Pandas 1.x Cookbook

Practical recipes for scientific computing, time series analysis,
and exploratory data analysis using Python

Stephen CUI

October 25, 2022

Contents

Chapter 1

Essential DataFrame Operations

This chapter covers many fundamental operations of the DataFrame. Many of the recipes will be similar to those in Chapter **?? ??**, Pandas Foundations, which primarily covered operations on a Series.

1.1 Introduction

It is often necessary to focus on a subset of the current working dataset, which is accomplished by selecting multiple columns.

Recipes 1.1.1

```
import numpy as np
import pandas as pd
```

1.2 Selecting multiple DataFrame columns

1.2.1 How to do it...

1. Read in the movie dataset, and pass in a list of the desired columns to the indexing operator:

   Recipes 1.2.1

   ```
   movies = pd.read_csv('movie.csv')
   movie_actor_director = movies[
       [
           'actor_1_name',
           'actor_2_name',
           'actor_3_name',
           'director_name'
       ]
   ]
   movie_actor_director.head(3)
   ```

2. Using the index operation can return either a Series or a DataFrame. If we pass in a list with a single item, we will get back a DataFrame. If we pass in just a string with the column name, we will get a Series back:

Recipes 1.2.2

```
type(movies[['director_name']])
type(movies['director_name'])
```

We can also use .loc to pull out a column by name. Because this index operation requires that we pass in a row selector first, we will use a colon (:) to indicate a slice that selects all of the rows. This can also return either a DataFrame or a Series:

Recipes 1.2.3

```
type(movies.loc[:, ['director_name']])
type(movies.loc[:, 'director_name'])
```

1.2.2 How it works

The DataFrame index operator is very flexible and capable of accepting a number of different objects. If a string is passed, it will return a single-dimensional Series. If a list is passed to the indexing operator, it returns a DataFrame of all the columns in the list in the specified order.

1.2.3 There's more

Passing a long list inside the indexing operator might cause readability issues. To help with this, you may save all your column names to a list variable first.

Recipes 1.2.4

```
cols = ['actor_1_name', 'actor_2_name', 'actor_3_name', 'director_name']
movie_actor_director = movies[cols]
```

One of the most common exceptions raised when working with pandas is KeyError. This error is mainly due to mistyping of a column or index name. This same error is raised whenever a multiple column selection is attempted without the use of a list:

Recipes 1.2.5

```
movies[
    'actor_1_name',
    'actor_2_name',
    'actor_3_name',
    'director_name',
]
```

1.3 Selecting columns with methods

Although column selection is usually done with the indexing operator, there are some DataFrame methods that facilitate their selection in an alternative manner. The `.select_dtypes` and `.filter` methods are two useful methods to do this. If you want to select by type, you need to be familiar with pandas data types(**??**).

1.3.1 How to do it...

1. Use the `.get_dtype_counts` method to output the number of columns with each specific data type:

Recipes 1.3.1

```
movies = pd.read_csv('movie.csv')
def shorten(col):
    return (
        str(col)
        .replace('facebook_likes', 'fb')
        .replace('_for_reviews', '')
    )
movies = movies.rename(columns=shorten)
movies.dtypes.value_counts()
```

2. Use the `.select_dtypes` method to select only the integer columns:

Recipes 1.3.2

```
movies.select_dtypes(include='int').head(3)
```

3. If you would like to select all the numeric columns, you may pass the string `number` to the `include` parameter:

Recipes 1.3.3

```
movies.select_dtypes(include='number').head()
```

4. If we wanted integer and string columns we could do the following:

Recipes 1.3.4

```
movies.select_dtypes(include=['int', 'object']).head()
```

5. To exclude only floating-point columns, do the following:

Recipes 1.3.5

```
movies.select_dtypes(exclude='float').head()
```

6. An alternative method to select columns is with the `.filter` method. This method is flexible and searches column names (or index labels) based on which parameter is used. Here, we use the `like` parameter to search for all the Facebook columns or the names that contain the exact string, fb. The `like` parameter is checking for substrings in column names:

Recipes 1.3.6

```
movies.filter(like='fb').head()
```

7. The `.filter` method has more tricks (or parameters) up its sleeve. If you use the `items` parameters, you can pass in a list of column names:

Recipes 1.3.7

```
cols = [
    'actor_1_name',
    'actor_2_name',
    'actor_3_name',
    'director_name',
]
movies.filter(items=cols).head()
```

8. The `.filter` method allows columns to be searched with *regular expressions* using the *regex* parameter. Here, we search for all columns that have a digit somewhere in their name:

Recipes 1.3.8

```
movies.filter(regex=r'\d').head()
```

1.3.2 How it works...

Step 1 lists the frequencies of all the different data types. Alternatively, you may use the .dtypes attribute to get the exact data type for each column. The .select_dtypes method accepts either a list or single data type in its include or exclude parameters and returns a DataFrame with columns of just those given data types (or not those types if excluding columns). The list values may be either the string name of the data type or the actual Python object.

