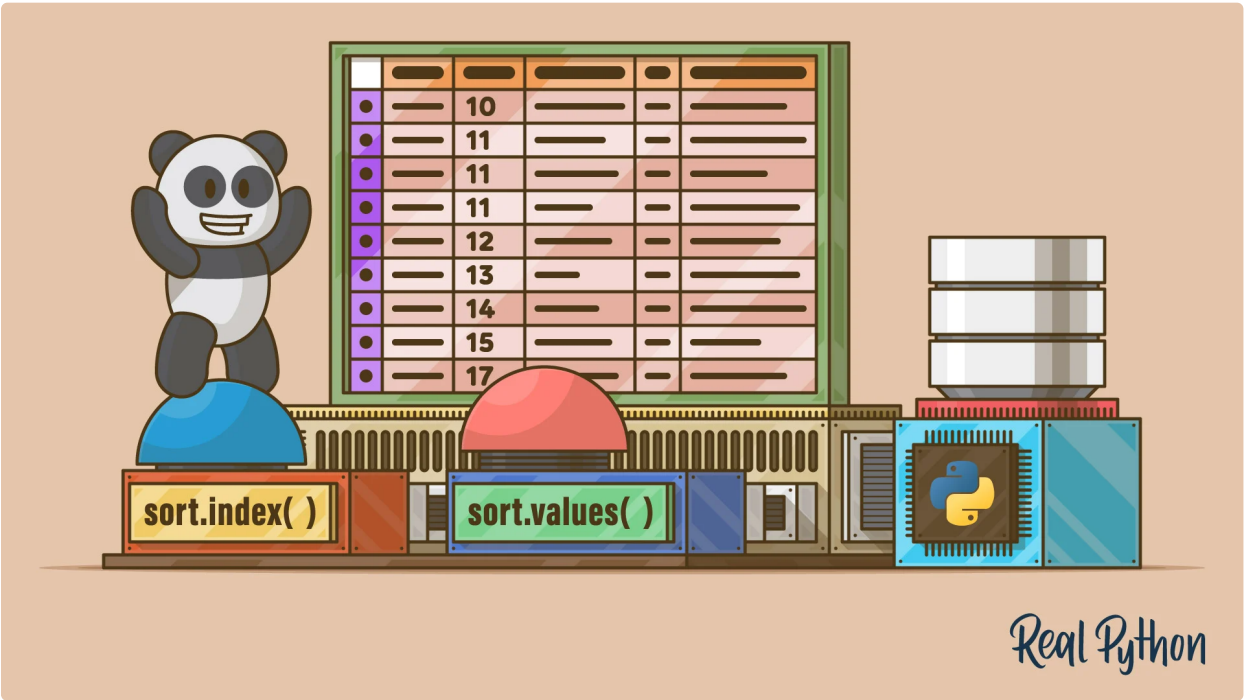

pandas Sort: Your Guide to Sorting Data in Python

by Spencer Guy 0 Comments data-science intermediate

Mark as Completed

Tweet Share Email

Table of Contents

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Sorting Data in Python With Pandas**

Learning pandas **sort methods** is a great way to start with or practice doing basic data analysis using Python. Most commonly, data analysis is done with spreadsheets, SQL, or pandas. One of the great things about using pandas is that it can handle a large amount of data and offers highly performant data manipulation capabilities.

In this tutorial, you'll learn how to use `.sort_values()` and `.sort_index()`, which will enable you to sort data efficiently in a DataFrame.

By the end of this tutorial, you'll know how to:

- Sort a **pandas DataFrame** by the values of one or more columns
- Use the `ascending` parameter to change the **sort order**
- Sort a DataFrame by its `index` using **`.sort_index()`**
- Organize **missing data** while sorting values
- Sort a DataFrame **in place** using `inplace` set to `True`

To follow along with this tutorial, you'll need a basic understanding of pandas DataFrames and some familiarity with reading in data from files.

Free Bonus: Click here to get a Python Cheat Sheet and learn the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

Getting Started With Pandas Sort Methods

As a quick reminder, a **DataFrame** is a data structure with labeled axes for both rows and columns. You can sort a DataFrame by row or column value as well as by row or column index.

Both rows and columns have **indices**, which are numerical representations of where the data is in your DataFrame. You can retrieve data from specific rows or columns using the

DataFrame's index locations. By default, index numbers start from zero. You can also manually assign your own index.

Preparing the Dataset

In this tutorial, you'll be working with fuel economy data compiled by the US Environmental Protection Agency (EPA) on vehicles made between 1984 and 2021. The EPA fuel economy dataset is great because it has many different types of information that you can sort on, from textual to numeric data types. The dataset contains eighty-three columns in total.

To follow along, you'll need to have the pandas Python library installed. The code in this tutorial was executed using pandas 1.2.0 and Python 3.9.1.

