it's often useful to sort a DataFrame by a particular column or set of columns. The `by` argument to `sort_values` takes a list of columns by which to sort the DataFrame and will sort based on the order of column names in the list.

By default, the `ascending` argument is set to `True`, so it will sort the values lowest to highest. If we wanted the oldest passengers instead of the youngest, we could set it to `False`.

3.6 Replacing Values

Problem

You need to replace values in a DataFrame.

Solution

The pandas `replace` method is an easy way to find and replace values. For example, we can replace any instance of `"female"` in the `Sex` column with `"Woman"`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Replace values, show two rows
dataframe['Sex'].replace("female", "Woman").head(2)
0    Woman
1    Woman
Name: Sex, dtype: object
```

We can also replace multiple values at the same time:

```
# Replace "female" and "male" with "Woman" and "Man"
dataframe['Sex'].replace(["female", "male"], ["Woman", "Man"]).head(5)
0    Woman
1    Woman
2      Man
3    Woman
4      Man
Name: Sex, dtype: object
```

We can also find and replace across the entire `DataFrame` object by specifying the whole dataframe instead of a single column:

```
# Replace values, show two rows
dataframe.replace(1, "One").head(2)
```

	Name	PClass	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	1st	29	female	One
1	Allison, Miss Helen Loraine	1st	2	female	0

`replace` also accepts regular expressions:

```
# Replace values, show two rows
dataframe.replace(r"1st", "First", regex=True).head(2)
```

	Name	PClass	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	First	29.0	female	1
1	Allison, Miss Helen Loraine	First	2.0	female	0

Discussion

`replace` is a tool we use to replace values. It is simple and yet has the powerful ability to accept regular expressions.

3.7 Renaming Columns

Problem

You want to rename a column in a pandas DataFrame.

Solution

Rename columns using the `rename` method:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Rename column, show two rows
dataframe.rename(columns={'PClass': 'Passenger Class'}).head(2)
```

	Name	Passenger Class	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0

Notice that the `rename` method can accept a dictionary as a parameter. We can use the dictionary to change multiple column names at once:

```
# Rename columns, show two rows
dataframe.rename(columns={'PClass': 'Passenger Class', 'Sex': 'Gender'}).head(2)
```

	Name	Passenger Class	Age	Gender	Survive
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0

Discussion

Using `rename` with a dictionary as an argument to the `columns` parameter is my preferred way to rename columns because it works with any number of columns. If we want to rename all columns at once, this helpful snippet of code creates a dictionary with the old column names as keys and empty strings as values:

```
# Load library
```

```
import collections

# Create dictionary
column_names = collections.defaultdict(str)

# Create keys
for name in dataframe.columns:
    column_names[name]

# Show dictionary
column_names
defaultdict(str,
            {'Age': '',
             'Name': '',
             'PClass': '',
             'Sex': '',
             'SexCode': '',
             'Survived': ''})
```

3.8 Finding the Minimum, Maximum, Sum, Average, and Count

Problem

You want to find the min, max, sum, average, or count of a numeric column.

Solution

pandas comes with some built-in methods for commonly used descriptive statistics such as `min`, `max`, `mean`, `sum`, and `count`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Calculate statistics
print('Maximum:', dataframe['Age'].max())
print('Minimum:', dataframe['Age'].min())
print('Mean:', dataframe['Age'].mean())
print('Sum:', dataframe['Age'].sum())
print('Count:', dataframe['Age'].count())
Maximum: 71.0
Minimum: 0.17
Mean: 30.397989417989415
Sum: 22980.879999999997
Count: 756
```

Discussion

In addition to the statistics used in the solution, pandas offers variance (`var`), standard deviation (`std`), kurtosis (`kurt`), skewness (`skew`), standard error of the mean (`sem`), mode (`mode`), median (`median`), value counts, and a number of others.

Furthermore, we can also apply these methods to the whole DataFrame:

```
# Show counts
dataframe.count()
Name        1313
PClass      1313
Age          756
Sex         1313
Survived    1313
SexCode     1313
dtype: int64
```

3.9 Finding Unique Values

Problem

You want to select all unique values in a column.

Solution

Use `unique` to view an array of all unique values in a column:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Select unique values
dataframe['Sex'].unique()
array(['female', 'male'], dtype=object)
```

Alternatively, `value_counts` will display all unique values with the number of times each value appears:

```
# Show counts
dataframe['Sex'].value_counts()
male      851
female    462
Name: Sex, dtype: int64
```

