How to Easily Perform Linear Interpolation in Python

Authored by
stats writer

December 2, 2025

RECOMMENDED CITATION

stats writer (2025). *How to Easily Perform Linear Interpolation in Python*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=103773

ARABPSYCHOLOGY.COM

Understanding the Concept of Linear Interpolation

Linear interpolation is a fundamental mathematical technique employed across various fields, including numerical analysis, computer graphics, and data science. At its core, it provides a method for estimating an unknown value that lies between two known values. When dealing with a series of data points, we often need to predict the behavior of the underlying function at locations where measurements were not taken. This is precisely where **linear interpolation** becomes invaluable. In the context of Python programming, this technique is used to calculate the value of a function between two adjacent points on a line, assuming that the function behaves linearly over that short interval.

The objective is simple: given a set of discrete measurements, we want to create a continuous approximation. By drawing a straight line connecting two adjacent known data points, we can derive an estimate for any intermediate input value. This approach is powerful due to its computational simplicity and efficiency, making it a common choice for quick estimations. When implementing this in a robust environment like **Python**, specialized libraries handle the complex calculations, providing efficient and reliable results for large datasets.

To effectively perform this calculation in **Python**, practitioners rely heavily on the scientific ecosystem, specifically modules like SciPy. The `interp1d` function, residing within the `scipy.interpolate` module, is the primary tool for one-dimensional interpolation. This function ingests the known independent (x) and dependent (y) variable arrays, and subsequently generates a callable interpolated function. This generated function can then be queried with any new x-value within the range of the original data set to return a corresponding estimated y-value.

Linear interpolation is the process of estimating an unknown value of a function between two known values.

Given two known values (x1, y1) and (x2, y2), we can estimate the y-value for some point x by using the following formula:

$$y = y1 + (x-x1)(y2-y1)/(x2-x1)$$

The Mathematical Principle Behind Linear Interpolation

The core of **linear interpolation** relies on the fundamental principles of geometry and algebra, specifically the concept of similar triangles used to determine the slope between two known data points. If we consider two points, $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, the goal is to find a point $P = (x, y)$ that lies exactly on the straight line segment connecting P_1 and P_2. The mathematical formula provided above is simply a rearrangement of the slope equality principle, ensuring that the rate of change is constant between all points on that line segment.

ARABPSYCHOLOGY.COM

The term $(y_2 - y_1) / (x_2 - x_1)$ represents the constant slope (m) of the line connecting the two known points. The formula is essentially calculating the vertical change (rise) corresponding to the horizontal distance $(x - x_1)$ from the first point, and then adding that change to the initial value y_1. This ensures proportional scaling. While the formula is straightforward, implementing it manually for large datasets in **Python** can be cumbersome, which is why specialized libraries like SciPy are preferred for performance and stability.

When applying this method, it is crucial that the target input value x falls strictly between x_1 and x_2. If x falls outside this range, the process is referred to as **linear extrapolation**. While similar mathematically, extrapolation carries a much higher risk of error, as it assumes the linear trend observed locally continues indefinitely. For reliable data processing, particularly within scientific computing environments, interpolation is the safer and more widely accepted practice.

Implementing Interpolation Using SciPy

While it is theoretically possible to implement the interpolation formula using basic **Python** arithmetic, the robust and efficient approach involves leveraging the specialized tools available in the SciPy library, which is built upon the foundational numerical capabilities of NumPy. SciPy's `scipy.interpolate` module provides a suite of functions designed for various interpolation needs, ranging from simple linear methods to complex spline and multi-dimensional techniques. This module ensures high numerical accuracy and optimized performance, critical for handling real-world scientific and engineering data.

The primary function for one-dimensional **linear interpolation** is `interp1d`. This function handles the heavy lifting by taking the input data arrays and generating an interpolation object. Unlike manual calculation, `interp1d` automatically manages the sorting of data and ensures that the interpolation is performed correctly across the entire defined range of x-values. This abstracts away the need to manually identify the bounding points (x_1, y_1) and (x_2, y_2) for every new query point x.