> **Note:** The whole fuel economy dataset is around 18 MB. Reading the entire dataset into memory could take a minute or two. Limiting the number of rows and columns will help performance, but it will still take a few seconds before the data is downloaded.

For analysis purposes, you'll be looking at MPG (miles per gallon) data on vehicles by make, model, year, and other vehicle attributes. You can specify which columns to read into your DataFrame. For this tutorial, you'll need only a subset of the available columns.

Here are the commands to read the relevant columns of the fuel economy dataset into a DataFrame and to display the first five rows:

DataFrame and to display the first five rows.

Python >>>

```
>>> import pandas as pd

>>> column_subset = [
...     "id",
...     "make",
...     "model",
...     "year",
...     "cylinders",
...     "fuelType",
...     "trany",
...     "mpgData",
...     "city08",
...     "highway08"
... ]

>>> df = pd.read_csv(
...     "https://www.fueleconomy.gov/feg/epadata/vehicles.csv",
...     usecols=column_subset,
...     nrows=100
... )

>>> df.head()
   city08  cylinders fuelType  ...  mpgData            trany  year
0      19          4  Regular  ...        Y     Manual 5-spd  1985
1       9         12  Regular  ...        N     Manual 5-spd  1985
2      23          4  Regular  ...        Y     Manual 5-spd  1985
3      10          8  Regular  ...        N  Automatic 3-spd  1985
4      17          4  Premium  ...        N     Manual 5-spd  1993
[5 rows x 10 columns]
```

By calling `.read_csv()` with the dataset URL, you're able to load the data into a DataFrame. Narrowing down the columns results in faster load times and lower memory use. To further limit memory consumption and to get a quick feel for the data, you can specify how many

rows to load using `nrows`.

Getting Familiar With `.sort_values()`

You use `.sort_values()` to sort values in a DataFrame along either axis (columns or rows). Typically, you want to sort the rows in a DataFrame by the values of one or more columns:

The figure above shows the results of using `.sort_values()` to sort the DataFrame's rows based on the values in the `highway08` column. This is similar to how you would sort data in a spreadsheet using a column.

Getting Familiar With `.sort_index()`

You use `.sort_index()` to sort a DataFrame by its row index or column labels. The difference from using `.sort_values()` is that you're sorting the DataFrame based on its row index or column names, not by the values in these rows or columns:

The row index of the DataFrame is outlined in blue in the figure above. An index isn't considered a column, and you typically have only a single row index. The row index can be thought of as the row numbers, which start from zero.

Sorting Your DataFrame on a Single Column

To sort the DataFrame based on the values in a single column, you'll use `.sort_values()`. By default, this will return a new DataFrame sorted in ascending order. It does not modify the original DataFrame.

Sorting by a Column in Ascending Order

To use `.sort_values()`, you pass a single argument to the method containing the name of the column you want to sort by. In this example, you sort the DataFrame by the `city02`

the column you want to sort by. In this example, you sort the DataFrame by the `city08` column, which represents city MPG for fuel-only cars:

```
Python                                                                    >>>
>>> df.sort_values("city08")
    city08  cylinders fuelType  ...  mpgData            trany  year
99       9          8  Premium  ...        N  Automatic 4-spd  1993
1        9         12  Regular  ...        N     Manual 5-spd  1985
80       9          8  Regular  ...        N  Automatic 3-spd  1985
47       9          8  Regular  ...        N  Automatic 3-spd  1985
3       10          8  Regular  ...        N  Automatic 3-spd  1985
..     ...        ...      ...  ...      ...              ...   ...
9       23          4  Regular  ...        Y  Automatic 4-spd  1993
8       23          4  Regular  ...        Y     Manual 5-spd  1993
7       23          4  Regular  ...        Y  Automatic 3-spd  1993
76      23          4  Regular  ...        Y     Manual 5-spd  1993
2       23          4  Regular  ...        Y     Manual 5-spd  1985
[100 rows x 10 columns]
```

This sorts your DataFrame using the column values from `city08`, showing the vehicles with the lowest MPG first. By default, `.sort_values()` sorts your data in **ascending order**. Although you didn't specify a name for the argument you passed to `.sort_values()`, you actually used the `by` parameter, which you'll see in the next example.

