Math Module All Versions

Examples

Rounding: round, floor, ceil, trunc

In addition to the built-in `round` function, the `math` module provides the `floor` , `ceil` , and `trunc` functions.

```
x = 1.55
y = -1.55

# round to the nearest integer
round(x)       #  2
round(y)       # -2

# the second argument gives how many decimal places to round to (defaults to 0)
round(x, 1)    #  1.6
round(y, 1)    # -1.6

# math is a module so import it first, then use it.
import math

# get the largest integer less than x
math.floor(x)  #  1
math.floor(y)  # -2

# get the smallest integer greater than x
math.ceil(x)   #  2
math.ceil(y)   # -1

# drop fractional part of x
math.trunc(x)  #  1, equivalent to math.floor for positive numbers
math.trunc(y)  # -1, equivalent to math.ceil for negative numbers
```

Python 2.x ≤2.7

`floor` , `ceil` , `trunc` , and `round` always return a `float` .

```
round(1.3)  # 1.0
```

`round` always breaks ties away from zero.

```
round(0.5)  # 1.0
round(1.5)  # 2.0
```

Python 3.x ≥3.0

`floor` , `ceil` , and `trunc` always return an `Integral` value, while `round` returns an `Integral` value if called with one argument.

```
round(1.3)      # 1
round(1.33, 1)  # 1.3
```

`round` breaks ties towards the nearest even number. This corrects the bias towards larger numbers when performing a large number of calculations.

```
round(0.5)  # 0
round(1.5)  # 2
```

Warning!

As with any floating-point representation, some fractions *cannot be represented exactly* . This can lead to some unexpected rounding behavior.

```
round(2.675, 2)  # 2.67, not 2.68!
```

Warning about the floor, trunc, and integer division of negative numbers

Python (and C++ and Java) round away from zero for negative numbers. Consider:

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

Trigonometry

Calculating the length of the hypotenuse

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)
# Out: 4.47213595499958
```

Converting degrees to/from radians

All `math` functions expect **radians** so you need to convert degrees to radians:

```
math.radians(45)             # Convert 45 degrees to radians
# Out: 0.7853981633974483
```

All results of the inverse trigonometic functions return the result in radians, so you may need to convert it back to degrees:

```
math.degrees(math.asin(1))    # Convert the result of asin to degrees
# Out: 90.0
```

Sine, cosine, tangent and inverse functions

```
# Sine and arc sine
math.sin(math.pi / 2)
# Out: 1.0
math.sin(math.radians(90))   # Sine of 90 degrees
# Out: 1.0

math.asin(1)
# Out: 1.5707963267948966    # "= pi / 2"
math.asin(1) / math.pi
# Out: 0.5

# Cosine and arc cosine:
math.cos(math.pi / 2)
# Out: 6.123233995736766e-17
# Almost zero but not exactly because "pi" is a float with limited precision!

math.acos(1)
# Out: 0.0

# Tangent and arc tangent:
math.tan(math.pi/2)
# Out: 1.633123935319537e+16
# Very large but not exactly "Inf" because "pi" is a float with limited precision
```

Python 3.x ≥3.5

```
math.atan(math.inf)
# Out: 1.5707963267948966 # This is just "pi / 2"
```

```
math.atan(float('inf'))
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Apart from the `math.atan` there is also a two-argument `math.atan2` function, which computes the correct quadrant and avoids pitfalls of division by zero:

```
math.atan2(1, 2)   # Equivalent to "math.atan(1/2)"
# Out: 0.4636476090008061 # ≈ 26.57 degrees, 1st quadrant

math.atan2(-1, -2) # Not equal to "math.atan(-1/-2)" == "math.atan(1/2)"
# Out: -2.677945044588987 # ≈ -153.43 degrees (or 206.57 degrees), 3rd quadrant

math.atan2(1, 0)   # math.atan(1/0) would raise ZeroDivisionError
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Hyperbolic sine, cosine and tangent

```
# Hyperbolic sine function
math.sinh(math.pi) # = 11.548739357257746
math.asinh(1)      # = 0.8813735870195429

# Hyperbolic cosine function
math.cosh(math.pi) # = 11.591953275521519
math.acosh(1)      # = 0.0

# Hyperbolic tangent function
math.tanh(math.pi) # = 0.99627207622075
math.atanh(0.5)    # = 0.5493061443340549
```

Constants

`math` modules includes two commonly used mathematical constants.

- `math.pi` - The mathematical constant pi
- `math.e` - The mathematical constant *e* (base of natural logarithm)

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>>
```

Python 3.5 and higher have constants for infinity and NaN ("not a number"). The older syntax of passing a string to `float()` still works.

Python 3.x ≥3.5

```
math.inf == float('inf')
# Out: True

-math.inf == float('-inf')
# Out: True

# NaN never compares equal to anything, even itself
math.nan == float('nan')
# Out: False
```

Infinity and NaN ("not a number")

In all versions of Python, we can represent infinity and NaN ("not a number") as follows:

```
pos_inf = float('inf')     # positive infinity
neg_inf = float('-inf')    # negative infinity
not_a_num = float('nan')   # NaN ("not a number")
```

In Python 3.5 and higher, we can also use the defined constants `math.inf` and `math.nan` :

Python 3.x ≥3.5

```
pos_inf = math.inf
neg_inf = -math.inf
not_a_num = math.nan
```

The string representations display as `inf` and `-inf` and `nan` :

```
pos_inf, neg_inf, not_a_num
# Out: (inf, -inf, nan)
```

We can test for either positive or negative infinity with the `isinf` method:

```
math.isinf(pos_inf)
# Out: True

math.isinf(neg_inf)
# Out: True
```

