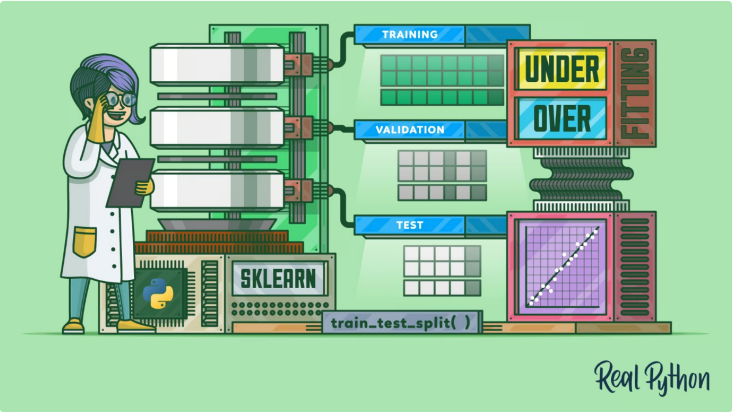



# Split Your Dataset With scikit-learn's train_test_split()

by Mirko Stojiljković Nov 23, 2020 3 Comments

data-science intermediate machine-learning numpy



## Table of Contents


- The Importance of Data Splitting
  - Training, Validation, and Test Sets
  - Underfitting and Overfitting
- Prerequisites for Using train_test_split()
- Application of train_test_split()
- Supervised Machine Learning With train_test_split()
  - Minimalist Example of Linear Regression
  - Regression Example
  - Classification Example
- Other Validation Functionalities
- Conclusion




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: **Splitting Datasets With scikit-learn and train_test_split()**

One of the key aspects of supervised machine learning is model evaluation and validation. When you evaluate the predictive performance of your model, it's essential that the process be unbiased. Using **`train_test_split()`** from the data science library scikit-learn, you can split your dataset into subsets that minimize the potential for bias in your evaluation and validation process.

**In this tutorial, you'll learn:**

- Why you need to **split your dataset** in supervised machine learning
- Which **subsets** of the dataset you need for an unbiased evaluation of your model
- How to use **`train_test_split()`** to split your data
- How to combine `train_test_split()` with **prediction methods**

In addition, you'll get information on related tools from `sklearn.model_selection`.

**Free Bonus: Click here to get access to a free NumPy Resources Guide** that points you to the best tutorials, videos, and books for improving your NumPy skills.

## The Importance of Data Splitting

Supervised machine learning is about creating models that precisely map the given inputs (independent variables, or **predictors**) to the given outputs (dependent variables, or



Table of Contents

- The Importance of Data Splitting
- Prerequisites for Using train_test_split()
- Application of train_test_split()
- Supervised Machine Learning With train_test_split()
- Other Validation Functionalities
- Conclusion

**responses**).

How you measure the precision of your model depends on the type of a problem you're trying to solve. In regression analysis, you typically use the coefficient of determination, root-mean-square error, mean absolute error, or similar quantities. For classification problems, you often apply accuracy, precision, recall, F1 score, and related indicators.

The acceptable numeric values that measure precision vary from field to field. You can find detailed explanations from Statistics By Jim, Quora, and many other resources.

What's most important to understand is that you usually need **unbiased evaluation** to properly use these measures, assess the predictive performance of your model, and validate the model.

This means that you can't evaluate the predictive performance of a model with the same data you used for training. You need evaluate the model with **fresh data** that hasn't been seen by the model before. You can accomplish that by splitting your dataset before you use it.



### Training, Validation, and Test Sets

Splitting your dataset is essential for an unbiased evaluation of prediction performance. In most cases, it's enough to split your dataset randomly into three subsets:

1. **The training set** is applied to train, or **fit**, your model. For example, you use the training set to find the optimal weights, or coefficients, for linear regression, logistic regression, or neural networks.

2. **The validation set** is used for unbiased model evaluation during hyperparameter tuning. For example, when you want to find the optimal number of neurons in a neural network or the best kernel for a support vector machine, you experiment with different values. For each considered setting of hyperparameters, you fit the model with the training set and assess its performance with the validation set.

3. **The test set** is needed for an unbiased evaluation of the final model. You shouldn't use it for fitting or validation.