Changing the Sort Order

Another parameter of `.sort_values()` is `ascending`. By default `.sort_values()` has `ascending` set to `True`. If you want the DataFrame sorted in **descending order**, then you can pass `False` to this parameter:

```
Python                                                                >>>
>>> df.sort_values(
...     by="city08",
...     ascending=False
... )
    city08  cylinders fuelType  ...  mpgData           trany  year
9       23          4  Regular  ...        Y  Automatic 4-spd  1993
2       23          4  Regular  ...        Y     Manual 5-spd  1985
7       23          4  Regular  ...        Y  Automatic 3-spd  1993
8       23          4  Regular  ...        Y     Manual 5-spd  1993
76      23          4  Regular  ...        Y     Manual 5-spd  1993
..     ...        ...      ...  ...      ...              ...   ...
58      10          8  Regular  ...        N  Automatic 3-spd  1985
80       9          8  Regular  ...        N  Automatic 3-spd  1985
1        9         12  Regular  ...        N     Manual 5-spd  1985
47       9          8  Regular  ...        N  Automatic 3-spd  1985
99       9          8  Premium  ...        N  Automatic 4-spd  1993
[100 rows x 10 columns]
```

By passing `False` to `ascending`, you reverse the sort order. Now your DataFrame is sorted in descending order by the average MPG measured in city conditions. The vehicles with the highest MPG values are in the first rows.

Choosing a Sorting Algorithm

It's good to note that pandas allows you to choose different **sorting algorithms** to use with both `.sort_values()` and `.sort_index()`. The available algorithms are `quicksort`, `mergesort`, and `heapsort`. For more information on these different sorting algorithms, check out Sorting Algorithms in Python.

The algorithm used by default when sorting on a single column is `quicksort`. To change this to a stable sorting algorithm, use `mergesort`. You can do that with the `kind` parameter in

to a stable sorting algorithm, use `mergesort`. You can do that with the `kind` parameter in `.sort_values()` or `.sort_index()`, like this:

Python >>>

```
>>> df.sort_values(
...     by="city08",
...     ascending=False,
...     kind="mergesort"
... )
    city08  cylinders fuelType  ...  mpgData            trany  year
2       23          4  Regular  ...        Y     Manual 5-spd  1985
7       23          4  Regular  ...        Y  Automatic 3-spd  1993
8       23          4  Regular  ...        Y     Manual 5-spd  1993
9       23          4  Regular  ...        Y  Automatic 4-spd  1993
10      23          4  Regular  ...        Y     Manual 5-spd  1993
..     ...        ...      ...  ...      ...              ...   ...
69      10          8  Regular  ...        N  Automatic 3-spd  1985
1        9         12  Regular  ...        N     Manual 5-spd  1985
47       9          8  Regular  ...        N  Automatic 3-spd  1985
80       9          8  Regular  ...        N  Automatic 3-spd  1985
99       9          8  Premium  ...        N  Automatic 4-spd  1993
[100 rows x 10 columns]
```

Using `kind`, you set the sorting algorithm to `mergesort`. The previous output used the default `quicksort` algorithm. Looking at the highlighted indices, you can see the rows are in a different order. This is because `quicksort` is not a stable sorting algorithm, but `mergesort` is.

> **Note:** In pandas, `kind` is ignored when you sort on more than one column or label.

When you're sorting multiple records that have the same key, a **stable sorting algorithm** will maintain the original order of those records after sorting. For that reason, using a stable sorting algorithm is necessary if you plan to perform multiple sorts.

Sorting Your DataFrame on Multiple Columns

In data analysis, it's common to want to sort your data based on the values of multiple columns. Imagine you have a dataset with people's first and last names. It would make sense to sort by last name and then first name, so that people with the same last name are arranged alphabetically according to their first names.

In the first example, you sorted your DataFrame on a single column named `city08`. From an analysis standpoint, the MPG in city conditions is an important factor that could determine a car's desirability. In addition to the MPG in city conditions, you may also want to look at MPG for highway conditions. To sort by two keys, you can pass a list of column names to `by`:

Python >>>

```
>>> df.sort_values(
...     by=["city08", "highway08"]
... )[["city08", "highway08"]]
    city08  highway08
80       9         10
47       9         11
99       9         13
1        9         14
58      10         11
..     ...        ...
9       23         30
10      23         30
8       23         31
76      23         31
2       23         33
[100 rows x 2 columns]
```

By specifying a list of the column names `city08` and `highway08`, you sort the DataFrame on two columns using `.sort_values()`. The next example will explain how to specify the sort order and why it's important to pay attention to the list of column names you use.

Sorting by Multiple Columns in Ascending Order

To sort the DataFrame on multiple columns, you must provide a list of column names. For example, to sort by `make` and `model`, you should create the following list and then pass it to `.sort_values()`:

Python >>>

```
>>> df.sort_values(
...     by=["make", "model"]
... )[["make", "model"]]
          make                model
0   Alfa Romeo  Spider Veloce 2000
18        Audi                 100
19        Audi                 100
20         BMW                740i
21         BMW               740il
..         ...                 ...
12  Volkswagen      Golf III / GTI
13  Volkswagen           Jetta III
15  Volkswagen           Jetta III
16       Volvo                 240
17       Volvo                 240
[100 rows x 2 columns]
```

Now your DataFrame is sorted in ascending order by `make`. If there are two or more identical makes, then it's sorted by `model`. The order in which the column names are specified in your list corresponds to how your DataFrame will be sorted.