Discussion

Both `unique` and `value_counts` are useful for manipulating and exploring categorical columns. Very often in categorical columns there will be classes that need to be handled in the data wrangling phase. For example, in the *Titanic* dataset, `PClass` is a column indicating the class of a passenger's ticket. There were three classes on the *Titanic*; however, if we use `value_counts` we can see a problem:

```
# Show counts
dataframe['PClass'].value_counts()
3rd    711
1st    322
2nd    279
*        1
Name: PClass, dtype: int64
```

While almost all passengers belong to one of three classes as expected, a single passenger has the class `*`. There are a number of strategies for handling this type of issue, which we will address in Chapter 5, but for now just realize that "extra" classes are common in categorical data and should not be ignored.

Finally, if we simply want to count the number of unique values, we can use `nunique`:

```
# Show number of unique values
dataframe['PClass'].nunique()
4
```

3.10 Handling Missing Values

Problem

You want to select missing values in a DataFrame.

Solution

`isnull` and `notnull` return booleans indicating whether a value is missing:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

## Select missing values, show two rows
dataframe[dataframe['Age'].isnull()].head(2)
```

	Name	PClass	Age	Sex	Survive
12	Aubert, Mrs Leontine Pauline	1st	NaN	female	1
13	Barkworth, Mr Algernon H	1st	NaN	male	1

Discussion

Missing values are a ubiquitous problem in data wrangling, yet many underestimate the difficulty of working with missing data. pandas uses NumPy's `NaN` (Not a Number) value to denote missing values, but it is important to note that `NaN` is not fully implemented natively in pandas. For example, if we wanted to replace all strings containing `male` with missing values, we get an error:

```
# Attempt to replace values with NaN
dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-7-5682d714f87d> in <module>()
      1 # Attempt to replace values with NaN
----> 2 dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)


NameError: name 'NaN' is not defined
---------------------------------------------------------------------------
```

To have full functionality with `NaN` we need to import the NumPy library first:

```
# Load library
import numpy as np

# Replace values with NaN
dataframe['Sex'] = dataframe['Sex'].replace('male', np.nan)
```

Oftentimes a dataset uses a specific value to denote a missing observation, such as `NONE`, `-999`, or `...`. The pandas `read_csv` function includes a parameter allowing us to specify the values used to indicate missing values:

```
# Load data, set missing values
dataframe = pd.read_csv(url, na_values=[np.nan, 'NONE', -999])
```

We can also use the pandas `fillna` function to impute the missing values of a column. Here, we show the places where `Age` is null using the `isna` function and then fill those values with

the mean age of passengers.

```
# Get a single null row
null_entry = dataframe[dataframe["Age"].isna()].head(1)

print(null_entry)
```

	Name	PClass	Age	Sex	Survive
12	Aubert, Mrs Leontine Pauline	1st	NaN	female	1

```
# Fill all null values with the mean age of passengers
null_entry.fillna(dataframe["Age"].mean())
```

	Name	PClass	Age	Sex	Survive
12	Aubert, Mrs Leontine Pauline	1st	30.397989	female	1

3.11 Deleting a Column

Problem

You want to delete a column from your DataFrame.

Solution

The best way to delete a column is to use `drop` with the parameter `axis=1` (i.e., the column axis):

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Delete column
dataframe.drop('Age', axis=1).head(2)
```

	Name	PClass	Sex	Survived	SexCod
0	Allen, Miss Elisabeth Walton	1st	female	1	1
1	Allison, Miss Helen Loraine	1st	female	0	1

You can also use a list of column names as the main argument to drop multiple columns at once:

```
# Drop columns
dataframe.drop(['Age', 'Sex'], axis=1).head(2)
```

	Name	PClass	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	1	1
1	Allison, Miss Helen Loraine	1st	0	1

If a column does not have a name (which can sometimes happen), you can drop it by its column index using `dataframe.columns`:

```
# Drop column
dataframe.drop(dataframe.columns[1], axis=1).head(2)
```

	Name	Age	Sex	Survived	SexCod
0	Allen, Miss Elisabeth Walton	29.0	female	1	1
1	Allison, Miss Helen Loraine	2.0	female	0	1

Discussion

`drop` is the idiomatic method of deleting a column. An alternative method is `del dataframe['Age']`, which works most of the time but is not recommended because of how it is called within pandas (the details of which are outside the scope of this book).