In less complex cases, when you don't have to tune hyperparameters, it's okay to work with only the training and test sets.

### Underfitting and Overfitting

Splitting a dataset might also be important for detecting if your model suffers from one of two very common problems, called underfitting and overfitting:

1. **Underfitting** is usually the consequence of a model being unable to encapsulate the relations among data. For example, this can happen when trying to represent nonlinear relations with a linear model. Underfitted models will likely have poor performance with both training and test sets.

2. **Overfitting** usually takes place when a model has an excessively complex structure and learns both the existing relations among data and noise. Such models often have bad generalization capabilities. Although they work well with training data, they usually yield poor performance with unseen (test) data.

You can find a more detailed explanation of underfitting and overfitting in Linear Regression in Python.

## Prerequisites for Using `train_test_split()`

Now that you understand the need to split a dataset in order to perform unbiased model evaluation and identify underfitting or overfitting, you're ready to learn how to split your own datasets.

You'll use version 0.23.1 of **scikit-learn**, or **`sklearn`**. It has many packages for data science and machine learning, but for this tutorial you'll focus on the **`model_selection`** package, specifically on the function **`train_test_split()`**.

You can install `sklearn` with `pip install`:

Shell

```
$ python -m pip install -U "scikit-learn==0.23.1"
```

If you use Anaconda, then you probably already have it installed. However, if you want to use a fresh environment, ensure that you have the specified version, or use Miniconda, then you can install `sklearn` from Anaconda Cloud with `conda install`:

```
Shell
$ conda install -c anaconda scikit-learn=0.23
```

You'll also need NumPy, but you don't have to install it separately. You should get it along with `sklearn` if you don't already have it installed. If you want to refresh your NumPy knowledge, then take a look at the official documentation or check out Look Ma, No `for` Loops: Array Programming With NumPy.

## Application of `train_test_split()`

You need to import `train_test_split()` and NumPy before you can use them, so you can start with the `import` statements:

```
Python
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
```

Now that you have both imported, you can use them to split data into training sets and test sets. You'll split inputs and outputs at the same time, with a single function call.

With `train_test_split()`, you need to provide the sequences that you want to split as well as any optional arguments. It returns a list of NumPy arrays, other sequences, or SciPy sparse matrices if appropriate:

```
Python
sklearn.model_selection.train_test_split(*arrays, **options) -> list
```

**`arrays`** is the sequence of lists, NumPy arrays, pandas DataFrames, or similar array-like objects that hold the data you want to split. All these objects together make up the dataset and must be of the same length.

In supervised machine learning applications, you'll typically work with two such sequences:

1. A two-dimensional array with the inputs (`x`)
2. A one-dimensional array with the outputs (`y`)

**`options`** are the optional keyword arguments that you can use to get desired behavior:

- **`train_size`** is the number that defines the size of the training set. If you provide a `float`, then it must be between `0.0` and `1.0` and will define the share of the dataset used for testing. If you provide an `int`, then it will represent the total number of the training samples. The default value is `None`.
- **`test_size`** is the number that defines the size of the test set. It's very similar to `train_size`. You should provide either `train_size` or `test_size`. If neither is given, then the default share of the dataset that will be used for testing is `0.25`, or 25 percent.
- **`random_state`** is the object that controls randomization during splitting. It can be either an `int` or an instance of `RandomState`. The default value is `None`.
- **`shuffle`** is the Boolean object (`True` by default) that determines whether to shuffle the dataset before applying the split.
- **`stratify`** is an array-like object that, if not `None`, determines how to use a stratified split.

Now it's time to try data splitting! You'll start by creating a simple dataset to work with. The dataset will contain the inputs in the two-dimensional array `x` and outputs in the one-dimensional array `y`:

```
Python
>>> x = np.arange(1, 25).reshape(12, 2)
>>> y = np.array([0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0])
>>> x
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12],
       [13, 14],
```

```
       [15, 16],
       [17, 18],
       [19, 20],
       [21, 22],
       [23, 24]])
>>> y
array([0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0])
```

To get your data, you use `arange()`, which is very convenient for generating arrays based on numerical ranges. You also use `.reshape()` to modify the shape of the array returned by `arange()` and get a two-dimensional data structure.