Changing the Column Sort Order

Since you're sorting using multiple columns, you can specify the order by which your columns get sorted. If you want to change the logical sort order from the previous example, then you can change the order of the column names in the list you pass to the `by` parameter:

```
Python >>>
>>> df.sort_values(
...     by=["model", "make"]
... )[["make", "model"]]
              make        model
18           Audi          100
19           Audi          100
16          Volvo          240
17          Volvo          240
75          Mazda          626
..            ...          ...
62           Ford  Thunderbird
63           Ford  Thunderbird
88     Oldsmobile     Toronado
42  CX Automotive        XM v6
43  CX Automotive       XM v6a
[100 rows x 2 columns]
```

Your DataFrame is now sorted by the `model` column in ascending order, then sorted by `make` if there are two or more of the same model. You can see that changing the order of columns also changes the order in which the values get sorted.

Sorting by Multiple Columns in Descending Order

Up to this point, you've sorted only in ascending order on multiple columns. In the next example, you'll sort in descending order based on the `make` and `model` columns. To sort in descending order, set `ascending` to `False`:

```
Python >>>
>>> df.sort_values(
...     by=["make", "model"],
...     ascending=False
... )[["make", "model"]]
          make               model
16      Volvo                 240
17      Volvo                 240
13  Volkswagen           Jetta III
15  Volkswagen           Jetta III
11  Volkswagen      Golf III / GTI
..         ...                 ...
21         BMW               740il
20         BMW                740i
18        Audi                 100
19        Audi                 100
0   Alfa Romeo  Spider Veloce 2000
[100 rows x 2 columns]
```

The values in the `make` column are in reverse alphabetical order, and the values in the `model` column are in descending order for any cars with the same `make`. With textual data, the sort is **case sensitive**, meaning capitalized text will appear first in ascending order and last in descending order.

Sorting by Multiple Columns With Different Sort Orders

You might be wondering if it's possible to sort using multiple columns and to have those columns use different `ascending` arguments. With pandas, you can do this with a single method call. If you want to sort some columns in ascending order and some columns in descending order, then you can pass a list of Booleans to `ascending`.

In this example, you sort your DataFrame by the `make`, `model`, and `city08` columns, with the

first two columns sorted in ascending order and `city08` sorted in descending order. To do so, you pass a list of column names to `by` and a list of Booleans to `ascending`:

```
Python                                                    >>>
>>> df.sort_values(
...     by=["make", "model", "city08"],
...     ascending=[True, True, False]
... )[["make", "model", "city08"]]
          make               model  city08
0   Alfa Romeo  Spider Veloce 2000      19
18        Audi                 100      17
19        Audi                 100      17
20         BMW                740i      14
21         BMW               740il      14
..         ...                 ...     ...
11  Volkswagen      Golf III / GTI      18
15  Volkswagen           Jetta III      20
13  Volkswagen           Jetta III      18
17       Volvo                 240      19
16       Volvo                 240      18
[100 rows x 3 columns]
```

Now your DataFrame is sorted by `make` and `model` in ascending order, but with the `city08` column in descending order. This is helpful because it groups the cars in a categorical order and shows the highest MPG cars first.

Sorting Your DataFrame on Its Index

Before sorting on the index, it's a good idea to know what an index represents. A DataFrame has an **`.index`** property, which by default is a numerical representation of its rows' locations. You can think of the index as the row numbers. It helps in quick row lookup and identification.

Sorting by Index in Ascending Order

Sorting by Index in Ascending Order

You can sort a DataFrame based on its row index with `.sort_index()`. Sorting by column values like you did in the previous examples reorders the rows in your DataFrame, so the index becomes disorganized. This can also happen when you filter a DataFrame or when you drop or add rows.

To illustrate the use of `.sort_index()`, start by creating a new sorted DataFrame using `.sort_values()`:

Python >>>

```
>>> sorted_df = df.sort_values(by=["make", "model"])
>>> sorted_df
    city08  cylinders fuelType  ...  mpgData            trany  year
0       19          4  Regular  ...        Y     Manual 5-spd  1985
18      17          6  Premium  ...        Y  Automatic 4-spd  1993
19      17          6  Premium  ...        N     Manual 5-spd  1993
20      14          8  Premium  ...        N  Automatic 5-spd  1993
21      14          8  Premium  ...        N  Automatic 5-spd  1993
..     ...        ...      ...  ...      ...              ...   ...
12      21          4  Regular  ...        Y     Manual 5-spd  1993
13      18          4  Regular  ...        N  Automatic 4-spd  1993
15      20          4  Regular  ...        N     Manual 5-spd  1993
16      18          4  Regular  ...        Y  Automatic 4-spd  1993
17      19          4  Regular  ...        Y     Manual 5-spd  1993
[100 rows x 10 columns]
```

