Continue

Math absolute value python

You can't perform that action at this time. You can't perform that action at this time. In Python, the absolute value of a number refers to its non-negative value, regardless of its sign. The built-in "abs()" function allows you to quickly determine the absolute value of any number. Whether the number is an integer, floating-point number, or even a complex number, the abs() function can handle them all. Today let PythonCentral take you through the basics, instructions to use Python's absolute value function "abs()", its syntax, practical examples, and a few alternatives as well. What is Absolute Value First, let us learn about absolute value before we head to the advanced concepts. The absolute value of a number is its distance from zero on the number line. This means: Absolute value of 5 is "5" Absolute value of -5 is "5" Absolute value of 0 is "0" How to Use Python abs() Function The "abs()" function is the easiest way to calculate absolute values in Python. Here is the basic syntax for Python absolute value function: abs(number) The number can be an int, float, or complex number (more on these later). Here is an example to understand better: print(abs(-10)) # Gives you an output 10 print(abs(3.5)) # Output will be 3.5 print(abs(0)) # Output will be 0 How to Calculate the Absolute Value of Complex Numbers with Python Let's say you have to calculate "3 + 4j". To calculate the absolute value of complex numbers, we are about to use the abs(complex_num) syntax. Here is an example: complex_num = 3 + 4j print(abs(complex_num)) # Output will be 5.0, calculated as square root of ($3^2 + 4^2$)) Alternative Method to Calculate Absolute Value While the standard practice is to use the Python absolute value function, there are other methods as well that gets you same results. Let us look at few of them. How to Calculate Absolute Value Using Conditional Statements Here is an example syntax where we use conditional statements to calculate the absolute value of a number: def absolute_value(n): return n if n >= 0 else -n print(absolute_value(-7)) # Output will be 7 How to Calculate Absolute Value using NumPy abs Function For arrays and matrices, the numpy.abs() function is more efficient. We strongly recommend going through our NumPy article to learn more later. import numpy as np arr = np.array([-1, -2, 3, -4]) print(np.abs(arr)) # Output will be [1 2 3 4] How to Handle Absolute Values in Real-World Applications Now that we learnt the theory, let us see some practical applications. How to Calculate Distance with Python In case you would like to calculate the distance between two points. Here is how you can do that: def distance(x1, x2): return abs(x1 - x2) print(distance(10, 3)) # Output will return 7 How to Normalize Data Here is a sample syntax where you can calculate the normalized data: data = [-100, -50, 0, 50, 100] norm_data = [abs(num) for num in data] print(norm_data) # Output will fetch [100, 50, 0, 50, 100] Wrapping Up The Python absolute function is a powerful and efficient way to calculate absolute values in Python. Whether you are working with integers, floating points, or complex numbers, using the Python abs() function gives you accuracy and simplicity. We sincerely hope you are now familiar with the abs() function. We recommend you learn more from our other articles as well. Related Articles NumPy where(): Using Conditional Array Operations How To Use NumPy Pad(): Examples And Syntax How to Generate a Random Number in Python Absolute values are commonly used in mathematics, physics, and engineering. Although the school definition of an absolute value might seem straightforward, you can actually look at the concept from many different angles. If you intend to work with absolute values in Python, then you've come to the right place. In this tutorial, you'll learn how to: Implement the absolute value function from scratch Use the built-in abs() function in Python Calculate the absolute values of numbers Call abs() on NumPy arrays and pandas series Customize the behavior of abs() on objects Don't worry if your mathematical knowledge of the absolute value function is a little rusty. You'll begin by refreshing your memory before diving deeper into Python code. That said, feel free to skip the next section and jump right into the nitty-gritty details that follow. The absolute value lets you determine the size or magnitude of an object, such as a number or a vector, regardless of its direction. Real numbers can have one of two directions when you ignore zero: they can be either positive or negative. On the other hand, complex numbers and vectors can have many more directions. Note: When you take the absolute value of a number, you lose information about its sign or, more generally, its direction. Consider a temperature measurement as an example. If the thermometer reads -12°C, then you can say it's twelve degrees Celsius below freezing. Notice how you decomposed the temperature in the last sentence into a magnitude, twelve, and a sign. The phrase below