You can split both input and output datasets with a single function call:

Python

```
>>> x_train, x_test, y_train, y_test = train_test_split(x, y)
>>> x_train
array([[15, 16],
       [21, 22],
       [11, 12],
       [17, 18],
       [13, 14],
       [ 9, 10],
       [ 1,  2],
       [ 3,  4],
       [19, 20]])
>>> x_test
array([[ 5,  6],
       [ 7,  8],
       [23, 24]])
>>> y_train
array([1, 1, 0, 1, 0, 1, 0, 1, 0])
>>> y_test
array([1, 0, 0])
```

Given two sequences, like `x` and `y` here, `train_test_split()` performs the split and returns four sequences (in this case NumPy arrays) in this order:

1. **`x_train`:** The training part of the first sequence (`x`)
2. **`x_test`:** The test part of the first sequence (`x`)
3. **`y_train`:** The training part of the second sequence (`y`)
4. **`y_test`:** The test part of the second sequence (`y`)

You probably got different results from what you see here. This is because dataset splitting is random by default. The result differs each time you run the function. However, this often isn't what you want.

Sometimes, to make your tests reproducible, you need a random split with the same output for each function call. You can do that with the parameter `random_state`. The value of `random_state` isn't important—it can be any non-negative integer. You could use an instance of `numpy.random.RandomState` instead, but that is a more complex approach.

In the previous example, you used a dataset with twelve observations (rows) and got a training sample with nine rows and a test sample with three rows. That's because you didn't specify the desired size of the training and test sets. By default, 25 percent of samples are assigned to the test set. This ratio is generally fine for many applications, but it's not always what you need.

Typically, you'll want to define the size of the test (or training) set explicitly, and sometimes you'll even want to experiment with different values. You can do that with the parameters `train_size` or `test_size`.

Modify the code so you can choose the size of the test set and get a reproducible result:

Python

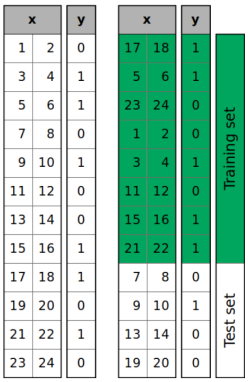
```
>>> x_train, x_test, y_train, y_test = train_test_split(
...     x, y, test_size=4, random_state=4
... )
>>> x_train
array([[17, 18],
       [ 5,  6],
       [23, 24],
       [ 1,  2],
       [ 3,  4],
       [11, 12],
       [15, 16],
       [21, 22]])
>>> x_test
array([[ 7,  8],
       [ 9, 10],
```

```
       [ 9, 10],
       [13, 14],
       [19, 20]])
>>> y_train
array([1, 1, 0, 0, 1, 0, 1, 1])
>>> y_test
array([0, 1, 0, 0])
```

With this change, you get a different result from before. Earlier, you had a training set with nine items and test set with three items. Now, thanks to the argument `test_size=4`, the training set has eight items and the test set has four items. You'd get the same result with `test_size=0.33` because 33 percent of twelve is approximately four.

There's one more very important difference between the last two examples: You now get the same result each time you run the function. This is because you've fixed the random number generator with `random_state=4`.

The figure below shows what's going on when you call `train_test_split()`:




The samples of the dataset are shuffled randomly and then split into the training and test sets according to the size you defined.

You can see that `y` has six zeros and six ones. However, the test set has three zeros out of four items. If you want to (approximately) keep the proportion of `y` values through the training and test sets, then pass `stratify=y`. This will enable stratified splitting:

Python

```
>>> x_train, x_test, y_train, y_test = train_test_split(
...     x, y, test_size=0.33, random_state=4, stratify=y
... )
>>> x_train
array([[21, 22],
       [ 1,  2],
       [15, 16],
       [13, 14],
       [17, 18],
       [19, 20],
       [23, 24],
       [ 3,  4]])
>>> x_test
array([[11, 12],
       [ 7,  8],
       [ 5,  6],
       [ 9, 10]])
>>> y_train
array([1, 0, 1, 0, 1, 0, 0, 1])
>>> y_test
array([0, 0, 1, 1])
```

Now `y_train` and `y_test` have the same ratio of zeros and ones as the original `y` array.