You've created a DataFrame that's sorted using multiple values. Notice how the row index is in no particular order. To get your new DataFrame back to the original order, you can use `.sort_index()`:

Python >>>

```
>>> sorted_df.sort_index()
    city08  cylinders fuelType  ...  mpgData            trany  year
0       19          4  Regular  ...        Y     Manual 5-spd  1985
1        9         12  Regular  ...        N     Manual 5-spd  1985
2       23          4  Regular  ...        Y     Manual 5-spd  1985
3       10          8  Regular  ...        N  Automatic 3-spd  1985
4       17          4  Premium  ...        N     Manual 5-spd  1993
..     ...        ...      ...  ...      ...              ...   ...
95      17          6  Regular  ...        Y  Automatic 3-spd  1993
96      17          6  Regular  ...        N  Automatic 4-spd  1993
97      15          6  Regular  ...        N  Automatic 4-spd  1993
98      15          6  Regular  ...        N     Manual 5-spd  1993
99       9          8  Premium  ...        N  Automatic 4-spd  1993
[100 rows x 10 columns]
```

Now the index is in ascending order. Just like `.sort_values()`, the default argument for `ascending` in `.sort_index()` is `True`, and you can change to descending order by passing `False`. Sorting on the index has no impact on the data itself as the values are unchanged.

This is particularly useful when you've assigned a custom index with **`.set_index()`**. If you want to set a custom index using the `make` and `model` columns, then you can pass a list to `.set_index()`:

Python >>>

```
>>> assigned_index_df = df.set_index(
...     ["make", "model"]
... )
>>> assigned_index_df
                                 city08  cylinders  ...            trany  yea
make        model                                   ...
Alfa Romeo  Spider Veloce 2000       19          4  ...     Manual 5-spd  198
Ferrari     Testarossa               9         12  ...     Manual 5-spd  198
Dodge       Charger                 23          4  ...     Manual 5-spd  198
            B150/B250 Wagon 2WD     10          8  ...  Automatic 3-spd  198
Subaru      Legacy AWD Turbo        17          4  ...     Manual 5-spd  199
                                   ...        ...  ...              ...   ...
Pontiac     Grand Prix              17          6  ...  Automatic 3-spd  199
            Grand Prix              17          6  ...  Automatic 4-spd  199
            Grand Prix              15          6  ...  Automatic 4-spd  199
            Grand Prix              15          6  ...     Manual 5-spd  199
Rolls-Royce Brooklands/Brklnds L     9          8  ...  Automatic 4-spd  199
[100 rows x 8 columns]
```

Using this method, you replace the default integer-based row index with two axis labels. This is considered a `MultiIndex` or a **hierarchical index**. Your DataFrame is now indexed by more than one key, which you can sort on with `.sort_index()`:

Python >>>

```
>>> assigned_index_df.sort_index()
                              city08  cylinders  ...            trany  year
make       model                                 ...
Alfa Romeo Spider Veloce 2000     19          4  ...     Manual 5-spd  1985
Audi       100                    17          6  ...  Automatic 4-spd  1993
           100                    17          6  ...     Manual 5-spd  1993
BMW        740i                   14          8  ...  Automatic 5-spd  1993
           740il                  14          8  ...  Automatic 5-spd  1993
                              ...        ...  ...              ...   ...
Volkswagen Golf III / GTI         21          4  ...     Manual 5-spd  1993
           Jetta III              18          4  ...  Automatic 4-spd  1993
           Jetta III              20          4  ...     Manual 5-spd  1993
Volvo      240                    18          4  ...  Automatic 4-spd  1993
           240                    19          4  ...     Manual 5-spd  1993
[100 rows x 8 columns]
```

First you assign a new index to your DataFrame using the `make` and `model` columns, then you sort the index using `.sort_index()`. You can read more on using `.set_index()` in the pandas documentation.

Sorting by Index in Descending Order

For the next example, you'll sort your DataFrame by its index in descending order. Remember from sorting your DataFrame with `.sort_values()` that you can reverse the sort order by setting `ascending` to `False`. This parameter also works with `.sort_index()`,

so you can sort your DataFrame in reverse order like this:

```
Python                                                                    >>>
>>> assigned_index_df.sort_index(ascending=False)
                              city08  cylinders  ...            trany  year
make       model                                ...
Volvo      240                    18          4  ...  Automatic 4-spd  1993
           240                    19          4  ...     Manual 5-spd  1993
Volkswagen Jetta III              18          4  ...  Automatic 4-spd  1993
           Jetta III              20          4  ...     Manual 5-spd  1993
           Golf III / GTI         18          4  ...  Automatic 4-spd  1993
                                 ...        ...  ...              ...   ...
BMW        740il                  14          8  ...  Automatic 5-spd  1993
           740i                   14          8  ...  Automatic 5-spd  1993
Audi       100                    17          6  ...  Automatic 4-spd  1993
           100                    17          6  ...     Manual 5-spd  1993
Alfa Romeo Spider Veloce 2000     19          4  ...     Manual 5-spd  1985
[100 rows x 8 columns]
```

Now your DataFrame is sorted by its index in descending order. One difference between using `.sort_index()` and `.sort_values()` is that `.sort_index()` has no `by` parameter since it sorts a DataFrame on the row index by default.