freezing means the same as below zero degrees Celsius. The temperature's size or absolute value is identical to the absolute value of the much warmer +12°C. Using mathematical notation, you can define the absolute value of *x* as a piecewise function, which behaves differently depending on the range of input values. A common symbol for absolute value consists of two vertical lines: Absolute Value Defined as a Piecewise Function This function returns values greater than or equal to zero without alteration. On the other hand, values smaller than zero have their sign flipped from a minus to a plus. Algebraically, this is equivalent to taking the square root of a number squared: Absolute Value Defined Algebraically When you square a real number, you always get a positive result, even if the number that you started with was negative. For example, the square of -12 and the square of 12 have the same value, equal to 144. Later, when you compute the square root of 144, you'll only get 12 without the minus sign. Geometrically, you can think of an absolute value as the distance from the origin, which is zero on a number line in the case of the temperature reading from before: Absolute Value on a Number Line To calculate this distance, you can subtract the origin from the temperature reading (-12°C - 0°C = -12°C) or the other way around (0°C - (-12°C) = +12°C), and then drop the sign of the result. Subtracting zero doesn't make much difference here, but the reference point may sometimes be shifted. That's the case for vectors bound to a fixed point in space, which becomes their origin. Vectors, just like numbers, convey information about the direction and the magnitude of a physical quantity, but in more than one dimension. For example, you can express the velocity of a falling snowflake as a three-dimensional vector: This vector indicates the snowflake's current position relative to the origin of the coordinate system. It also shows the snowflake's direction and pace of motion through the space. The longer the vector, the greater the magnitude of the snowflake's speed. As long as the coordinates of the vector's initial and terminal points are expressed in meters, calculating its length will get you the snowflake's speed measured in meters per unit of time. Note: There are two ways to look at a vector. A bound vector is an ordered pair of fixed points in space, whereas a free vector only tells you about the displacement of the coordinates from point A to point B without revealing their absolute locations. Consider the following code snippet as an example: A bound vector wraps both points, providing quite a bit of information. In contrast, a free vector only represents the shift from A to B. You can calculate a free vector by subtracting the initial point, A, from the terminal one, B. One way to do so is by iterating over the consecutive pairs of coordinates with a list comprehension. A free vector is essentially a bound vector translated to the origin of the coordinate system, so it begins at zero. The length of a vector, also known as its magnitude, is the distance between its initial and terminal points, *A* and *B*, which you can calculate using the Euclidean norm: The Length of a Bound Vector as a Euclidean Norm This formula calculates the length of the *n*-dimensional vector *AB*, by summing the squares of the differences between the coordinates of points *A* and *B* in each dimension indexed by *i*. For a free vector, the initial point, *A*, becomes the origin of the coordinate system—or zero—which simplifies the formula, as you only need to square the coordinates of your vector. Recall the algebraic definition of an absolute value. For numbers, it was the square root of a number squared. Now, when you add more dimensions to the equation, you end up with the formula for the Euclidean norm, shown above. So, the absolute value of a vector is equivalent to its length! All right. Now that you know when absolute values might be useful, it's time to implement them in Python! To implement the absolute value function in Python, you can take one of the earlier mathematical definitions and translate it into code. For instance, the piecewise function may look like this: You use a conditional statement to check whether the given number denoted with the letter x is greater than or equal to zero. If so, then you return the same number. Otherwise, you flip the number's sign. Because there are only two possible outcomes here, you can rewrite the above function using a conditional expression that comfortably fits on a single line: It's exactly the same behavior as before, only implemented in a slightly more compact way. Conditional expressions are useful when you don't have a lot of logic that goes into the two alternative branches in your code. Note: Alternatively, you can write this even more concisely by relying on Python's built-in max() function, which returns the largest argument: If the number *x* is negative, then this function will return its positive value. Otherwise, it'll return *x* itself. The algebraic