How to Calculate Z-Scores in Python Using SciPy

Authored by

stats writer

March 15, 2026

Calculate Z-Scores in Python

In the expansive realm of **statistical analysis**, the **z-score** serves as a fundamental metric for understanding the relative position of a data point within a larger dataset. By definition, a **z-score** (also frequently referred to as a **standard score**) quantifies exactly how many **standard deviations** a specific observation or raw value lies above or below the **arithmetic mean** of the group. This transformation is vital for **normalization**, allowing researchers and data scientists to compare scores from different distributions or scales on a standardized basis. When a **z-score** is positive, it indicates the value is higher than the average; conversely, a negative score signifies the value is lower than the average.

The mathematical procedure to derive this value is remarkably straightforward but powerful. We utilize a specific formula to calculate a **z-score** that accounts for both the central tendency and the dispersion of the data. By subtracting the **population mean** from a raw score and then dividing the result by the **population standard deviation**, we effectively "center" the data around zero and "scale" it according to its variance. This process is the cornerstone of many advanced statistical techniques and **machine learning** algorithms, ensuring that features with different magnitudes do not disproportionately influence the results of a model.

$\mathbf{z} = (X - \mu) / \sigma$

Within this formula, the variables are defined as follows:

X represents a single raw data value or observation from the dataset that you wish to standardize.
μ (mu) denotes the **mean** of the entire population, providing the central anchor point for the calculation.
σ (sigma) signifies the **standard deviation** of the population, which measures the extent of variation or dispersion in the values.

This comprehensive tutorial is designed to guide you through the practical steps of calculating **z-scores** for raw data values using **Python**. We will explore various libraries and methods to handle different data structures, ranging from simple lists to complex multi-dimensional matrices and organized tables. By the end of this guide, you will be equipped to implement these statistical transformations efficiently in your own data science projects.

Understanding the Core Mechanics of Z-Score Calculation in Python

While one could manually implement the **z-score** formula using basic arithmetic operators, the **Python** ecosystem offers highly optimized libraries that streamline this process. The most common and robust tool for this task is the **SciPy** library, specifically the **stats** module. This module contains a dedicated function called **zscore**, which is engineered to handle large-scale computations with high numerical precision. Utilizing pre-built functions not only reduces the likelihood of manual coding errors but also leverages **vectorized operations** for significantly faster performance on extensive datasets.

The **scipy.stats.zscore** function is versatile, allowing for the calculation of scores across different axes and the adjustment of **degrees of freedom**. This flexibility is essential when moving between theoretical population parameters and practical sample statistics. Furthermore, the function provides built-in mechanisms for handling missing data, which is a common challenge in real-world **data cleaning** and preprocessing workflows. Understanding the nuances of these parameters is key to obtaining accurate and meaningful results.

The standard syntax for invoking this function is as follows:

scipy.stats.zscore(a, axis=0, ddof=0, nan_policy='propagate')

To use this tool effectively, it is important to understand what each parameter controls within the calculation engine:

a: This is the primary input, typically an **array**-like object such as a list, a **NumPy** array, or a series that contains the data points you intend to transform.
axis: This parameter determines the direction of the calculation. In a 2D matrix, setting the axis to 0 calculates scores column-wise, while setting it to 1 calculates them row-wise. The default value is 0.
ddof: This stands for "Delta Degrees of Freedom." It allows you to adjust the **standard deviation** calculation. By default, it is set to 0 (for population statistics), but you might set it to 1 when working with sample data to apply Bessel's correction.
nan_policy: This defines the behavior when the input contains "Not a Number" (**NaN**) values. "Propagate" returns **NaN** for the entire calculation, "raise" triggers an error, and "omit" performs the calculation while ignoring the missing entries.

The following sections will provide detailed illustrations of how to apply this function across various common data formats, including **one-dimensional** arrays, **multi-dimensional** matrices, and **Pandas** **DataFrames**.

Calculating Z-Scores for One-Dimensional NumPy Arrays

The most basic application of the **z-score** involves a simple list or a **one-dimensional** **NumPy array**. This scenario is common when you are analyzing a single variable, such as the heights of individuals in a group or the test scores of a specific class. To begin, you must ensure that your environment has the necessary libraries installed and imported. We typically import **Pandas**, **NumPy**, and the **stats** module from **SciPy** to create a robust workspace for our analysis.

Step 1: Environment Setup and Module Importation.

import pandas as pd
import numpy as np
import scipy.stats as stats

Once the environment is ready, the next step involves defining the dataset. In this example, we will manually create a **NumPy** **array** containing ten integer values. These values represent our raw data points before any statistical transformation has been applied.

Step 2: Constructing the Data Array.

data = np.array()

With the data prepared, we can now invoke the **zscore** function. The function will iterate through each element in the **array**, calculate the **mean** and **standard deviation** of the entire set, and then apply the formula to each individual point. The output will be a new **array** of the same length, containing the calculated **standard scores**.