The .filter method selects columns by only inspecting the column names and not the actual data values. It has three mutually exclusive parameters: items, like, and regex, only one of which can be used at a time.

The like parameter takes a string and attempts to find all the column names that contain that exact string somewhere in the name. To gain more flexibility, you may use the regex parameter instead to select column names through a regular expression. This particular regular expression, r'\d', represents all digits from zero to nine and matches any string with at least a single digit in it.

The filter method comes with another parameter, items, which takes a list of exact column names. This is nearly an exact duplication of the index operation, except that a KeyError will not be raised if one of the strings does not match a column name. For instance, `movies.filter(items=['actor_1_name', 'asdf'])` runs without error and returns a single column DataFrame.

1.3.3 There's more...

One confusing aspect of .select_dtypes is its flexibility to take both strings and Python objects. The following list should clarify all the possible ways to select the many different column data types. There is no standard or preferred method of referring to data types in pandas, so it's good to be aware of both ways:

- `np.number`, `'number'` – Selects both integers and floats regardless of size
- `np.float64`, `np.float_`, `float`, `'float64'`, `'float_'`, `'float'` – Selects only 64-bit floats
- `np.float16`, `np.float32`, `np.float128`, `'float16'`, `'float32'`, `'float128'` – Respectively selects exactly 16, 32, and 128-bit floats
- np.floating, 'floating' – Selects all floats regardless of size
- np.int0, np.int64, np.int_, int, 'int0', 'int64', 'int_', 'int' – Selects only 64-bit integers
- np.int8, np.int16, np.int32, 'int8', 'int16', 'int32' – Respectively selects exactly 8, 16, and 32-bit integers
- np.integer, 'integer' – Selects all integers regardless of size
- 'Int64' – Selects nullable integer; no NumPy equivalent

- np.object, 'object', 'O' – Select all object data types
- np.datetime64, 'datetime64', 'datetime' – All datetimes are 64 bits
- np.timedelta64, 'timedelta64', 'timedelta' – All timedeltas are 64 bits
- pd.Categorical, 'category' – Unique to pandas; no NumPy equivalent

Because all integers and floats default to 64 bits, you may select them by using the string 'int' or 'float' as you can see from the preceding bullet list. If you want to select all integers and floats regardless of their specific size, use the string 'number'.

1.4 Ordering column names

There are no standardized set of rules that dictate how columns should be organized within a dataset. However, it is good practice to develop a set of guidelines that you consistently follow. This is especially true if you work with a group of analysts who share lots of datasets. The following is a guideline to order columns:

- Classify each column as either categorical or continuous
- Group common columns within the categorical and continuous columns
- Place the most important groups of columns first with categorical columns before continuous ones

1.4.1 How to do it...

Recipes 1.4.1

```
movies = pd.read_csv('movie.csv')
def shorten(col):
    return col.replace('facebook_likes', 'fb').replace('_for_reviews', '')
movies = movies.rename(columns=shorten)
cat_core = [
    'movie_title',
    'title_year',
    'content_rating',
    'genres'
]
movies.columns
```

1. The columns don't appear to have any logical ordering to them. Organize the names sensibly into lists so that the guideline from the previous section is followed:

   Recipes 1.4.2

   ```
   cat_people = ['director_name', 'actor_1_name', 'actor_2_name', 'actor_3_name']
   cat_other = ['color', 'country', 'language', 'plot_keywords', 'movies_imdb_link']
   cont_fb = ['director_fb', 'actor_1_fb', 'actor_2_fb',
              'actor_3_fb', 'cast_total_fb','movier_fb']
   cont_finance = ['budget', 'gross']
   cont_num_reviews = ['num_voted_users', 'num_user', 'num_critic']
   cont_other = ['imdb_score', 'duration', 'aspect_ratio', 'facenumber_in_poster']
   ```

2. Concatenate all the lists together to get the final column order. Also, ensure that this list contains all the columns from the original:

Recipes 1.4.3

```
new_col_order = (
    cat_core
    + cat_people
    + cat_other
    + cont_fb
    + cont_finance
    + cont_num_reviews
    + cont_other
)
set(movies.columns) == set(new_col_order)
```

3. Pass the list with the new column order to the indexing operator of the DataFrame to reorder the columns:

Recipes 1.4.4

```
movies[new_col_order].head(3)
```

1.4.2 There's more...

There are alternative guidelines for ordering columns besides the suggestion mentioned earlier. Hadley Wickham's seminal paper on Tidy Data suggests placing the fixed variables first, followed by measured variables. As this data does not come from a controlled experiment, there is some flexibility in determining which variables are fixed and which ones are measured.