To begin, we must first ensure that SciPy is imported into our **Python** environment. Once imported, we can call the `interp1d` function, passing it our known x and y arrays. The output of this function call is not the interpolated value itself, but rather a function--a callable object--that is optimized to calculate the interpolated y-value corresponding to any new x-value we feed it. This object-oriented approach is highly efficient, especially when performing numerous interpolation lookups against the same initial data set.

We can use the following basic syntax to perform linear interpolation in Python:

import scipy.interpolate

ARABPSYCHOLOGY.COM

```
y_interp = scipy.interpolate.interp1d(x, y)

#find y-value associated with x-value of 13
print(y_interp(13))
```

The following example shows how to use this syntax in practice.

Detailed Breakdown of the interp1d Function

The `interp1d` function is central to this process. It requires, at minimum, two arguments: the array of known independent variables (`x`) and the array of known dependent variables (`y`). It is designed to handle one-dimensional arrays, meaning it works along a single axis. Crucially, the function offers optional parameters that allow for customization of the interpolation method. By default, `interp1d` performs **linear interpolation**, which assumes straight-line segments between points.

However, for situations requiring smoother approximations, the `kind` parameter can be specified. While we focus on the default linear method here, users can opt for other kinds of interpolation, such as cubic spline interpolation, which provides a much smoother curve by using higher-order polynomials to connect points. The choice of interpolation kind depends entirely on the nature of the data and the required smoothness of the estimate. For engineering data where local changes are often sharp, **linear interpolation** often provides a sufficiently accurate and conservative estimate.

Another vital parameter is `fill_value`. When attempting to query the interpolated function with an x-value that lies outside the range of the initial input data (extrapolation), `interp1d` defaults to raising a `ValueError`, thus preventing potentially misleading results from extrapolation. By setting `fill_value='extrapolate'`, we explicitly instruct the function to perform linear extrapolation using the slope defined by the closest two boundary points. Alternatively, `fill_value` can be set to a fixed numerical value (e.g., 0) to return a constant value whenever the query falls outside the bounds of the original data set. Understanding these parameters is essential for robust numerical analysis in Python.

Practical Example: Setting Up the Data Set

To demonstrate the utility of **linear interpolation** in Python, let us consider a practical scenario involving sequential measurements. Suppose we have collected time-series data where `x` represents time intervals (perhaps in minutes) and `y` represents a measured quantity (such as temperature or pressure). Our goal is to estimate the value of this quantity at a specific time point that falls between two consecutive measurements.

ARABPSYCHOLOGY.COM

We begin by defining our known data points using standard **Python** lists, which will later be converted into NumPy arrays internally by the SciPy function. The input arrays must be of equal length and must be ordered according to the independent variable (x). If the x-values are not strictly increasing, interpolation methods will often fail or yield incorrect results, making data pre-processing a critical step before feeding the arrays into `interp1d`.

The following defined lists simulate a scenario where the dependent variable `y` exhibits a clear non-linear, increasing trend relative to the independent variable `x`. Even though the overall trend is non-linear, **linear interpolation** assumes that the change between any two adjacent points is approximately linear, which is a reasonable assumption for sufficiently dense data.

Suppose we have the following two lists of values in Python:

x =
y =

Visualizing the Initial Data Distribution

Before performing any estimation, it is always beneficial practice to visualize the raw data set. Visualization provides immediate insight into the distribution, magnitude, and relationship between the variables, helping to confirm assumptions about linearity (or lack thereof) between points. For data visualization in Python, the `matplotlib` library is the standard choice. We import `matplotlib.pyplot` and use the `plot` function to generate a scatter plot connected by lines.

