`math` — Mathematical functions

This module provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

Number-theoretic and representation functions

`math.`**`ceil`**(*x*)

Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__()`, which should return an `Integral` value.

`math.`**`comb`**(*n*, *k*)

Return the number of ways to choose *k* items from *n* items without repetition and without order.

Evaluates to `n! / (k! * (n - k)!)` when `k <= n` and evaluates to zero when `k > n`.

Also called the binomial coefficient because it is equivalent to the coefficient of k-th term in polynomial expansion of the expression `(1 + x) ** n`.

Raises `TypeError` if either of the arguments are not integers. Raises `ValueError` if either of the arguments are negative.

New in version 3.8.

`math.`**`copysign`**(*x*, *y*)

Return a float with the magnitude (absolute value) of *x* but the sign of *y*. On platforms that support signed zeros, `copysign(1.0, -0.0)` returns *-1.0*.

`math.`**`fabs`**(*x*)

Return the absolute value of *x*.

`math.`**`factorial`**(*x*)

Return *x* factorial as an integer. Raises `ValueError` if *x* is not integral or is negative.

`math.`**`floor`**(*x*)

Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.__floor__()`, which should return an `Integral` value.

`math.`**`fmod`**(*x*, *y*)

Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer *n* such that the result has the same sign as *x* and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising `1e100`. For this reason, function `fmod()` is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

`math.`**`frexp`**(*x*)

Return the mantissa and exponent of *x* as the pair `(m, e)`. *m* is a float and *e* is an integer such that `x == m * 2**e` exactly. If *x* is zero, returns `(0.0, 0)`, otherwise `0.5 <= abs(m) < 1`. This is used to "pick apart" the internal representation of a float in a portable way.

`math.`**`fsum`**(*iterable*)

Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

The algorithm's accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.

For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating point summation.

`math.`**`gcd`**(*a*, *b*)

Return the greatest common divisor of the integers *a* and *b*. If either *a* or *b* is nonzero, then the value of `gcd(a, b)` is the largest positive integer that divides both *a* and *b*. `gcd(0, 0)` returns `0`.

New in version 3.5.

`math.`**`isclose`**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)

Return `True` if the values *a* and *b* are close to each other and `False` otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances.

rel_tol is the relative tolerance – it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *rel_tol* must be greater than zero.

abs_tol is the minimum absolute tolerance – useful for comparisons near zero. *abs_tol* must be at least zero.

If no errors occur, the result will be: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

The IEEE 754 special values of `NaN`, `inf`, and `-inf` will be handled according to IEEE rules. Specifically, `NaN` is not considered close to any other value, including `NaN`. `inf` and `-inf` are only considered close to themselves.

New in version 3.5.

See also: **PEP 485** – A function for testing approximate equality

`math.`**`isfinite`**(*x*)

Return `True` if *x* is neither an infinity nor a NaN, and `False` otherwise. (Note that `0.0` *is* considered finite.)

New in version 3.2.

`math.`**`isinf`**(*x*)

Return `True` if *x* is a positive or negative infinity, and `False` otherwise.

`math.`**`isnan`**(*x*)

Return `True` if *x* is a NaN (not a number), and `False` otherwise.

`math.`**`isqrt`**(*n*)

Return the integer square root of the nonnegative integer *n*. This is the floor of the exact square root of *n*, or equivalently the greatest integer *a* such that $a^2 \le n$.

For some applications, it may be more convenient to have the least integer *a* such that $n \le a^2$, or in other words the ceiling of the exact square root of *n*. For positive *n*, this can be computed using `a = 1 + isqrt(n - 1)`.

New in version 3.8.

`math.`**`ldexp`**(*x*, *i*)

Return `x * (2**i)`. This is essentially the inverse of function `frexp()`.

`math.`**`modf`**(*x*)

Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.

`math.`**`perm`**(*n*, *k=None*)

Return the number of ways to choose *k* items from *n* items without repetition and with order.