definition of an absolute value is also pretty straightforward to implement in Python: First, you import the square root function from the math module and then call it on the given number raised to the power of two. The power function is built right into Python, so you don't have to import it. Alternatively, you can avoid the import statement altogether by leveraging Python's exponentiation operator (**), which can simulate the square root function: This is sort of a mathematical trick because using a fractional exponent is equivalent to computing the *n*th root of a number. In this case, you take a squared number to the power of one-half (0.5) or one over two (½), which is the same as calculating the square root. Note that both Python implementations based on the algebraic definition suffer from a slight deficiency: You always end up with a floating-point number, even if you started with an integer. So, if you'd like to preserve the original data type of a number, then you might prefer the piecewise-based implementation instead. As long as you stay within integers and floating-point numbers, you can also write a somewhat silly implementation of the absolute value function by leveraging the textual representation of numbers in Python: You convert the function's argument, x, to a Python string using the built-in str() function. This lets you strip the leading minus sign, if there is one, with an empty string. Then, you convert the result to a floating-point number with float(). Note this implementation always converts integers to floats. Implementing the absolute value function from scratch in Python is a worthwhile learning exercise. However, in real-life applications, you should take advantage of the built-in abs() function that comes with Python. You'll find out why in the next section. The last function that you implemented above was probably the least efficient one because of the data conversions and the string operations, which are usually slower than direct number manipulation. But in truth, all of your hand-made implementations of an absolute value pale in comparison to the abs() function that's built into the language. That's because abs() is compiled to blazing-fast machine code, while your pure-Python code isn't. You should always prefer abs() over your custom functions. It runs much more quickly, an advantage that can really add up when you have a lot of data to process. Additionally, it's much more versatile, as you're about to find out. The abs() function is one of the built-in functions that are part of the Python language. That means you can start using it right away without importing: As you can see, abs() preserves the original data type. In the first case, you passed an integer literal and got an integer result. When called with a floating-point number, the function returned a Python float. But these two data types aren't the only ones that you can call abs() on. The third numeric type that abs() knows how to handle is Python's complex data type, which represents complex numbers. You can think of a complex number as a pair consisting of two floating-point values, commonly known as the real part and the imaginary part. One way to define a complex number in Python is by calling the built-in complex() function: It accepts two arguments. The first one represents the real part, while the second one represents the imaginary part. At any point, you can access the complex number's .real and .imag attributes to get those parts back: Both of them are read-only and are always expressed as floating-point values. Also, the absolute value of a complex number returned by abs() happens to be a floating-point number: This might surprise you until you find out that complex numbers have a visual representation that resembles two-dimensional vectors fixed at the coordinate system's origin: You already know the formula to calculate the length of such a vector, which in this case agrees with the number returned by abs(). Note that the absolute value of a complex number is more commonly referred to as the magnitude, modulus, or radius of a complex number. While integers, floating-point numbers, and complex numbers are the only numeric types supported natively by Python, you'll find two additional numeric types in its standard library. They, too, can interoperate with the abs() function. The abs() function in Python accepts all numeric data types available, including the lesser-known fractions and decimals. For instance, you can get the absolute value of one-third or minus three-quarters defined as Fraction instances: In both cases, you get another Fraction object back, but it's unsigned. That can be convenient if you plan to continue your computations on fractions, which offer higher precision than floating-point numbers. If you're working in finance, then you'll probably want to use Decimal objects to help mitigate the floating-point representation error. Luckily, you can take the absolute value of these objects: Again, the abs() function conveniently returns the same data type as the one