Continue

Log base 2 python

Depends on whether the input or output is int or float. assert 5.392317422778761 == math.log2(42.0) assert 5.392317422778761 == math.log(42.0, 2.0) assert 5 == math.frexp(42.0)[1] - 1 assert 5 == (42).bit_length() - 1 float → float math.log2(x) import math log2 = math.log(x, 2.0) log2 = math.log2(x) # python 3.3 or later Thanks @akashchandrakar and @unutbu. If all you need is the integer part of log base 2 of a floating point number, extracting the exponent is pretty efficient: log2int_slow = int(math.floor(math.log(x, 2.0))) # these give the log2int_fast = math.frexp(x)[1] - 1 # same result Python frexp() calls the C function frexp() which just grabs and tweaks the exponent. Python frexp() returns a tuple (mantissa, exponent). So [1] gets the exponent part. For integral powers of 2 the exponent is one more than you might expect. For example 32 is stored as $0.5x2^6$. This explains the - 1 above. Also works for 1/32 which is stored as $0.5x2^{-4}$. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. $\log_2(1/17)$ is -5 not -4. int → int x.bit_length() If both input and output are integers, this native integer method could be very efficient: log2int_faster = x.bit_length() - 1 - 1 because 2^n requires n+1 bits. Works for very large integers, e.g. 2**10000. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. numpy.log2(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature]) = # Base-2 logarithm of x. Parameters: xarray_likeInput values. outndarray, None, or tuple of ndarray and None, optionalA location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs. wherearray_like, optionalThis condition is broadcast over the input. At locations where the condition is True, the out array will be set to the ufunc result. Elsewhere, the out array will retain its original value. Note that if an uninitialized out array is created via the default out=None, locations within it where the condition is False will remain uninitialized. **kwargsFor other keyword-only arguments, see the ufunc docs. Returns: yndarrayBase-2 logarithm of x. This is a scalar if x is a scalar. See also log, log10, log1p, emath.log2 Notes Logarithm is a multivalued function: for each x there is an infinite number of z such that 2**z = x. The convention is to return the z whose imaginary part lies in (-pi, pi]. For real-valued input data types, log2 always returns real output. For each value that cannot be expressed as a real number or infinity, it yields nan and sets the invalid floating point error flag. For complex-valued input, log2 is a complex analytical function that has a branch cut [-inf, 0] and is continuous from above on it. log2 handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard. In the cases where the input has a negative real part and a very small negative complex part (approaching 0), the result is so close to -pi that it evaluates to exactly -pi. Examples >>> import numpy as np >>> x = np.array([0, 1, 2, 2**4]) >>> np.log2(x) array([-inf, 0., 1., 4.]) >>> xi = np.array([0+1.j, 1, 2+0.j, 4.j]) >>> np.log2(xi) array([0.+2.26618007j, 0.+0.j , 1.+0.j , 2.+2.26618007j]) Depends on whether the input or output is int or float. assert 5.392317422778761 == math.log2(42.0) assert 5.392317422778761 == math.log(42.0, 2.0) assert 5 == math.frexp(42.0)[1] - 1 assert 5 == (42).bit_length() - 1 float → float math.log2(x) import math log2 = math.log(x, 2.0) log2 = math.log2(x) # python 3.3 or later Thanks @akashchandrakar and @unutbu. If all you need is the integer part of log base 2 of a floating point number, extracting the exponent is pretty efficient: log2int_slow = int(math.floor(math.log(x, 2.0))) # these give the log2int_fast = math.frexp(x)[1] - 1 # same result Python frexp() calls the C function frexp() which just grabs and tweaks the exponent. Python frexp() returns a tuple (mantissa, exponent). So [1] gets the exponent part. For integral powers of 2 the exponent is one more than you might expect. For example 32 is stored as $0.5x2^6$. This explains the - 1 above. Also works for 1/32 which is stored as $0.5x2^{-4}$. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. $\log_2(1/17)$ is -5 not -4. int → int x.bit_length() If both input and output are integers, this native integer method could be very efficient: log2int_faster = x.bit_length() - 1 - 1 because 2^n requires n+1 bits. Works for very large integers, e.g. 2**10000. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. Depends on whether the input or output is int or float. assert 5.392317422778761 == math.log2(42.0) assert 5.392317422778761 == math.log(42.0, 2.0) assert 5 == math.frexp(42.0)[1] - 1 assert 5 == (42).bit_length() - 1 float → float math.log2(x) import math log2 = math.log(x, 2.0) log2 = math.log2(x) # python 3.3 or later Thanks @akashchandrakar and @unutbu. If all you need is the integer part of log base 2 of a floating point number, extracting the exponent is pretty efficient: log2int_slow = int(math.floor(math.log(x, 2.0))) # these give the log2int_fast = math.frexp(x)[1] - 1 # same result Python frexp() calls the C function frexp() which just grabs and tweaks the exponent. Python frexp() returns a tuple (mantissa, exponent). So [1] gets the exponent part. For integral powers of 2 the exponent is one more than you might expect. For example 32 is stored as $0.5x2^6$. This explains the - 1 above. Also works for 1/32 which is stored as $0.5x2^{-4}$. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. $\log_2(1/17)$ is -5 not -4. int → int x.bit_length() If both input and output are integers, this native integer method could be very efficient: log2int_faster = x.bit_length() - 1 - 1 because 2^n requires n+1 bits. Works for very large integers, e.g. 2**10000. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. Python offers many inbuilt logarithmic functions under the module "math" which allows us to compute logs using a single line. There are 4 variants of logarithmic functions, all of which are discussed in this article.1. log(a,(Base)) : This function is used to compute the natural logarithm (Base e) of a. If 2 arguments are passed, it computes the logarithm of the desired base of argument a, numerically value of log(a)/log(Base). Syntax :math.log(a,Base)Parameters : a : The numeric valueBase : Base to which the logarithm has to be computed.Return Value : Returns natural log if 1 argument is passed and log withspecified base if 2 arguments are passed.Exceptions : Raises ValueError if a negative no. is passed as argument. Python3 # Python code to demonstrate the working of # log(a,Base) import math # Printing the log base e of 14 print ("Natural logarithm of 14 is : ", end="") print (math.log(14)) # Printing the log base 5 of 14 print ("Logarithm base 5 of 14 is : ", end="") print (math.log(14,5)) Output : Natural logarithm of 14 is : 2.6390573296152584Logarithm base 5 of 14 is : 1.63973851319556062. log2(a) : This function is used to compute the logarithm base 2 of a. Displays more accurate result than log(a,2).Syntax :math.log2(a)Parameters : a : The numeric valueReturn Value : Returns logarithm base 2 of aExceptions : Raises ValueError if a negative no. is passed as argument. Python3 # Python code to demonstrate the working of # log2(a) import math # Printing the log base 2 of 14 print ("Logarithm base 2 of 14 is : ", end="") print (math.log2(14)) Output : Logarithm base 2 of 14 is : 3.8073549220576043. log10(a) : This function is used to compute the logarithm base 10 of a. Displays more accurate result than log(a,10).Syntax :math.log10(a)Parameters : a : The numeric valueReturn Value : Returns logarithm base 10 of aExceptions : Raises ValueError if a negative no. is passed as argument. Python3 # Python code to demonstrate the working of # log10(a) import math # Printing the log base 10 of 14 print ("Logarithm base 10 of 14 is : ", end="") print (math.log10(14)) Output : Logarithm base 10 of 14 is : 1.1461280356782383. log1p(a) : This function is used to compute logarithm(1+a) . Syntax :math.log1p(a)Parameters : a : The numeric valueReturn Value : Returns log(1+a)Exceptions : Raises ValueError if a negative no. is passed as argument. Python3 # Python code to demonstrate the working of # log1p(a) import math # Printing the log(1+a) of 14 print ("Logarithm(1+a) value of 14 is : ", end="") print (math.log1p(14)) Output : Logarithm(1+a) value of 14 is : 2.70805020110221Exception1. ValueError : This function returns value error if number is negative. Python3 # Python code to demonstrate the Exception of # log(a) import math # Printing the log(a) of -14 # Throws Exception print ("log(a) value of -14 is : ", end="") print (math.log(-14)) Output : log(a) value of -14 is : Runtime Error : Traceback (most recent call last): File "/home/8a74e9d7e5adfdb902ab15712cbaafe2.py", line 9, in print (math.log(-14))ValueError: math domain errorPractical ApplicationOne of the application of log10() function is that it is used to compute the no. of digits of a number. Code below illustrates the same. Python3 # Python code to demonstrate the Application