Evaluates to `n! / (n - k)!` when `k <= n` and evaluates to zero when `k > n`.

If *k* is not specified or is None, then *k* defaults to *n* and the function returns `n!`.

Raises `TypeError` if either of the arguments are not integers. Raises `ValueError` if either of the arguments are negative.

New in version 3.8.

`math.`**`prod`**`(`*`iterable`*`, *, `*`start=1`*`)`

Calculate the product of all the elements in the input *iterable*. The default *start* value for the product is `1`.

When the iterable is empty, return the start value. This function is intended specifically for use with numeric values and may reject non-numeric types.

New in version 3.8.

`math.`**`remainder`**`(`*x*, *y*`)`

Return the IEEE 754-style remainder of *x* with respect to *y*. For finite *x* and finite nonzero *y*, this is the difference `x - n*y`, where `n` is the closest integer to the exact value of the quotient `x / y`. If `x / y` is exactly halfway between two consecutive integers, the nearest *even* integer is used for `n`. The remainder `r = remainder(x, y)` thus always satisfies `abs(r) <= 0.5 * abs(y)`.

Special cases follow IEEE 754: in particular, `remainder(x, math.inf)` is *x* for any finite *x*, and `remainder(x, 0)` and `remainder(math.inf, x)` raise `ValueError` for any non-NaN *x*. If the result of the remainder operation is zero, that zero will have the same sign as *x*.

On platforms using IEEE 754 binary floating-point, the result of this operation is always exactly representable: no rounding error is introduced.

New in version 3.7.

`math.`**`trunc`**`(`*x*`)`

Return the `Real` value *x* truncated to an `Integral` (usually an integer). Delegates to `x.__trunc__()`.

Note that `frexp()` and `modf()` have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an ‘output parameter’ (there is no such thing in Python).

For the `ceil()`, `floor()`, and `modf()` functions, note that *all* floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float *x* with `abs(x) >= 2**52` necessarily has no fractional bits.

Power and logarithmic functions

`math.`**`exp`**(*x*)

Return *e* raised to the power *x*, where *e* = 2.718281… is the base of natural logarithms. This is usually more accurate than `math.e ** x` or `pow(math.e, x)`.

`math.`**`expm1`**(*x*)

Return *e* raised to the power *x*, minus 1. Here *e* is the base of natural logarithms. For small floats *x*, the subtraction in `exp(x) - 1` can result in a significant loss of precision; the `expm1()` function provides a way to compute this quantity to full precision:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

New in version 3.2.

`math.`**`log`**(*x*[, *base*])

With one argument, return the natural logarithm of *x* (to base *e*).

With two arguments, return the logarithm of *x* to the given *base*, calculated as `log(x)/log(base)`.

`math.`**`log1p`**(*x*)

Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.

`math.`**`log2`**(*x*)

Return the base-2 logarithm of *x*. This is usually more accurate than `log(x, 2)`.

New in version 3.3.

See also: `int.bit_length()` returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

`math.`**`log10`**(*x*)

Return the base-10 logarithm of *x*. This is usually more accurate than `log(x, 10)`.

`math.`**`pow`**(*x*, *y*)

Return `x` raised to the power `y`. Exceptional cases follow Annex 'F' of the C99 standard as far as possible. In particular, `pow(1.0, x)` and `pow(x, 0.0)` always return `1.0`, even when `x` is a zero or a NaN. If both `x` and `y` are finite, `x` is negative, and `y` is not an integer then `pow(x, y)` is undefined, and raises `ValueError`.

Unlike the built-in `**` operator, `math.pow()` converts both its arguments to type `float`. Use `**` or the built-in `pow()` function for computing exact integer powers.

`math.`**`sqrt`**(*x*)
Return the square root of *x*.

Trigonometric functions

`math.`**`acos`**(*x*)
Return the arc cosine of *x*, in radians.

`math.`**`asin`**(*x*)
Return the arc sine of *x*, in radians.

`math.`**`atan`**(*x*)
Return the arc tangent of *x*, in radians.