Step 3: Execution and Interpretation of Results.

stats.zscore(data)

The resulting **z-scores** provide immediate insight into the distribution of the data. By examining these numbers, we can draw the following conclusions regarding our specific dataset:

The first value in our array (6) results in a **z-score** of **-1.394**. This indicates that the value is approximately 1.39 **standard deviations** *below* the **mean**.
The fifth value (13) has a **z-score** of exactly **0**. This reveals that the value is perfectly equal to the **mean** of the dataset, representing the center point.
The final value (22) yields a **z-score** of **1.793**. This tells us the value is 1.79 **standard deviations** *above* the **mean**, identifying it as one of the higher observations in this set.

Processing Multi-Dimensional Arrays and Axis Specification

In more complex data scenarios, you will likely encounter **multi-dimensional** arrays or matrices. When working with these structures, it is rarely useful to calculate a single **z-score** across the entire matrix. Instead, you usually need to standardize values relative to their specific row or column. This is where the **axis** parameter becomes critical. By adjusting the axis, you can instruct

SciPy to treat each sub-array as an independent distribution, which is essential for tasks like **feature scaling** in **machine learning** where different columns represent different physical units or variables.

Consider a situation where we have a 3x5 matrix. Each row might represent a different experimental trial, and we want to know how each data point compares to the other values within its specific trial. We define our **multi-dimensional NumPy array** as follows:

data = np.array(,
,
])

To calculate the **z-scores** within each row independently, we set the **axis** parameter to 1. This tells the function to compute the **mean** and **standard deviation** across the columns for each row. The resulting output will maintain the original shape of the matrix but with standardized values.

stats.zscore(data, axis=1)

,
,
]

Analyzing these results reveals how the same raw number can have a different **standard score** depending on the context of its group. For instance, notice the following observations:

The first value in the first row is 5, and its **z-score** is **-1.569** relative to its row's average.
In the second row, the first value is 8, which results in a **z-score** of **-0.816**. Even though 8 is numerically larger than 5, it is actually closer to its respective row's **mean** than 5 was to its own.
The first value of 2 in the third row shows a **z-score** of **-1.167**. This demonstrates how **normalization** removes the absolute scale, focusing instead on the **variance** within each specific group.

Advanced Integration with Pandas DataFrames

For data scientists and analysts, the **DataFrame** is the primary structure for data manipulation. **Pandas** provides a high-level interface that is ideal for managing tabular data with named columns. While **SciPy** functions work well with **NumPy**, we can easily integrate them into a **Pandas** workflow using the **apply** method. This allows us to perform a column-wise **z-score** transformation across the entire table in a single, readable line of code.

Let's create a sample **DataFrame** containing random integers to simulate a dataset with three different features, labeled 'A', 'B', and 'C'. These features could represent anything from stock prices to sensor readings.

data = pd.DataFrame(np.random.randint(0, 10, size=(5, 3)), columns=)
data

A B C
0 8 0 9
1 4 0 7
2 9 6 8
3 1 8 1
4 8 0 8

To standardize this entire dataset, we use the **apply** function, passing **stats.zscore** as the argument. By default, **apply** works column-wise, which is typically the desired behavior in data science since each column usually represents a distinct variable with its own **mean** and **standard deviation**.

data.apply(stats.zscore)

A B C
0 0.659380 -0.802955 0.836080
1 -0.659380 -0.802955 0.139347
2 0.989071 0.917663 0.487713
3 -1.648451 1.491202 -1.950852
4 0.659380 -0.802955 0.487713

The resulting **DataFrame** now contains the **z-scores** for every cell. This transformation is incredibly useful for identifying **outliers** or preparing data for **algorithms** like K-Means clustering or Principal Component Analysis (PCA). We can interpret the results as follows:

In column A, the value 8 is **0.659** **standard deviations** *above* the column's **mean**.
In column B, the value 0 is **0.803** **standard deviations** *below* the column's **mean**.
In column C, the value 9 is **0.836** **standard deviations** *above* the column's **mean**.

Practical Applications and Conclusion

Standardizing data using **z-scores** is a critical step in many analytical pipelines. One of the most common applications is in the detection of **outliers**. Generally, any data point with a **z-score** greater than 3 or less than -3 is considered an extreme outlier, as it lies very far from the **mean** in a

normal distribution. Identifying these points allows analysts to investigate potential data entry errors or unique phenomena that deviate from the norm.

Another essential application is in **machine learning**. Many **gradient-based algorithms**, such as linear regression or neural networks, converge much faster when the input features are on a similar scale. Without **normalization** via **z-scores**, a feature with a large numerical range (like "Annual Income") would dominate a feature with a small range (like "Age"), leading to biased models and poor predictive performance. By transforming all features to have a **mean** of 0 and a **standard deviation** of 1, you ensure a level playing field for all variables.

In summary, **Python** provides an elegant and efficient set of tools through **SciPy**, **NumPy**, and **Pandas** to calculate **z-scores**. Whether you are working with simple **one-dimensional** lists or complex **DataFrames**, these libraries offer the flexibility to handle various dimensions, **degrees of freedom**, and missing data patterns. Mastering these techniques is an essential skill for any aspiring data professional, enabling more accurate comparisons and robust statistical modeling.