I recommend that you avoid using the pandas `inplace=True` argument. Many pandas

methods include an `inplace` parameter that, when set to `True`, edits the DataFrame directly. This can lead to problems in more complex data processing pipelines because we are treating the DataFrames as mutable objects (which they technically are). I recommend treating DataFrames as immutable objects. For example:

```
# Create a new DataFrame
dataframe_name_dropped = dataframe.drop(dataframe.columns[0], axis=1)
```

In this example, we are not mutating the DataFrame `dataframe` but instead are making a new DataFrame that is an altered version of `dataframe` called `dataframe_name_dropped`. If you treat your DataFrames as immutable objects, you will save yourself a lot of headaches down the road.

3.12 Deleting a Row

Problem

You want to delete one or more rows from a DataFrame.

Solution

Use a boolean condition to create a new DataFrame excluding the rows you want to delete:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Delete rows, show first three rows of output
dataframe[dataframe['Sex'] != 'male'].head(3)
```

	Name	PClass	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0
3	Allison, Mrs Hudson JC	1st	25.00	female	0

	(Bessie Waldo Daniels)

Discussion

While technically you can use the `drop` method (for example, `dataframe.drop([0, 1], axis=0)` to drop the first two rows), a more practical method is simply to wrap a boolean condition inside `dataframe[]`. This enables us to use the power of conditionals to delete either a single row or (far more likely) many rows at once.

We can use boolean conditions to easily delete single rows by matching a unique value:

```
# Delete row, show first two rows of output
dataframe[dataframe['Name'] != 'Allison, Miss Helen Loraine'].head(2)
```

	Name	PClass	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0

We can even use it to delete a single row by specifying the row index:

```
# Delete row, show first two rows of output
dataframe[dataframe.index != 0].head(2)
```

	Name	PClass	Age	Sex	Survive
1	Allison, Miss Helen Loraine	1st	2.0	female	0
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0

3.13 Dropping Duplicate Rows

Problem

You want to drop duplicate rows from your DataFrame.

Solution

Use `drop_duplicates`, but be mindful of the parameters:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Drop duplicates, show first two rows of output
dataframe.drop_duplicates().head(2)
```

	Name	PClass	Age	Sex	Survive
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0

Discussion

A keen reader will notice that the solution didn't actually drop any rows:

```
# Show number of rows
print("Number Of Rows In The Original DataFrame:", len(dataframe))
print("Number Of Rows After Deduping:", len(dataframe.drop_duplicates()))
Number Of Rows In The Original DataFrame: 1313
Number Of Rows After Deduping: 1313
```

This is because `drop_duplicates` defaults to dropping only rows that match perfectly across all columns. Because every row in our DataFrame is unique, none will be dropped. However, often we want to consider only a subset of columns to check for duplicate rows. We can accomplish this using the `subset` parameter:

```
# Drop duplicates
dataframe.drop_duplicates(subset=['Sex'])
```

	Name	PClass	Age	Sex	Survive

0	Allen, Miss Elisabeth Walton	1st	29.0	female	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0

Take a close look at the preceding output: we told `drop_duplicates` to only consider any two rows with the same value for `Sex` to be duplicates and to drop them. Now we are left with a DataFrame of only two rows: one woman and one man. You might be asking why `drop_duplicates` decided to keep these two rows instead of two different rows. The answer is that `drop_duplicates` defaults to keeping the first occurrence of a duplicated row and dropping the rest. We can control this behavior using the `keep` parameter:

```
# Drop duplicates
dataframe.drop_duplicates(subset=['Sex'], keep='last')
```

	Name	PClass	Age	Sex	Survive
1307	Zabour, Miss Tamini	3rd	NaN	female	0
1312	Zimmerman, Leo	3rd	29.0	male	0

A related method is `duplicated`, which returns a boolean series denoting whether a row is a duplicate or not. This is a good option if you don't want to simply drop duplicates:

```
dataframe.duplicated()
0       False
1       False
2       False
3       False
4       False
        ...
1308    False
1309    False
1310    False
1311    False
1312    False
Length: 1313, dtype: bool
```

3.14 Grouping Rows by Values

Problem

You want to group individual rows according to some shared value.