that you supplied, but it gives you an appropriate positive value. Wow, abs() can deal with an impressive variety of numeric data types! But it turns out that abs() is even more clever than that. You can even call it on some objects delivered by third-party libraries, as you'll try out in the next section. Say you want to compute the absolute values of average daily temperature readings over some period. Unfortunately, as soon as you try calling abs() on a Python list with those numbers, you get an error: That's because abs() doesn't know how to process a list of numbers. To work around this, you could use a list comprehension or call Python's map() function, like so: Both implementations do the job but require an additional step, which may not always be desirable. If you want to cut that extra step, then you may look into external libraries that change the behavior of abs() for your convenience. That's what you'll explore below. One of the most popular libraries for extending Python with high-performance arrays and matrices is NumPy. Its *n*-dimensional array data structure, ndarray, is the cornerstone of numerical computing in Python, so many other libraries use it as a foundation. Once you convert a regular Python list to a NumPy array with np.array(), you'll be able to call some of the built-in functions, including abs(), on the result: In response to calling abs() on a NumPy array, you get another array with the absolute values of the original elements. It's as if you iterated over the list of temperature readings yourself and applied the abs() function on each element individually, just as you did with a list comprehension before. You can convert a NumPy array back to a Python list if you find that more suitable: However, note that NumPy arrays share most of the Python list interface. For example, they support indexing and slicing, and their methods are similar to those of plain lists, so most people usually just stick to using NumPy arrays without ever looking back at lists. pandas is another third-party library widely used in data analysis thanks to its Series and DataFrame objects. A series is a sequence of observations or a column, whereas a DataFrame is like a table or a collection of columns. You can call abs() on both of them. Suppose you have a Python dictionary that maps a city name to its lowest average temperatures observed monthly over the course of a year: Each city has twelve temperature readings, spanning from January to December. Now, you can turn that dictionary into a pandas DataFrame object so that you can draw some interesting insights going forward: Instead of using the default zero-based index, your DataFrame is indexed by abbreviated month names, which you obtained with the help of the calendar module. Each column in the DataFrame has a sequence of temperatures from the original dictionary, represented as a Series object: By using the square bracket ([]) syntax and a city name like Rovaniemi, you can extract a single Series object from the DataFrame and narrow down the amount of information displayed. pandas, just like NumPy, lets you call many of Python's built-in functions on its objects, including its DataFrame and Series objects. Specifically, you can call abs() to calculate more than one absolute value in one go: Calling abs() on the entire DataFrame applies the function to each element in every column. You can also call abs() on the individual column. How did NumPy and pandas change the behavior of Python's built-in abs() function without modifying its underlying code? Well, it was possible because the function was designed with such extensions in mind. If you're looking for an advanced use of abs(), then read on to make your own data type that'll play nicely with that function. Depending on the data type, Python will handle the computation of absolute values differently. When you call abs() on an integer, it'll use a custom code snippet that resembles your piecewise function. However, that function will be implemented in the C programming language for efficiency. If you pass a floating-point number, then Python will delegate that call to C's fabs() function. In the case of a complex number, it'll call the hypot() function instead. What about container objects like DataFrames, series, and arrays? Understandably, when you define a new data type in Python, it won't work with the abs() function because its default behavior is unknown. However, you can optionally customize the behavior of abs() against the instances of your class by implementing the special .__abs__() method using pure Python. There's a finite set of predefined special methods in Python that let you override how certain functions and operators should work. Consider the following class representing a free *n*-dimensional vector in the Euclidean space: This class accepts one or more coordinate values, describing the displacement in each dimension from the origin of the coordinate system. Your special .