Stratified splits are desirable in some cases, like when you're classifying an **imbalanced dataset**, a dataset with a significant difference in the number of samples that belong to distinct classes.

Finally, you can turn off data shuffling and random split with `shuffle=False`:

Python

```
>>> x_train, x_test, y_train, y_test = train_test_split(
...     x, y, test_size=0.33, shuffle=False
... )
>>> x_train
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
```

```
       [ 7,  8],
       [ 9, 10],
       [11, 12],
       [13, 14],
       [15, 16]])
>>> x_test
array([[17, 18],
       [19, 20],
       [21, 22],
       [23, 24]])
>>> y_train
array([0, 1, 1, 0, 1, 0, 0, 1])
>>> y_test
array([1, 0, 1, 0])
```

Now you have a split in which the first two-thirds of samples in the original x and y arrays are assigned to the training set and the last third to the test set. No shuffling. No randomness.



## Supervised Machine Learning With `train_test_split()`

Now it's time to see `train_test_split()` in action when solving supervised learning problems. You'll start with a small regression problem that can be solved with linear regression before looking at a bigger problem. You'll also see that you can use `train_test_split()` for classification as well.

### Minimalist Example of Linear Regression

In this example, you'll apply what you've learned so far to solve a small regression problem. You'll learn how to create datasets, split them into training and test subsets, and use them for linear regression.

As always, you'll start by importing the necessary packages, functions, or classes. You'll need NumPy, `LinearRegression`, and `train_test_split()`:

Python

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.model_selection import train_test_split
```

Now that you've imported everything you need, you can create two small arrays, x and y, to represent the observations and then split them into training and test sets just as you did before:

Python

```
>>> x = np.arange(20).reshape(-1, 1)
>>> y = np.array([5, 12, 11, 19, 30, 29, 23, 40, 51, 54, 74,
...               62, 68, 73, 89, 84, 89, 101, 99, 106])
>>> x
array([[ 0],
       [ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
       [10],
       [11],
       [12],
       [13],
       [14],
       [15],
       [16],
       [17],
       [18],
       [19]])
>>> y
array([  5,  12,  11,  19,  30,  29,  23,  40,  51,  54,  74,  62,  68,
        73,  89,  84,  89, 101,  99, 106])
>>> x_train, x_test, y_train, y_test = train_test_split(
...     x, y, test_size=8, random_state=0
... )
```

Your dataset has twenty observations, or x-y pairs. You specify the argument `test_size=8`, so the dataset is divided into a training set with twelve observations and a test set with eight observations.

Now you can use the training set to fit the model:

```
Python
>>> model = LinearRegression().fit(x_train, y_train)
>>> model.intercept_
3.1617195496417523
>>> model.coef_
array([5.53121801])
```

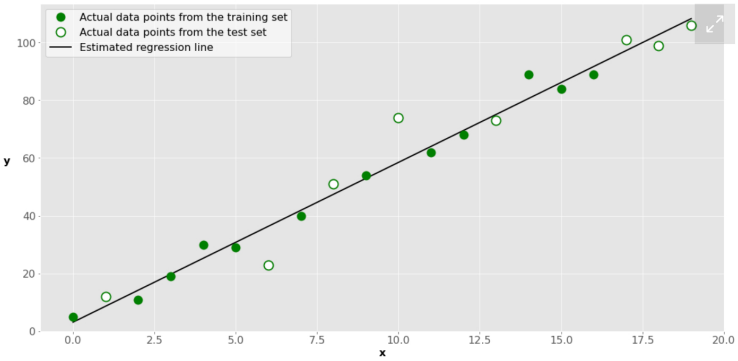
`LinearRegression` creates the object that represents the model, while `.fit()` trains, or fits, the model and returns it. With linear regression, fitting the model means determining the best intercept (`model.intercept_`) and slope (`model.coef_`) values of the regression line.

Although you can use `x_train` and `y_train` to check the goodness of fit, this isn't a best practice. An unbiased estimation of the predictive performance of your model is based on test data:

```
Python
>>> model.score(x_train, y_train)
0.9868175024574795
>>> model.score(x_test, y_test)
0.9465896927715023
```

`.score()` returns the **coefficient of determination**, or $R^2$, for the data passed. Its maximum is 1. The higher the $R^2$ value, the better the fit. In this case, the training data yields a slightly higher coefficient. However, the $R^2$ calculated with test data is an unbiased measure of your model's prediction performance.

This is how it looks on a graph:




The green dots represent the x-y pairs used for training. The black line, called the estimated regression line, is defined by the results of model fitting: the intercept and the slope. So, it reflects the positions of the green dots only.

The white dots represent the test set. You use them to estimate the performance of the model (regression line) with data not used for training.

## Regression Example

Now you're ready to split a larger dataset to solve a regression problem. You'll use a well-known Boston house prices dataset, which is included in `sklearn`. This dataset has 506 samples, 13 input variables, and the house values as the output. You can retrieve it with `load_boston()`.

First, import `train_test_split()` and `load_boston()`:

```
Python
>>> from sklearn.datasets import load_boston
>>> from sklearn.model_selection import train_test_split
```

Now that you have both functions imported, you can get the data to work with:

```
Python
>>> x, y = load_boston(return_X_y=True)
```

As you can see, `load_boston()` with the argument `return_X_y=True` returns a tuple with two NumPy arrays.

two NumPy arrays:

1. A two-dimensional array with the inputs
2. A one-dimensional array with the outputs

The next step is to split the data the same way as before:

```
Python
>>> x_train, x_test, y_train, y_test = train_test_split(
...     x, y, test_size=0.4, random_state=0
... )
```

Now you have the training and test sets. The training data is contained in `x_train` and `y_train`, while the data for testing is in `x_test` and `y_test`.

When you work with larger datasets, it's usually more convenient to pass the training or test size as a ratio. `test_size=0.4` means that approximately 40 percent of samples will be assigned to the test data, and the remaining 60 percent will be assigned to the training data.

Finally, you can use the training set (`x_train` and `y_train`) to fit the model and the test set (`x_test` and `y_test`) for an unbiased evaluation of the model. In this example, you'll apply three well-known regression algorithms to create models that fit your data:

1. Linear regression with `LinearRegression()`
2. Gradient boosting with `GradientBoostingRegressor()`
3. Random forest with `RandomForestRegressor()`

The process is pretty much the same as with the previous example:

1. **Import** the classes you need.
2. **Create** model instances using these classes.
3. **Fit** the model instances with **`.fit()`** using the training set.
4. **Evaluate** the model with **`.score()`** using the test set.

Here's the code that follows the steps described above for all three regression algorithms:

```
Python
>>> from sklearn.linear_model import LinearRegression
>>> model = LinearRegression().fit(x_train, y_train)
>>> model.score(x_train, y_train)
0.7668160223286261
>>> model.score(x_test, y_test)
0.6882607142538016

>>> from sklearn.ensemble import GradientBoostingRegressor
>>> model = GradientBoostingRegressor(random_state=0).fit(x_train, y_train)
>>> model.score(x_train, y_train)
0.9859065238883613
>>> model.score(x_test, y_test)
0.8530127436482149