Solution

`groupby` is one of the most powerful features in pandas:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Group rows by the values of the column 'Sex', calculate mean # of each group
dataframe.groupby('Sex').mean(numeric_only=True)
```

Sex	Age	Survived	SexCode
female	29.396424	0.666667	1.0
male	31.014338	0.166863	0.0

Discussion

`groupby` is where data wrangling really starts to take shape. It is very common to have a DataFrame where each row is a person or an event and we want to group them according to some criterion and then calculate a statistic. For example, you can imagine a DataFrame where each row is an individual sale at a national restaurant chain and we want the total sales per restaurant. We can accomplish this by grouping rows by individual restaurants and then calculating the sum of each group.

Users new to `groupby` often write a line like this and are confused by what is returned:

```
# Group rows
dataframe.groupby('Sex')
<pandas.core.groupby.DataFrameGroupBy object at 0x10efacf28>
```

Why didn't it return something more useful? The reason is that `groupby` needs to be paired with some operation that we want to apply to each group, such as calculating an aggregate statistic (e.g., mean, median, sum). When talking about grouping we often use shorthand and say "group by gender," but that is incomplete. For grouping to be useful, we need to group by something and then apply a function to each of those groups:

```
# Group rows, count rows
```

```
dataframe.groupby('Survived')['Name'].count()
Survived
0    863
1    450
Name: Name, dtype: int64
```

Notice `Name` added after the `groupby`? That is because particular summary statistics are meaningful only to certain types of data. For example, while calculating the average age by gender makes sense, calculating the total age by gender does not. In this case, we group the data into survived or not, and then count the number of names (i.e., passengers) in each group.

We can also group by a first column, then group that grouping by a second column:

```
# Group rows, calculate mean
dataframe.groupby(['Sex','Survived'])['Age'].mean()
Sex     Survived
female  0           24.901408
        1           30.867143
male    0           32.320780
        1           25.951875
Name: Age, dtype: float64
```

3.15 Grouping Rows by Time

Problem

You need to group individual rows by time periods.

Solution

Use `resample` to group rows by chunks of time:

```
# Load libraries
import pandas as pd
import numpy as np

# Create date range
time_index = pd.date_range('06/06/2017', periods=100000, freq='30S')

# Create DataFrame
dataframe = pd.DataFrame(index=time_index)

# Create column of random values
dataframe['Sale_Amount'] = np.random.randint(1, 10, 100000)

# Group rows by week, calculate sum per week
dataframe.resample('W').sum()
```

	Sale_Amount
2017-06-11	86423

2017-06-18	101045
2017-06-25	100867
2017-07-02	100894
2017-07-09	100438
2017-07-16	10297

Discussion

Our standard *Titanic* dataset does not contain a datetime column, so for this recipe we have generated a simple DataFrame where each row is an individual sale. For each sale we know its date and time and its dollar amount (this data isn't realistic because the sales take place precisely 30 seconds apart and are exact dollar amounts, but for the sake of simplicity let's pretend).

The raw data looks like this:

```
# Show three rows
dataframe.head(3)
```

	Sale_Amount
2017-06-06 00:00:00	7
2017-06-06 00:00:30	2
2017-06-06 00:01:00	7

Notice that the date and time of each sale is the index of the DataFrame; this is because `resample` requires the index to be a datetime-like value.

Using `resample` we can group the rows by a wide array of time periods (offsets) and then we can calculate statistics on each time group:

```
# Group by two weeks, calculate mean
dataframe.resample('2W').mean()
```

	Sale_Amount
2017-06-11	5.001331
2017-06-25	5.007738

2017-07-09	4.993353
2017-07-23	4.950481

```
# Group by month, count rows
dataframe.resample('M').count()
```

	Sale_Amount
2017-06-30	72000
2017-07-31	28000

You might notice that in the two outputs the datetime index is a date even though we are grouping by weeks and months, respectively. The reason is that by default `resample` returns the label of the right "edge" (the last label) of the time group. We can control this behavior using the `label` parameter:

```
# Group by month, count rows
dataframe.resample('M', label='left').count()
```

	Sale_Amount
2017-05-31	72000
2017-06-30	28000

See Also

- List of pandas time offset aliases

3.16 Aggregating Operations and Statistics

Problem

You need to aggregate an operation over each column (or a set of columns) in a dataframe.

Solution

Use the pandas `agg` method. Here, we can easily get the minimum value of every column:

```
# Load library
import pandas as pd
```

```
# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Get the minimum of every column
dataframe.agg("min")
Name        Abbing, Mr Anthony
PClass                       *
Age                       0.17
Sex                     female
Survived                     0
SexCode                      0
dtype: object
```

Sometimes, we want to apply specific functions to specific sets of columns:

```
# Mean Age, min and max SexCode
dataframe.agg({"Age":["mean"], "SexCode":["min", "max"]})
```

	Age	SexCode
mean	30.397989	NaN
min	NaN	0.0
max	NaN	1.0

We can also apply aggregate functions to groups to get more specific, descriptive statistics:

```
# Number of people who survived and didn't survive in each class
dataframe.groupby(
    ["PClass","Survived"]).agg({"Survived":["count"]}
  ).reset_index()
```

PClass	Survived		Count
0	*	0	1
1	1st	0	129
2	1st	1	193
3	2nd	0	160
4	2nd	1	119

5	3rd	0	573
6	3rd	1	138

Discussion

Aggregate functions are especially useful during exploratory data analysis to learn information about different subpopulations of data and the relationship between variables. By grouping the data and applying aggregate statistics, you can view patterns in the data that may prove useful during the machine learning or feature engineering process. While visual charts are also helpful, it's often useful to have such specific, descriptive statistics as a reference to better understand the data.

See Also

- pandas agg documentation

3.17 Looping over a Column

Problem

You want to iterate over every element in a column and apply some action.

Solution

You can treat a pandas column like any other sequence in Python and loop over it using the standard Python syntax:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Print first two names uppercased
for name in dataframe['Name'][0:2]:
    print(name.upper())
ALLEN, MISS ELISABETH WALTON
ALLISON, MISS HELEN LORAINE
```

Discussion

In addition to loops (often called `for` loops), we can also use list comprehensions:

```
# Show first two names uppercased
[name.upper() for name in dataframe['Name'][0:2]]
['ALLEN, MISS ELISABETH WALTON', 'ALLISON, MISS HELEN LORAINE']
```

Despite the temptation to fall back on `for` loops, a more Pythonic solution would use the pandas `apply` method, described in Recipe 3.18.

3.18 Applying a Function over All Elements in a Column

Problem

You want to apply some function over all elements in a column.

Solution

Use `apply` to apply a built-in or custom function on every element in a column:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Create function
def uppercase(x):
    return x.upper()

# Apply function, show two rows
dataframe['Name'].apply(uppercase)[0:2]
0    ALLEN, MISS ELISABETH WALTON
1    ALLISON, MISS HELEN LORAINE
Name: Name, dtype: object
```

Discussion

`apply` is a great way to do data cleaning and wrangling. It is common to write a function to perform some useful operation (separate first and last names, convert strings to floats, etc.) and then map that function to every element in a column.

3.19 Applying a Function to Groups

Problem

You have grouped rows using `groupby` and want to apply a function to each group.

Solution

Combine `groupby` and `apply`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Group rows, apply function to groups
dataframe.groupby('Sex').apply(lambda x: x.count())
```

Sex	Name	PClass	Age	Sex	Survive
female	462	462	288	462	462
male	851	851	468	851	851

Discussion

In Recipe 3.18 I mentioned `apply`. `apply` is particularly useful when you want to apply a function to groups. By combining `groupby` and `apply` we can calculate custom statistics or apply any function to each group separately.

3.20 Concatenating DataFrames

Problem

You want to concatenate two DataFrames.

Solution

Use `concat` with `axis=0` to concatenate along the row axis:

```
# Load library
import pandas as pd

# Create DataFrame
data_a = {'id': ['1', '2', '3'],
          'first': ['Alex', 'Amy', 'Allen'],
          'last': ['Anderson', 'Ackerman', 'Ali']}
dataframe_a = pd.DataFrame(data_a, columns = ['id', 'first', 'last'])

# Create DataFrame
data_b = {'id': ['4', '5', '6'],
```

```
            'first': ['Billy', 'Brian', 'Bran'],
            'last': ['Bonder', 'Black', 'Balwner']}
dataframe_b = pd.DataFrame(data_b, columns = ['id', 'first', 'last'])

# Concatenate DataFrames by rows
pd.concat([dataframe_a, dataframe_b], axis=0)
```

	id	first	last
0	1	Alex	Anderson
1	2	Amy	Ackerman
2	3	Allen	Ali
0	4	Billy	Bonder
1	5	Brian	Black
2	6	Bran	Balwner

You can use `axis=1` to concatenate along the column axis:

```
# Concatenate DataFrames by columns
pd.concat([dataframe_a, dataframe_b], axis=1)
```

	id	first	last	id	first
0	1	Alex	Anderson	4	Billy
1	2	Amy	Ackerman	5	Brian
2	3	Allen	Ali	6	Bran

Discussion

Concatenating is not a word you hear much outside of computer science and programming, so if you have not heard it before, do not worry. The informal definition of *concatenate* is to glue two objects together. In the solution we glued together two small DataFrames using the `axis` parameter to indicate whether we wanted to stack the two DataFrames on top of each other or place them side by side.