Exploring Advanced Index-Sorting Concepts

There are many cases in data analysis where you want to sort on a hierarchical index. You've already seen how you can use `make` and `model` in a `MultiIndex`. For this dataset, you could also use the `id` column as an index.

Setting the `id` column as the index could be helpful in linking related datasets. For example, the EPA's emissions dataset also uses `id` to represent vehicle record IDs. This links the emissions data to the fuel economy data. Sorting the index of both datasets in DataFrames could speed up using other methods such as `.merge()`. To learn more about combining data

in pandas, check out Combining Data in Pandas With merge(), .join(), and concat().

Sorting the Columns of Your DataFrame

You can also use the column labels of your DataFrame to sort row values. Using `.sort_index()` with the optional parameter `axis` set to `1` will sort the DataFrame by the column labels. The sorting algorithm is applied to the **axis labels** instead of to the actual data. This can be helpful for visual inspection of the DataFrame.

Working With the DataFrame `axis`

When you use `.sort_index()` without passing any explicit arguments, it uses `axis=0` as a default argument. The **axis** of a DataFrame refers to either the index (`axis=0`) or the columns (`axis=1`). You can use both axes for indexing and selecting data in a DataFrame as well as for sorting the data.

Using Column Labels to Sort

You can also use the column labels of a DataFrame as the sorting key for `.sort_index()`. Setting `axis` to `1` sorts the columns of your DataFrame based on the column labels:

Python >>>

```
>>> df.sort_index(axis=1)
    city08  cylinders fuelType  ...  mpgData            trany  year
0       19          4  Regular  ...        Y     Manual 5-spd  1985
1        9         12  Regular  ...        N     Manual 5-spd  1985
2       23          4  Regular  ...        Y     Manual 5-spd  1985
3       10          8  Regular  ...        N  Automatic 3-spd  1985
4       17          4  Premium  ...        N     Manual 5-spd  1993
..     ...        ...      ...  ...      ...              ...   ...
95      17          6  Regular  ...        Y  Automatic 3-spd  1993
96      17          6  Regular  ...        N  Automatic 4-spd  1993
97      15          6  Regular  ...        N  Automatic 4-spd  1993
98      15          6  Regular  ...        N     Manual 5-spd  1993
99       9          8  Premium  ...        N  Automatic 4-spd  1993
[100 rows x 10 columns]
```

The columns of your DataFrame are sorted from left to right in ascending alphabetical order. If you want to sort the columns in descending order, then you can use `ascending=False`:

Python >>>

```
>>> df.sort_index(axis=1, ascending=False)
    year            trany mpgData  ... fuelType cylinders  city08
0   1985     Manual 5-spd       Y  ...  Regular         4      19
1   1985     Manual 5-spd       N  ...  Regular        12       9
2   1985     Manual 5-spd       Y  ...  Regular         4      23
3   1985  Automatic 3-spd       N  ...  Regular         8      10
4   1993     Manual 5-spd       N  ...  Premium         4      17
..   ...              ...     ...  ...      ...       ...     ...
95  1993  Automatic 3-spd       Y  ...  Regular         6      17
96  1993  Automatic 4-spd       N  ...  Regular         6      17
97  1993  Automatic 4-spd       N  ...  Regular         6      15
98  1993     Manual 5-spd       N  ...  Regular         6      15
```

```
99  1993  Automatic 4-spd       N  ...  Premium         8       9
[100 rows x 10 columns]
```

Using `axis=1` in `.sort_index()`, you sorted the columns of your DataFrame in both ascending and descending order. This could be more useful in other datasets, such as one in which the column labels correspond to months of the year. In that case, it would make sense to arrange your data in ascending or descending order by month.

Working With Missing Data When Sorting in Pandas

Oftentimes real-world data has many imperfections. While pandas has several methods you can use to clean your data before sorting, sometimes it's nice to see which data is missing while you're sorting. You can do that with the **`na_position`** parameter.