>>> from sklearn.ensemble import RandomForestRegressor
>>> model = RandomForestRegressor(random_state=0).fit(x_train, y_train)
>>> model.score(x_train, y_train)
0.9811695664860354
>>> model.score(x_test, y_test)
0.8325867908704008
```

You've used your training and test datasets to fit three models and evaluate their performance. The measure of accuracy obtained with `.score()` is the coefficient of determination. It can be calculated with either the training or test set. However, as you already learned, the score obtained with the test set represents an unbiased estimation of performance.

As mentioned in the documentation, you can provide optional arguments to `LinearRegression()`, `GradientBoostingRegressor()`, and `RandomForestRegressor()`. `GradientBoostingRegressor()` and `RandomForestRegressor()` use the `random_state` parameter for the same reason that `train_test_split()` does: to deal with randomness in the algorithms and ensure reproducibility.

For some methods, you may also need feature scaling. In such cases, you should fit the scalers with training data and use them to transform test data.



## Classification Example

You can use `train_test_split()` to solve **classification** problems the same way you do for regression analysis. In machine learning, classification problems involve training a model to apply labels to, or classify, the input values and sort your dataset into categories.

In the tutorial Logistic Regression in Python, you'll find an example of a handwriting recognition task. The example provides another demonstration of splitting data into training and test sets to avoid bias in the evaluation process.

## Other Validation Functionalities

The package `sklearn.model_selection` offers a lot of functionalities related to model selection and validation, including the following:

- Cross-validation
- Learning curves
- Hyperparameter tuning

Cross-validation is a set of techniques that combine the measures of prediction performance to get more accurate model estimations.

One of the widely used cross-validation methods is *k*-fold cross-validation. In it, you divide your dataset into *k* (often five or ten) subsets, or **folds**, of equal size and then perform the training and test procedures *k* times. Each time, you use a different fold as the test set and all the remaining folds as the training set. This provides *k* measures of predictive performance, and you can then analyze their mean and standard deviation.

You can implement cross-validation with `KFold`, `StratifiedKFold`, `LeaveOneOut`, and a few other classes and functions from `sklearn.model_selection`.

A learning curve, sometimes called a training curve, shows how the prediction score of training and validation sets depends on the number of training samples. You can use `learning_curve()` to get this dependency, which can help you find the optimal size of the training set, choose hyperparameters, compare models, and so on.

Hyperparameter tuning, also called hyperparameter optimization, is the process of determining the best set of hyperparameters to define your machine learning model. `sklearn.model_selection` provides you with several options for this purpose, including `GridSearchCV`, `RandomizedSearchCV`, `validation_curve()`, and others. Splitting your data is also important for hyperparameter tuning.

## Conclusion

You now know why and how to use **`train_test_split()`** from `sklearn`. You've learned that, for an unbiased estimation of the predictive performance of machine learning models, you should use data that hasn't been used for model fitting. That's why you need to split your dataset into training, test, and in some cases, validation subsets.

**In this tutorial, you've learned how to:**

- Use **`train_test_split()`** to get training and test sets
- Control the size of the subsets with the parameters **`train_size`** and **`test_size`**
- Determine the **randomness** of your splits with the `random_state` parameter
- Obtain **stratified splits** with the `stratify` parameter
- Use `train_test_split()` as a part of **supervised machine learning** procedures

You've also seen that the `sklearn.model_selection` module offers several other tools for model validation, including cross-validation, learning curves, and hyperparameter tuning.

If you have questions or comments, then please put them in the comment section below.

Real Python team.

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

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

Email Address

Send Me Python Tricks »

## About **Mirko Stojiljković**

Mirko has a Ph.D. in Mechanical Engineering and works as a university professor. He is a Pythonista who applies hybrid optimization and machine learning methods to support decision making in the energy sector.

» More about Mirko

*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

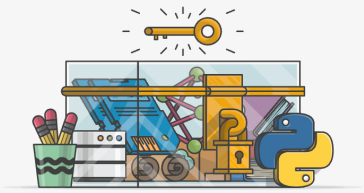
 Geir Arne

 Joanna

 Jacob

 Kyle

## Master Real-World Python Skills With Unlimited Access to Real Python



**Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:**

Level Up Your Python Skills »

## What Do You Think?

**Rate this article:**

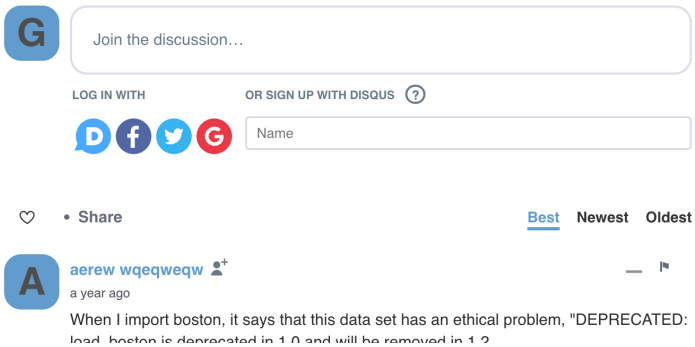
LinkedIn Twitter Facebook Email

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




Help