of # log10(a) import math # Printing no. of digits in 73293 print ("The number of digits in 73293 are : ", end="") print (int(math.log10(73293) + 1)) Output : The number of digits in 73293 are : 5The natural logarithm (log) is an important mathematical function in Python that is frequently used in scientific computing, data analysis, and machine learning applications. Here are some advantages, disadvantages, important points, and reference books related to log functions in Python:Advantages:The log function is useful for transforming data that has a wide range of values or a non-normal distribution into a more normally distributed form, which can improve the accuracy of statistical analyses and machine learning models.The log function is widely used in finance and economics to calculate compound interest, present values, and other financial metrics.The log function can be used to reduce the effect of outliers on statistical analyses by compressing the scale of the data.The log function can be used to visualize data with a large dynamic range or with values close to zero.Disadvantages:The log function can be computationally expensive for large datasets, especially if the log function is applied repeatedly.The log function may not be appropriate for all types of data, such as categorical data or data with a bounded range.Important points:The natural logarithm (log) is calculated using the numpy.log() function in Python.The logarithm with a base other than e can be calculated using the numpy.log10() or numpy.log2() functions in Python.The inverse of the natural logarithm is the exponential function, which can be calculated using the numpy.exp() function in Python.When using logarithms for statistical analyses or machine learning, it is important to remember to transform the data back to its original scale after analysis.Reference books:"Python for Data Analysis" by Wes McKinney covers the NumPy library and its applications in data analysis in depth, including the logarithmic function."Numerical Python: A Practical Techniques Approach for Industry" by Robert Johansson covers the NumPy library and its applications in numerical computing and scientific computing in depth, including the logarithmic function."Python Data Science Handbook" by Jake VanderPlas covers the NumPy library and its applications in data science in depth, including the logarithmic function. Depends on whether the input or output is int or float. assert 5.392317422778761 == math.log2(42.0) assert 5.392317422778761 == math.log(42.0, 2.0) assert 5 == math.frexp(42.0)[1] - 1 assert 5 == (42).bit_length() - 1 float → float math.log2(x) import math log2 = math.log(x, 2.0) log2 = math.log2(x) # python 3.3 or later Thanks @akashchandrakar and @unutbu. If all you need is the integer part of log base 2 of a floating point number, extracting the exponent is pretty efficient: log2int_slow = int(math.floor(math.log(x, 2.0))) # these give the log2int_fast = math.frexp(x)[1] - 1 # same result Python frexp() calls the C function frexp() which just grabs and tweaks the exponent. Python frexp() returns a tuple (mantissa, exponent). So [1] gets the exponent part. For integral powers of 2 the exponent is one more than you might expect. For example 32 is stored as $0.5x2^6$. This explains the - 1 above. Also works for 1/32 which is stored as $0.5x2^{-4}$. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. $\log_2(1/17)$ is -5 not -4. int → int x.bit_length() If both input and output are integers, this native integer method could be very efficient: log2int_faster = x.bit_length() - 1 - 1 because 2^n requires n+1 bits. Works for very large integers, e.g. 2**10000. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. Depends on whether the input or output is int or float. assert 5.392317422778761 == math.log2(42.0) assert 5.392317422778761 == math.log(42.0, 2.0) assert 5 == math.frexp(42.0)[1] - 1 assert 5 == (42).bit_length() - 1 float → float math.log2(x) import math log2 = math.log(x, 2.0) log2 = math.log2(x) # python 3.3 or later Thanks @akashchandrakar and @unutbu. If all you need is the integer part of log base 2 of a floating point number, extracting the exponent is pretty efficient: log2int_slow = int(math.floor(math.log(x, 2.0))) # these give the log2int_fast = math.frexp(x)[1] - 1 # same result Python frexp() calls the C function frexp() which just grabs and tweaks the exponent. Python frexp() returns a tuple (mantissa, exponent). So [1] gets the exponent part. For integral powers of 2 the exponent is one more than you might expect. For example 32 is stored as $0.5x2^6$. This explains the - 1 above. Also works for 1/32 which is stored as $0.5x2^{-4}$. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. $\log_2(1/17)$ is -5 not -4. int → int x.bit_length() If both input and output are integers, this native integer method could be very efficient: log2int_faster = x.bit_length() - 1 - 1 because 2^n requires n+1 bits. Works for very large integers, e.g. 2**10000. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. Depends on whether the input or output is int or float. assert 5.392317422778761 == math.log2(42.0) assert 5.392317422778761 == math.log(42.0, 2.0) assert 5 == math.frexp(42.0)[1] - 1 assert 5 == (42).bit_length() - 1 float → float math.log2(x) import math log2 = math.log(x, 2.0) log2 = math.log2(x) # python 3.3 or later Thanks @akashchandrakar and @unutbu. If all you need is the integer part of log base 2 of a floating point number, extracting the exponent is pretty efficient: log2int_slow = int(math.floor(math.log(x, 2.0))) # these give the log2int_fast = math.frexp(x)[1] - 1 # same result Python frexp() calls the C function frexp() which just grabs and tweaks the exponent. Python frexp() returns a tuple (mantissa, exponent). So [1] gets the exponent part. For integral powers of 2 the exponent is one more than you might expect. For example 32 is stored as $0.5x2^6$. This explains the - 1 above. Also works for 1/32 which is stored as $0.5x2^{-4}$. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. $\log_2(1/17)$ is -5 not -4. int → int x.bit_length() If both input and output are integers, this native integer method could be very efficient: log2int_faster = x.bit_length() - 1 - 1 because 2^n requires n+1 bits. Works for very large integers, e.g. 2**10000. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. Depends on whether the input or output is int or float. assert 5.392317422778761 == math.log2(42.0) assert 5.392317422778761 == math.log(42.0, 2.0) assert 5 == math.frexp(42.0)[1] - 1 assert 5 == (42).bit_length() - 1 float → float math.log2(x) import math log2 = math.log(x, 2.0) log2 = math.log2(x) # python 3.3 or later Thanks @akashchandrakar and @unutbu. If all you need is the integer part of log base 2 of a floating point number, extracting the exponent is pretty efficient: log2int_slow = int(math.floor(math.log(x, 2.0))) # these give the log2int_fast = math.frexp(x)[1] - 1 # same result Python frexp() calls the C function frexp() which just grabs and tweaks the exponent. Python frexp() returns a tuple (mantissa, exponent). So [1] gets the exponent part. For integral powers of 2 the exponent is one more than you might expect. For example 32 is stored as $0.5x2^6$. This explains the - 1 above. Also works for 1/32 which is stored as $0.5x2^{-4}$. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. $\log_2(1/17)$ is -5 not -4. int → int x.bit_length() If both input and output are integers, this native integer method could be very efficient: log2int_faster = x.bit_length() - 1 - 1 because 2^n requires n+1 bits. Works for very large integers, e.g. 2**10000. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. Depends on whether the input or output is int or float. assert 5.392317422778761 == math.log2(42.0) assert 5.392317422778761 == math.log(42.0, 2.0) assert 5 == math.frexp(42.0)[1] - 1 assert 5 == (42).bit_length() - 1 float → float math.log2(x) import math log2 = math.log(x, 2.0) log2 = math.log2(x) # python 3.3 or later Thanks @akashchandrakar and @unutbu. If all you need is the integer part of log base 2 of a floating point number, extracting the exponent is pretty efficient: log2int_slow = int(math.floor(math.log(x, 2.0))) # these give the log2int_fast = math.frexp(x)[1] - 1 # same result Python frexp() calls the C function frexp() which just grabs and tweaks the exponent. Python frexp() returns a tuple (mantissa, exponent). So [1] gets the exponent part. For integral powers of 2 the exponent is one more than you might expect. For example 32 is stored as $0.5x2^6$. This explains the - 1 above. Also works for 1/32 which is stored as $0.5x2^{-4}$. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5. $\log_2(1/17)$ is -5 not -4. int → int x.bit_length() If both input and output are integers, this native integer method could be very efficient: log2int_faster = x.bit_length() - 1 - 1 because 2^n requires n+1 bits. Works for very large integers, e.g. 2**10000. Floors toward negative infinity, so $\log_2 31$ computed this way is 4 not 5.