3.21 Merging DataFrames

Problem

You want to merge two DataFrames.

Solution

To inner join, use `merge` with the `on` parameter to specify the column to merge on:

```
# Load library
import pandas as pd

# Create DataFrame
employee_data = {'employee_id': ['1', '2', '3', '4'],
                 'name': ['Amy Jones', 'Allen Keys', 'Alice Bees',
                 'Tim Horton']}
dataframe_employees = pd.DataFrame(employee_data, columns = ['employee_id',
                                                             'name'])

# Create DataFrame
sales_data = {'employee_id': ['3', '4', '5', '6'],
              'total_sales': [23456, 2512, 2345, 1455]}
dataframe_sales = pd.DataFrame(sales_data, columns = ['employee_id',
                                                      'total_sales'])

# Merge DataFrames
pd.merge(dataframe_employees, dataframe_sales, on='employee_id')
```

	employee_id	name	total_sales
0	3	Alice Bees	23456
1	4	Tim Horton	2512

`merge` defaults to inner joins. If we want to do an outer join, we can specify that with the `how` parameter:

```
# Merge DataFrames
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='outer')
```

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0

4	5	NaN	2345.0
5	6	NaN	1455.0

The same parameter can be used to specify left and right joins:

```
# Merge DataFrames
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='left')
```

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0

We can also specify the column name in each DataFrame to merge on:

```
# Merge DataFrames
pd.merge(dataframe_employees,
         dataframe_sales,
         left_on='employee_id',
         right_on='employee_id')
```

	employee_id	name	total_sales
0	3	Alice Bees	23456
1	4	Tim Horton	2512

If, instead of merging on two columns, we want to merge on the indexes of each DataFrame, we can replace the `left_on` and `right_on` parameters with `left_index=True` and `right_index=True`.

Discussion

The data we need to use is often complex; it doesn't always come in one piece. Instead, in the real world, we're usually faced with disparate datasets from multiple database queries or files. To get all that data into one place, we can load each data query or data file into pandas as individual DataFrames and then merge them into a single DataFrame.

This process might be familiar to anyone who has used SQL, a popular language for doing merging operations (called *joins*). While the exact parameters used by pandas will be different, they follow the same general patterns used by other software languages and tools.

There are three aspects to specify with any `merge` operation. First, we have to specify the two DataFrames we want to merge. In the solution, we named them `dataframe_employees` and `dataframe_sales`. Second, we have to specify the name(s) of the columns to merge on—that is, the columns whose values are shared between the two DataFrames. For example, in our solution both DataFrames have a column named `employee_id`. To merge the two DataFrames we will match the values in each DataFrame's `employee_id` column. If these two columns use the same name, we can use the `on` parameter. However, if they have different names, we can use `left_on` and `right_on`.

What is the left and right DataFrame? The left DataFrame is the first one we specified in `merge`, and the right DataFrame is the second one. This language comes up again in the next sets of parameters we will need.

The last aspect, and most difficult for some people to grasp, is the type of merge operation we want to conduct. This is specified by the `how` parameter. `merge` supports the four main types of joins:

Inner

> Return only the rows that match in both DataFrames (e.g., return any row with an `employee_id` value appearing in both `dataframe_employees` and `dataframe_sales`).

Outer

> Return all rows in both DataFrames. If a row exists in one DataFrame but not in the other DataFrame, fill NaN values for the missing values (e.g., return all rows in both `dataframe_employee` and `dataframe_sales`).

Left

> Return all rows from the left DataFrame but only rows from the right DataFrame that match with the left DataFrame. Fill `NaN` values for the missing values (e.g., return all rows from `dataframe_employees` but only rows from `dataframe_sales` that have a value for `employee_id` that appears in `dataframe_employees`).

Right

> Return all rows from the right DataFrame but only rows from the left DataFrame that match with the right DataFrame. Fill `NaN` values for the missing values (e.g., return all rows from `dataframe_sales` but only rows from `dataframe_employees` that have a value for `employee_id` that appears in `dataframe_sales`).

If you did not understand all of that, I encourage you to play around with the `how` parameter in your code and see how it affects what `merge` returns.

See Also

- A Visual Explanation of SQL Joins
- pandas documentation: Merge, join, concatenate and compare