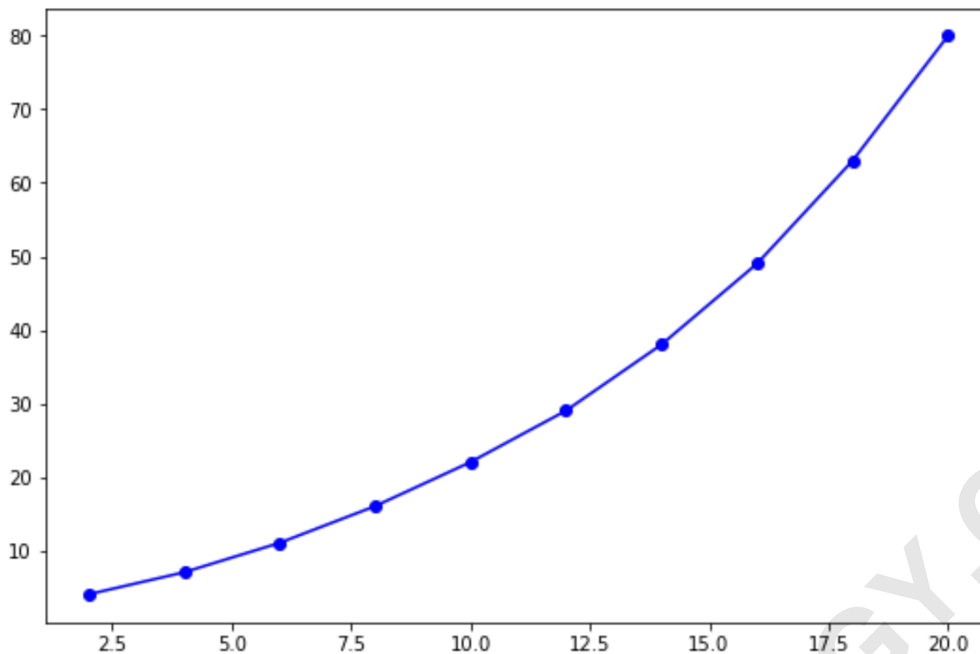
In the code snippet below, we instruct `matplotlib` to plot the `x` and `y` arrays. The format string `'-ob'` specifies that the points should be marked with blue circles (`o`), connected by solid lines (`-`). Observing the resulting graph allows us to visually locate the interval where our desired interpolation point lies, confirming that the function is indeed defined by the provided measurements.

ARABPSYCHOLOGY.COM

We can create a quick plot x vs. y:

import matplotlib.pyplot as plt

#create plot of x vs. y
plt.plot(x, y, '-ob')



ARABPSYCHOLOGY.COM

The resulting visualization confirms that while the data points follow an upward, curved trajectory, the connection between any two adjacent points is assumed to be a straight line for the purpose of **linear interpolation**. We can clearly see the distinct points and the segments connecting them.

Executing the Linear Interpolation Calculation

Now that the data is loaded and visualized, we proceed with the core task: finding the y-value corresponding to a new, arbitrary x-value of **13.** This value falls between the known x-points of 12 (where y=29) and 14 (where y=38). The **linear interpolation** process will calculate a y-value that is proportionally positioned between 29 and 38, based on the relative position of 13 between 12 and 14.

Now suppose that we'd like to find the y-value associated with a new x-value of **13.**

We can use the following code to do so:

```
import scipy.interpolate
y_interp = scipy.interpolate.interp1d(x, y)

#find y-value associated with x-value of 13
print(y_interp(13))

33.5
```

The first two lines initialize the interpolation engine by importing the necessary module and defining the callable interpolation function, `y_interp`, using our arrays `x` and `y`. By passing the target value `13` directly to the `y_interp` object, the SciPy library efficiently identifies the bounding points (12, 29) and (14, 38) and applies the linear formula.

The estimated y-value turns out to be **33.5**.

This result, **33.5**, represents the precise mathematical estimate derived from assuming a straight-line relationship between the data point (12, 29) and the data point (14, 38). Since 13 is exactly halfway between 12 and 14, the estimated y-value is exactly halfway between 29 and 38. The power of the `interp1d` function is that it handles this identification and calculation automatically, requiring minimal input from the user.

Validating the Result through Visualization

While the numerical result of **33.5** is mathematically correct based on the input data, it is imperative to visually confirm that the estimated point integrates seamlessly with the existing trend. Visualization serves as a powerful validation tool, helping to catch conceptual errors or potential issues arising from sparse or incorrectly ordered input data points.