1.5 Summarizing a DataFrame

In the **??** recipe in Chapter **??**, **??**, a variety of methods operated on a single column or Series of data. Many of these were aggregation or reducing methods that returned a single scalar value. When these same methods are called from a DataFrame, they perform that operation for each column at once and reduce the results for each column in the DataFrame. They return a Series with the column names in the index and the summary for each column as the value.

1.5.1 How to do it...

1. Read in the movie dataset, and examine the basic descriptive properties, .shape, .size, and .ndim, along with running the len function:

Recipes 1.5.1

```
movies = pd.read_csv('movie.csv')
movies.shape, movies.size, movies.ndim, len(movies)
```

2. The .count method shows the number of non-missing values for each column. It is an aggregation method as it summarizes every column in a single value. The output is a Series that has the original column names as its index:

Recipes 1.5.2

```
movies.count()
```

3. The other methods that compute summary statistics, `.min`, `.max`, `.mean`, `.median`, and `.std`, return Series that have the column names of the numeric columns in the index and their aggregations as the values:

Recipes 1.5.3

```
movies.min(numeric_only=True)
movies.max(numeric_only=True)
movies.mean(numeric_only=True)
movies.median(numeric_only=True)
movies.std(numeric_only=True)
```

4. The `.describe` method is very powerful and calculates all the descriptive statistics and quartiles at once. The end result is a DataFrame with the descriptive statistics names as its index. I like to transpose the results using `.T` as I can usually fit more information on the screen that way:

Recipes 1.5.4

```
movies.describe().T
```

5. It is possible to specify exact quantiles in the .describe method using the percentiles parameter:

Recipes 1.5.5

```
movies.describe(percentiles=[.025, .33, .99, .995]).T
```

1.5.2 How it works...

Step 1 gives basic information on the size of the dataset. The `.shape` attribute returns a tuple with the number of rows and columns. The `.size` attribute returns the total number of elements in the DataFrame, which is just the product of the number of rows and columns. The `.ndim` attribute returns the number of dimensions, which is two for all DataFrames. When a DataFrame is passed to the built-in len function, it returns the number of rows.

The `.describe` method displays the summary statistics of the numeric columns. You can expand its summary to include more quantiles by passing a list of numbers between 0 and 1 to the percentiles parameter. See the **??** recipe for more on the `.describe` method.

1.5.3 There's more...

To see how the `.skipna` parameter affects the outcome, we can set its value to False and rerun step 3 from the preceding recipe. Only numeric columns without missing values will calculate a result:

Recipes 1.5.6

```
movies.min(skipna=False, numeric_only=True)
```

1.6 Chaining DataFrame methods

The **??** recipe in Chapter **??**, **??**, showcased several examples of chaining Series methods together. All the method chains in this chapter will begin from a DataFrame. One of the keys to method chaining is to know the exact object being returned during each step of the chain. In pandas, this will nearly always be a DataFrame, Series, or scalar value.

1.6.1 How to do it...

1. We will use the `.isnull` method to get a count of the missing values. This method will change every value to a Boolean, indicating whether it is missing:

Recipes 1.6.1

```
movies = pd.read_csv('movie.csv')
def shorten(col):
    return col.replace('facebook_likes', 'fb').replace(
        '_for_reviews', ''
    )
movies = movies.rename(columns=shorten)
movies.isnull().head(3)
```

2. We will chain the .sum method that interprets True and False as 1 and 0, respectively. Because this is a reduction method, it aggregates the results into a Series:

Recipes 1.6.2

```
movies.isnull().sum().head(3)
```

3. We can go one step further and take the sum of this Series and return the count of the total number of missing values in the entire DataFrame as a scalar value:

Recipes 1.6.3

```
movies.isnull().sum().sum()
```

4. A way to determine whether there are any missing values in the DataFrame is to use the .any method twice in succession:

Recipes 1.6.4

```
movies.isnull().any().any()
```

1.6.2 There's more...

Most of the columns in the movie dataset with the object data type contain missing values. By default, aggregation methods (.min, .max, and .sum), do not return anything for object columns. To force pandas to return something for each column, we must fill in the missing values. Here, we choose an empty string:

Recipes 1.6.5

```
movies.select_dtypes(['object']).fillna('').max()
```

For purposes of readability, method chains are often written as one method call per line surrounded by parentheses. This makes it easier to read and insert comments on what is returned at each step of the chain, or comment out lines to debug what is happening:

Recipes 1.6.6

```
(
    movies.select_dtypes(['object'])
    .fillna('')
    .max()
)
```

1.7 DataFrame operations

The Python arithmetic and comparison operators work with DataFrames, as they do with Series.