The subset of the fuel economy data used for this tutorial doesn't have missing values. To illustrate the use of `na_position`, first you'll need to create some missing data. The following piece of code creates a new column based on the existing `mpgData` column, mapping `True` where `mpgData` equals `Y` and `NaN` where it doesn't:

Python >>>

```
>>> df["mpgData_"] = df["mpgData"].map({"Y": True})
>>> df
    city08  cylinders fuelType  ...            trany  year mpgData_
0       19          4  Regular  ...     Manual 5-spd  1985     True
1        9         12  Regular  ...     Manual 5-spd  1985      NaN
2       23          4  Regular  ...     Manual 5-spd  1985     True
3       10          8  Regular  ...  Automatic 3-spd  1985      NaN
4       17          4  Premium  ...     Manual 5-spd  1993      NaN
..     ...        ...      ...  ...              ...   ...      ...
95      17          6  Regular  ...  Automatic 3-spd  1993     True
96      17          6  Regular  ...  Automatic 4-spd  1993      NaN
97      15          6  Regular  ...  Automatic 4-spd  1993      NaN
98      15          6  Regular  ...     Manual 5-spd  1993      NaN
```

```
99       9          8  Premium  ...  Automatic 4-spd  1993      NaN
[100 rows x 11 columns]
```

Now you have a new column named `mpgData_` that contains both `True` and `NaN` values. You'll use this column to see what effect `na_position` has when you use the two sort methods. To find out more about using `.map()`, you can read Pandas Project: Make a Gradebook With Python & Pandas.

Understanding the `na_position` Parameter in `.sort_values()`

`.sort_values()` accepts a parameter named **`na_position`**, which helps to organize missing data in the column you're sorting on. If you sort on a column with missing data, then the rows with the missing values will appear at the end of your DataFrame. This happens regardless of whether you're sorting in ascending or descending order.

Here's what your DataFrame looks like when you sort on the column with missing data:

Python >>>

```
>>> df.sort_values(by="mpgData_")
    city08  cylinders fuelType  ...            trany  year mpgData_
0       19          4  Regular  ...     Manual 5-spd  1985     True
55      18          6  Regular  ...  Automatic 4-spd  1993     True
56      18          6  Regular  ...  Automatic 4-spd  1993     True
57      16          6  Premium  ...     Manual 5-spd  1993     True
59      17          6  Regular  ...  Automatic 4-spd  1993     True
..     ...        ...      ...  ...              ...   ...      ...
94      18          6  Regular  ...  Automatic 4-spd  1993      NaN
96      17          6  Regular  ...  Automatic 4-spd  1993      NaN
97      15          6  Regular  ...  Automatic 4-spd  1993      NaN
98      15          6  Regular  ...     Manual 5-spd  1993      NaN
99       9          8  Premium  ...  Automatic 4-spd  1993      NaN
[100 rows x 11 columns]
```

To change that behavior and have the missing data appear first in your DataFrame, you can set `na_position` to `first`. The `na_position` parameter only accepts the values `last`, which is the default, and `first`. Here's how to use `na_postion` in `.sort_values()`:

Python >>>

```
>>> df.sort_values(
...     by="mpgData_",
...     na_position="first"
... )
    city08  cylinders fuelType  ...            trany  year mpgData_
1        9         12  Regular  ...     Manual 5-spd  1985      NaN
3       10          8  Regular  ...  Automatic 3-spd  1985      NaN
4       17          4  Premium  ...     Manual 5-spd  1993      NaN
5       21          4  Regular  ...  Automatic 3-spd  1993      NaN
11      18          4  Regular  ...  Automatic 4-spd  1993      NaN
..     ...        ...      ...  ...              ...   ...      ...
32      15          8  Premium  ...  Automatic 4-spd  1993     True
33      15          8  Premium  ...  Automatic 4-spd  1993     True
37      17          6  Regular  ...  Automatic 3-spd  1993     True
85      17          6  Regular  ...  Automatic 4-spd  1993     True
95      17          6  Regular  ...  Automatic 3-spd  1993     True
[100 rows x 11 columns]
```

Now any missing data from the columns you used to sort on will be shown at the top of your DataFrame. This is most helpful when you're first starting to analyze your data and are unsure if there are missing values.

Understanding the `na_position` Parameter in `.sort_index()`

`.sort_index()` also accepts `na_position`. Your DataFrame typically won't have `NaN` values as a part of its index, so this parameter is less useful in `.sort_index()`. However, it's good to know that if your DataFrame does have `NaN` in either the row index or a column name, then you can quickly identify this using `.sort_index()` and `na_position`.

By default, this parameter is set to `last`, which places `NaN` values at the end of the sorted result. To change that behavior and have the missing data first in your DataFrame, set `na_position` to `first`.

Using Sort Methods to Modify Your DataFrame

In all the examples you've seen so far, both `.sort_values()` and `.sort_index()` have returned DataFrame objects when you called those methods. That's because sorting in pandas doesn't work in place by default. In general, this is the most common and preferred way to analyze your data with pandas since it creates a new DataFrame instead of modifying the original. This allows you to preserve the state of the data from when you read it from your file.

