10.2. `math` — Mathematical functions¶

This module is always available. It 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.

10.2.1. Number-theoretic and representation functions¶

`math.ceil`(*x*)¶
Return the ceiling of *x* as a float, the smallest integer value greater than or equal to *x*.

`math.copysign`(*x*, *y*)¶

Return *x* with the sign of *y*. `copysign` copies the sign bit of an IEEE 754 float, `copysign(1, -0.0)` returns *-1.0*.

New in version 2.6.

`math.fabs`(*x*)¶
Return the absolute value of *x*.

`math.factorial`(*x*)¶

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

New in version 2.6.

`math.floor`(*x*)¶

Return the floor of *x* as a float, the largest integer value less than or equal to *x*.

Changed in version 2.6: Added `__floor__()` delegation.

`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])
0.99999999999999989
>>> 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.

New in version 2.6.

`math.isinf(x)`¶

Checks if the float *x* is positive or negative infinite.

New in version 2.6.

`math.isnan(x)`¶

Checks if the float *x* is a NaN (not a number). NaNs are part of the IEEE 754 standards. Operation like but not limited to `inf * 0`, `inf / inf` or any operation involving a NaN, e.g. `nan * 1`, return a NaN.

New in version 2.6.

`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.trunc(x)`¶

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

New in version 2.6.

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.

10.2.2. Power and logarithmic functions¶

`math.exp(x)`¶
Return `e**x`.

`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)`.

Changed in version 2.3: *base* argument added.

`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.

New in version 2.6.

`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`.

Changed in version 2.6: The outcome of `1**nan` and `nan**0` was undefined.

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

10.2.3. 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.hypot`(*x*, *y*)¶
Return the Euclidean norm, `sqrt(x*x + y*y)`. This is the length of the vector from the origin to point `(x, y)`.

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

`math.tan`(*x*)¶
Return the tangent of *x* radians.

10.2.4. Angular conversion¶

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

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

10.2.5. Hyperbolic functions¶

`math.acosh`(*x*)¶

Return the inverse hyperbolic cosine of *x*.

New in version 2.6.

`math.asinh`(*x*)¶

Return the inverse hyperbolic sine of *x*.

New in version 2.6.

`math.atanh`(*x*)¶

Return the inverse hyperbolic tangent of *x*.

New in version 2.6.

`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*.

10.2.6. Constants¶

`math.pi`¶
The mathematical constant *pi*.

`math.e`¶
The mathematical constant *e*.

CPython implementation detail: The `math` module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases is loosely specified by the C standards, and Python inherits much of its math-function error-reporting behavior from the platform C implementation. As a result, the specific exceptions raised in error cases (and even whether some arguments are considered to be exceptional at all) are not defined in any useful cross-platform or cross-release way. For example, whether `math.log(0)` returns `-Inf` or raises `ValueError` or `OverflowError` isn't defined, and in cases where `math.log(0)` raises `OverflowError`, `math.log(0L)` may raise `ValueError` instead.

All functions return a quiet *NaN* if at least one of the args is *NaN*. Signaling *NaN*s raise an exception. The exception type still depends on the platform and libm implementation. It's usually `ValueError` for *EDOM* and `OverflowError` for errno *ERANGE*.

Changed in version 2.6: In earlier versions of Python the outcome of an operation with NaN as input depended on platform and libm implementation.

See also

Module `cmath`

Complex number versions of many of these functions.

Table Of Contents

Previous topic

10.1. `numbers` — Numeric abstract base classes

Next topic

10.3. `cmath` — Mathematical functions for complex numbers

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 10. Numeric and Mathematical Modules »