When an arithmetic or comparison operator is used with a DataFrame, each value of each column gets the operation applied to it. Typically, when an operator is used with a DataFrame, the columns are either all numeric or all object (usually strings). If the DataFrame does not contain homogeneous data, then the operation is likely to fail.

Attempting to add 5 to each value of the DataFrame raises a TypeError as integers cannot be added to strings:

Recipes 1.7.1

```
colleges = pd.read_csv('college.csv')
colleges + 5
```

To successfully use an operator with a DataFrame, first select homogeneous data. For this recipe, we will select all the columns that begin with `'UGDS_'`. These columns represent the fraction of undergraduate students by race. To get started, we import the data and use the institution name as the label for our index, and then select the columns we desire with the `.filter` method:

Recipes 1.7.2

```
colleges = pd.read_csv('college.csv', index_col='INSTNM')
colleges_ugds = colleges.filter(like='UGDS_')
colleges_ugds.head(3)
```

1.7.1 How to do it...

1. **pandas does bankers rounding, numbers that are exactly halfway between either side to the even side.**

   Recipes 1.7.3 bankers rounding

   ```
   name = 'Northwest-Shoals Community College'
   colleges_ugds.loc[name]
   colleges_ugds.loc[name].round(2)
   ```

1.7.2 How it works...

NumPy and Python 3 round numbers that are exactly halfway between either side to the even number. The bankers rounding (or ties to even http://bit.ly/2x3V5TU) technique is not usually what is formally taught in schools. It does not consistently bias numbers to the higher side (http://bit.ly/2zhsPy8).

1.7.3 There's more...

Just as with Series, DataFrames have method equivalents of the operators. You may replace the operators with their method equivalents:

Recipes 1.7.4

```
college2 = (
    colleges_ugds.add(0.00501).floordiv(0.01).div(100)
)
college2.equals(colleges_ugds_up_round)
```

1.8 Comparing missing values

pandas uses the NumPy NaN (np.nan) object to represent a missing value. This is an unusual object and has interesting mathematical properties. For instance, **it is not equal to itself**. **Even Python's None object evaluates as True when compared to itself**:

Recipes 1.8.1

```
np.nan == np.nan
None == None
```

All other comparisons against np.nan also return False, except not equal to (!=):

Recipes 1.8.2

```
np.nan > 5
5 > np.nan
np.nan != 5
```

1.8.1 Getting ready

Series and DataFrames use the equals operator, ==, to make element-by-element comparisons. The result is an object with the same dimensions. This recipe shows you how to use the equals operator, which is very different from the .equals method.

Recipes 1.8.3

```
college = pd.read_csv('college.csv', index_col='INSTNM')
college_ugds = college.filter(like='UGDS_')
```

1.8.2 How to do it...

1. To get an idea of how the equals operator works, let's compare each element to a scalar value.

2. This works as expected but becomes problematic whenever you attempt to compare DataFrames with missing values. You may be tempted to use the equals operator to compare two DataFrames with one another on an element-by-element basis. Take, for instance, college_ugds compared against itself, as follows.

3. At first glance, all the values appear to be equal, as you would expect. However, using the .all method to determine if each column contains only True values yields an unexpected result.

   Recipes 1.8.4

   ```
   college_ugds == 0

   college_self_compare = college_ugds == college_ugds
   college_self_compare.head(3)

   college_self_compare.all()
   ```

4. This happens because missing values do not compare equally with one another. If you tried to count missing values using the equal operator and summing up the Boolean columns, you would get zero for each one:

5. Instead of using == to find missing numbers, use the .isna method:

6. **The correct way to compare two entire DataFrames with one another is not with the equals operator (==) but with the .equals method. This method treats NaNs that are in the same location as equal (note that the .eq method is the equivalent of ==)**:

Recipes 1.8.5

```
(college_ugds == np.nan).sum()

college_ugds.isna().sum()

college_ugds.equals(college_ugds)

# colleges_ugds.eq(college_ugds)
```

1.8.3 There's more...

Inside the pandas.testing sub-package, a function exists that developers should use when creating unit tests. The `assert_frame_equal` function raises an AssertionError if two DataFrames are not equal. It returns None if the two DataFrames are equal:

Recipes 1.8.6

```
from pandas.testing import assert_frame_equal
assert_frame_equal(college_ugds, college_ugds) is None
```

Unit tests are a very important part of software development and ensure that the code is running correctly. pandas contains many thousands of unit tests that help ensure that it is running properly. To read more on how pandas runs its unit tests, see the Contributing to pandas section in the documentation (http://bit.ly/2vmCSU6).

1.9 Transposing the direction of a DataFrame operation

1.10 Determining college campus diversity