__abs__() method calculates the distance from the origin, according to the Euclidean norm definition that you learned at the beginning of this tutorial. To test your new class, you can create a three-dimensional velocity vector of a falling snowflake, for example, which might look like this: Notice how calling abs() on your Vector class instance returns the correct absolute value, equal to about 1.78. The speed units will be expressed in meters per second as long as the snowflake's displacement was measured in meters at two distinct time instants one second apart. In other words, it would take one second for the snowflake to travel from point A to point B. Using the mentioned formula forces you to define the origin point. However, because your Vector class represents a free vector rather than a bound one, you can simplify your code by calculating the multidimensional hypotenuse using Python's math.hypot() function: You get the same result with fewer lines of code. Note that hypot() is a variadic function accepting a variable number of arguments, so you must use the star operator (*) to unpack your tuple of coordinates into those arguments. Awesome! You can now implement your own library, and Python's built-in abs() function will know how to work with it. You'll get functionality similar to working with NumPy or pandas! Implementing formulas for an absolute value in Python is a breeze. However, Python already comes with the versatile abs() function, which lets you calculate the absolute value of various types of numbers, including integers, floating-point numbers, complex numbers, and more. You can also use abs() on instances of custom classes and third-party library objects. In this tutorial, you learned how to: Implement the absolute value function from scratch Use the built-in abs() function in Python Calculate the absolute values of numbers Call abs() on NumPy arrays and pandas series Customize the behavior of abs() on objects With this knowledge, you're equipped with an efficient tool to calculate absolute values in Python. The Python abs() function return the absolute value. The absolute value of any number is always positive it removes the negative sign of a number in Python. Example:Input: -29Output: 29Python abs() Function SyntaxThe abs() function in Python has the following syntax:Syntax: abs(number)number: Integer, floating-point number, complex number.Return: Returns the absolute value.Python abs() Function ExampleLet us see a few examples of the abs() function in Python.abs() Function with an Integer ArgumentIn this example, we will pass an Integer value as an argument to the abs() function in Python and print its value to see how it works. Python # An integer var = -94 print('Absolute value of integer is:', abs(var)) Output:Absolute value of integer is: 94abs() Function with a Floating-Point NumberIn this example, we will pass a float data into the abs() function and it will return an absolute value. Python3 # floating point number float_number = -54.26 print('Absolute value of float is:', abs(float_number)) Output: Absolute value of float is: 54.26abs() Function with a Complex NumberIn this example, we will pass Python complex number into the abs() function and it will return an absolute value. Python # A complex number complex_number = (3 - 4j) print('Absolute value or Magnitude of complex is:', abs(complex_number)) Output: Absolute value or Magnitude of complex is: 5.0Time-Distance calculation using Python abs() FunctionIn this example, the equation shows the relationship between speed, distance traveled, and time taken by an object. We know that speed, time, and distance are never negative. Hence we will use the abs() method to calculate the exact time, distance, and speed.Formula used: Distance = Speed * TimeTime = Distance / SpeedSpeed = Distance / TimeWe declared 3 functions to calculate speed, distance, and time. Then passed the positive and negative integer and float point values to them using the Python abs() function. The abs() function will automatically convert the negative values to positive values, which will be used to calculate speed, distance, and time. Python # Function to calculate speed def cal_speed(dist, time): print(" Distance(km) :", dist) print(" Time(hr) :", time) return dist / time # Function to calculate distance traveled def cal_dis(speed, time): print(" Time(hr) :", time) print(" Speed(km / hr) :", speed) return speed * time # Function to calculate time taken def cal_time(dist, speed): print(" Distance(km) :", dist) print(" Speed(km / hr) :", speed) return dist / speed # Driver Code # Calling function cal_speed() print(" The calculated Speed(km / hr) is :", cal_speed(abs(45.9), abs(-2))) print("") # Calling function cal_dis() print(" The calculated Distance(km) :", cal_dis(abs(-62.9), abs(2.5))) print("") # Calling function cal_time() print(" The calculated Time(hr) :", cal_time(abs(48.0), abs(4.5))) Output Distance(km) : 45.9 Time(hr) : 2 The calculated Speed(km / hr) is : 22.95 Time(hr) : 2.5 Speed(km / hr) : 62.9 The calculated Distance(km) : 157.25 Distance(km) : 48.0 Speed(km / hr) : 4.5 The calculated Time(hr) : 10.666666666666666