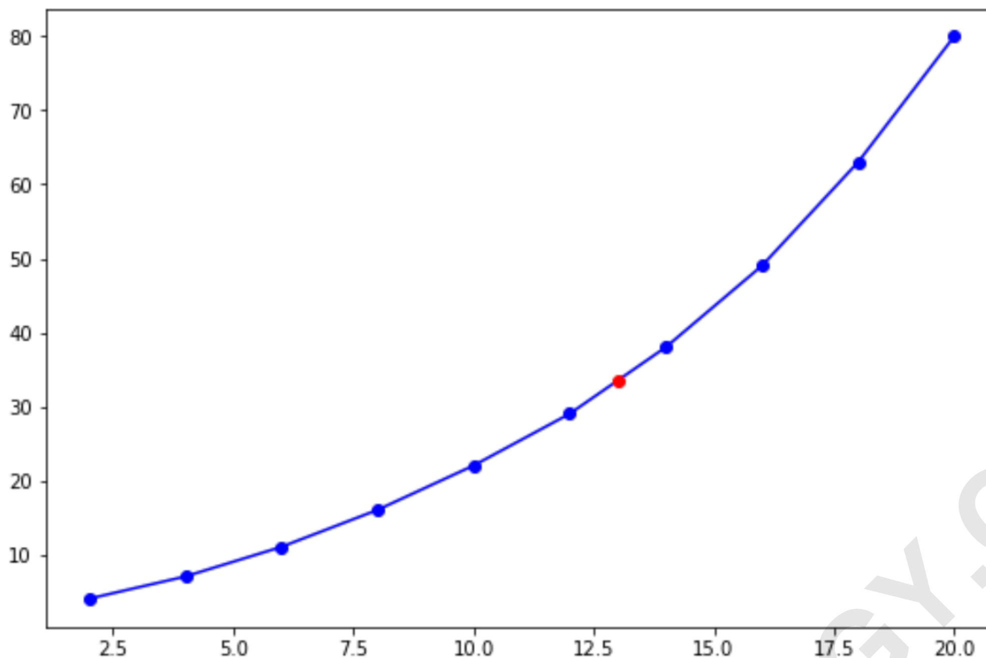
If we add the point (13, 33.5) to our plot, it appears to match the function quite well:

```
import matplotlib.pyplot as plt

#create plot of x vs. y
plt.plot(x, y, '-ob')

#add estimated y-value to plot
plt.plot(13, 33.5, 'ro')
```

ARABPSYCHOLOGY.COM



The updated plot clearly shows the new interpolated point plotted as a red circle (`'ro'`). As expected, this point lies directly on the straight-line segment connecting the known points at $x=12$ and $x=14$. This visualization confirms the successful application of the **linear interpolation** method. Had the calculation been performed using a different, non-linear method (like cubic splines), the new point might slightly deviate from the straight segment while remaining close to the overall trend, demonstrating the impact of method selection on the resulting estimates.

Expanding Beyond Linear Interpolation

While the example provided focused strictly on **linear interpolation**--a first-order method--it is important for analysts utilizing Python to recognize the existence of more complex techniques offered by the `scipy.interpolate` module. The choice of interpolation method significantly impacts the smoothness and accuracy of the resulting function estimate, particularly when dealing with rapidly changing or oscillatory data.

For instance, if the underlying physical process is known to be smooth, using cubic interpolation (setting `kind='cubic'` in `interp1d`) would likely yield a more representative result than the piece-wise linear approach. Cubic splines fit third-degree polynomials between points, ensuring continuity of the first and second derivatives, resulting in a much smoother curve. However, this comes at the cost of increased computational complexity.

The core principle remains constant: the `interp1d` function serves as a flexible gateway to various interpolation algorithms available within SciPy. Mastering this function allows users to quickly

ARABPSYCHOLOGY.COM

switch between estimation methods based on the specific demands of their data analysis, ensuring that the chosen estimation technique is appropriate for the scientific context. Ultimately, this enables robust and adaptable data processing workflows.

We can use this exact formula to perform linear interpolation for any new x-value.

The following tutorials explain how to fix other common errors in Python:

ARABPSYCHOLOGY.COM