`math.`**`atan2`**(*y*, *x*)
Return `atan(y / x)`, in radians. The result is between `-pi` and `pi`. The vector in the plane from the origin to point `(x, y)` makes this angle with the positive X axis. The point of `atan2()` is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, `atan(1)` and `atan2(1, 1)` are both `pi/4`, but `atan2(-1, -1)` is `-3*pi/4`.

`math.`**`cos`**(*x*)
Return the cosine of *x* radians.

`math.`**`dist`**(*p*, *q*)
Return the Euclidean distance between two points *p* and *q*, each given as a sequence (or iterable) of coordinates. The two points must have the same dimension.

Roughly equivalent to:

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

New in version 3.8.

`math.`**`hypot`**(**coordinates*)
Return the Euclidean norm, `sqrt(sum(x**2 for x in coordinates))`. This is the length of the vector from the origin to the point given by the coordinates.

For a two dimensional point `(x, y)`, this is equivalent to computing the hypotenuse of a right triangle using the Pythagorean theorem, `sqrt(x*x + y*y)`.

Changed in version 3.8: Added support for n-dimensional points. Formerly, only the two dimensional case was supported.

`math.`**`sin`**(*x*)
Return the sine of *x* radians.

`math.`**`tan`**(*x*)

Return the tangent of *x* radians.

Angular conversion

`math.`**`degrees`**(*x*)
Convert angle *x* from radians to degrees.

`math.`**`radians`**(*x*)
Convert angle *x* from degrees to radians.

Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

`math.`**`acosh`**(*x*)
Return the inverse hyperbolic cosine of *x*.

`math.`**`asinh`**(*x*)
Return the inverse hyperbolic sine of *x*.

`math.`**`atanh`**(*x*)
Return the inverse hyperbolic tangent of *x*.

`math.`**`cosh`**(*x*)
Return the hyperbolic cosine of *x*.

`math.`**`sinh`**(*x*)
Return the hyperbolic sine of *x*.

`math.`**`tanh`**(*x*)
Return the hyperbolic tangent of *x*.

Special functions

`math.`**`erf`**(*x*)
Return the error function at *x*.

The `erf()` function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

New in version 3.2.

math.**erfc**(*x*)

Return the complementary error function at *x*. The complementary error function is defined as `1.0 - erf(x)`. It is used for large values of *x* where a subtraction from one would cause a loss of significance.

New in version 3.2.

math.**gamma**(*x*)

Return the Gamma function at *x*.

New in version 3.2.

math.**lgamma**(*x*)

Return the natural logarithm of the absolute value of the Gamma function at *x*.

New in version 3.2.

Constants ¶

math.**pi**

The mathematical constant *π* = 3.141592…, to available precision.

math.**e**

The mathematical constant *e* = 2.718281…, to available precision.

math.**tau**

The mathematical constant *τ* = 6.283185…, to available precision. Tau is a circle constant equal to 2*π*, the ratio of a circle's circumference to its radius. To learn more about Tau, check out Vi Hart's video Pi is (still) Wrong, and start celebrating Tau day by eating twice as much pie!

New in version 3.6.

math.**inf**

A floating-point positive infinity. (For negative infinity, use `-math.inf`.) Equivalent to the output of `float('inf')`.

New in version 3.5.

math.**nan**

A floating-point "not a number" (NaN) value. Equivalent to the output of `float('nan')`.

New in version 3.5.

CPython implementation detail: The `math` module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard where appropriate. The current implementation will raise `ValueError` for invalid operations like `sqrt(-1.0)` or `log(0.0)` (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and `OverflowError` for results that overflow (for example,

`exp(1000.0)`). A NaN will not be returned from any of the functions above unless one or more of the input arguments was a NaN; in that case, most functions will return a NaN, but (again following C99 Annex F) there are some exceptions to this rule, for example `pow(float('nan'), 0.0)` or `hypot(float('nan'), float('inf'))`.

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

See also:

Module `cmath`
: Complex number versions of many of these functions.