However, you can modify the original DataFrame directly by specifying the optional parameter **`inplace`** with the value of `True`. The majority of pandas methods include the `inplace` parameter. Below, you'll see a few examples of using `inplace=True` to sort your DataFrame in place.

Using `.sort_values()` In Place

With `inplace` set to `True`, you modify the original DataFrame, so the sort methods return `None`. Sort your DataFrame by the values of the `city08` column like the very first example, but with `inplace` set to `True`:

```
Python                                                    >>>
>>> df.sort_values("city08", inplace=True)
```

Notice how calling `.sort_values()` doesn't return a DataFrame. Here's what the original `df` looks like:

Python >>>

```
>>> df
    city08  cylinders fuelType  ...            trany  year mpgData_
99       9          8  Premium  ...  Automatic 4-spd  1993      NaN
1        9         12  Regular  ...     Manual 5-spd  1985      NaN
80       9          8  Regular  ...  Automatic 3-spd  1985      NaN
47       9          8  Regular  ...  Automatic 3-spd  1985      NaN
3       10          8  Regular  ...  Automatic 3-spd  1985      NaN
..     ...        ...      ...  ...              ...   ...      ...
9       23          4  Regular  ...  Automatic 4-spd  1993     True
8       23          4  Regular  ...     Manual 5-spd  1993     True
7       23          4  Regular  ...  Automatic 3-spd  1993     True
76      23          4  Regular  ...     Manual 5-spd  1993     True
2       23          4  Regular  ...     Manual 5-spd  1985     True
[100 rows x 11 columns]
```

In the `df` object, the values are now sorted in ascending order based on the `city08` column. Your original DataFrame has been modified, and the changes will persist. It's generally a good idea to avoid using `inplace=True` for analysis because the changes to your DataFrame can't be undone.

Using `.sort_index()` In Place

The next example illustrates that `inplace` also works with `.sort_index()`.

Since the index was created in ascending order when you read your file into a DataFrame, you can modify your `df` object again to get it back to its initial order. Use `.sort_index()` with `inplace` set to `True` to modify the DataFrame:

Python >>>

```
>>> df.sort_index(inplace=True)
>>> df
    city08  cylinders fuelType  ...            trany  year mpgData_
0       19          4  Regular  ...     Manual 5-spd  1985     True
1        9         12  Regular  ...     Manual 5-spd  1985      NaN
2       23          4  Regular  ...     Manual 5-spd  1985     True
3       10          8  Regular  ...  Automatic 3-spd  1985      NaN
4       17          4  Premium  ...     Manual 5-spd  1993      NaN
..     ...        ...      ...  ...              ...   ...      ...
95      17          6  Regular  ...  Automatic 3-spd  1993     True
96      17          6  Regular  ...  Automatic 4-spd  1993      NaN
97      15          6  Regular  ...  Automatic 4-spd  1993      NaN
98      15          6  Regular  ...     Manual 5-spd  1993      NaN
99       9          8  Premium  ...  Automatic 4-spd  1993      NaN
[100 rows x 11 columns]
```

Now your DataFrame has been modified again using `.sort_index()`. Since your DataFrame still has its default index, sorting it in ascending order puts the data back into its original order.

If you're familiar with Python's built-in functions `sort()` and `sorted()`, then the `inplace` parameter available in the pandas sort methods might feel very similar. For more information, you can check out How to Use sorted() and sort() in Python.

Conclusion

Conclusion

You now know how to use two core methods of the pandas library: `.sort_values()` and `.sort_index()`. With this knowledge, you can perform basic data analysis with a DataFrame. While there are a lot of similarities between these two methods, seeing the difference between them makes it clear which one to use for different analytical tasks.

In this tutorial, you've learned how to:

- Sort a **pandas DataFrame** by the values of one or more columns
- Use the `ascending` parameter to change the **sort order**
- Sort a DataFrame by its `index` using **`.sort_index()`**
- Organize **missing data** while sorting values
- Sort a DataFrame **in place** using `inplace` set to `True`

These methods are a big part of being proficient with data analysis. They'll help you build a strong foundation on which you can perform more advanced pandas operations. If you want to see some examples of more advanced uses of pandas sort methods, then the pandas documentation is a great resource.

ever. Unsubscribe any time. Curated by the Real Python team.

```
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Spencer Guy**

Spencer is data engineer who loves Python and automation. He is a self-taught Python programmer who works with data science teams and builds data intensive applications.

» More about Spencer

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Bartosz

Geir Arne

Joanna

Jacob

What Do You Think?

Rate this article:

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: `data-science` `intermediate`

Recommended Video Course: Sorting Data in Python With Pandas