We can test specifically for positive infinity or for negative infinity by direct comparison:

```
pos_inf == float('inf')    # or  == math.inf in Python 3.5+
# Out: True

neg_inf == float('-inf')   # or  == -math.inf in Python 3.5+
# Out: True

neg_inf == pos_inf
# Out: False
```

Python 3.2 and higher also allows checking for finiteness:

Python 3.x ≥3.2

```
math.isfinite(pos_inf)
# Out: False

math.isfinite(0.0)
# Out: True
```

```
# Out: True
```

Comparison operators work as expected for positive and negative infinity:

```
import sys

sys.float_info.max
# Out: 1.7976931348623157e+308  (this is system-dependent)

pos_inf > sys.float_info.max
# Out: True

neg_inf < -sys.float_info.max
# Out: True
```

But if an arithmetic expression produces a value larger than the maximum that can be represented as a `float`, it will become infinity:

```
pos_inf == sys.float_info.max * 1.0000001
# Out: True

neg_inf == -sys.float_info.max * 1.0000001
# Out: True
```

However division by zero does not give a result of infinity (or negative infinity where appropriate), rather it raises a `ZeroDivisionError` exception.

```
try:
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Division by zero")

# Out: Division by zero
```

Arithmetic operations on infinity just give infinite results, or sometimes NaN:

```
-5.0 * pos_inf == neg_inf
# Out: True

-5.0 * neg_inf == pos_inf
# Out: True

pos_inf * neg_inf == neg_inf
# Out: True

0.0 * pos_inf
# Out: nan

0.0 * neg_inf
# Out: nan

pos_inf / pos_inf
# Out: nan
```

NaN is never equal to anything, not even itself. We can test for it is with the `isnan` method:

```
not_a_num == not_a_num
# Out: False

math.isnan(not_a_num)
Out: True
```

NaN always compares as "not equal", but never less than or greater than:

```
not_a_num != 5.0   # or any random value
# Out: True

not_a_num > 5.0   or   not_a_num < 5.0   or   not_a_num == 5.0
# Out: False
```

Arithmetic operations on NaN always give NaN. This includes multiplication by -1: there is no "negative NaN".

```
5.0 * not_a_num
# Out: nan

float('-nan')
# Out: nan
```

Python 3.x ≥3.5

```
-math.nan
# Out: nan
```

There is one subtle difference between the old `float` versions of NaN and infinity and the Python 3.5+ `math` library constants:

Python 3.x ≥3.5

```
math.inf is math.inf, math.nan is math.nan
# Out: (True, True)

float('inf') is float('inf'), float('nan') is float('nan')
# Out: (False, False)
```

Pow for faster exponentiation

Using the timeit module from the command line:

```
> python -m timeit 'for x in xrange(50000): b = x**3'
10 loops, best of 3: 51.2 msec per loop
> python -m timeit 'from math import pow' 'for x in xrange(50000): b = pow(x,3)'
100 loops, best of 3: 9.15 msec per loop
```

The built-in `**` operator often comes in handy, but if performance is of the essence, use math.pow. Be sure to note, however, that pow returns floats, even if the arguments are integers:

```
> from math import pow
> pow(5,5)
3125.0
```

Copying signs

In Python 2.6 and higher, `math.copysign(x, y)` returns `x` with the sign of `y` . The returned value is always a `float` .

Python 2.x ≥2.6

```
math.copysign(-2, 3)    # 2.0
math.copysign(3, -3)    # -3.0
math.copysign(4, 14.2)  # 4.0
math.copysign(1, -0.0)  # -1.0, on a platform which supports signed zero
```

Imaginary Numbers

Imaginary numbers in Python are represented by a "j" or "J" trailing the target number.

```
1j         # Equivalent to the square root of -1.
1j * 1j    # = (-1+0j)
```

Logarithms

`math.log(x)` gives the natural (base `e`) logarithm of `x` .

```
math.log(math.e)  # 1.0
math.log(1)       # 0.0
math.log(100)     # 4.605170185988092
```

`math.log` can lose precision with numbers close to 1, due to the limitations of floating-point numbers. In order to accurately calculate logs close to 1, use `math.log1p` , which evaluates the natural logarithm of 1 plus the argument:

```
math.log(1 + 1e-20)  # 0.0
math.log1p(1e-20)    # 1e-20
```

`math.log10` can be used for logs base 10:

```
math.log10(10)  # 1.0
```

Python 2.x ≥2.3.0

When used with two arguments, `math.log(x, base)` gives the logarithm of `x` in the given `base` (i.e. `log(x) / log(base)`).

```
math.log(100, 10) # 2.0
math.log(27, 3)   # 3.0
math.log(1, 10)   # 0.0
```

Complex numbers and the cmath module

The `cmath` module is similar to the `math` module, but defines functions appropriately for the complex plane.

First of all, complex numbers are a numeric type that is part of the Python language itself rather than being provided by a library class. Thus we don't need to `import cmath` for ordinary arithmetic expressions.

Note that we use `j` (or `J`) and not `i`.

```
z = 1 + 3j
```

We must use `1j` since `j` would be the name of a variable rather than a numeric literal.

```
1j * 1j
Out: (-1+0j)

1j ** 1j
# Out: (0.20787957635076193+0j)     # "i to the i"  ==  math.e ** -(math.pi/2)
```

We have the `real` part and the `imag` (imaginary) part, as well as the complex `conjugate`:

```
# real part and imaginary part are both float type
z.real, z.imag
# Out: (1.0, 3.0)

z.conjugate()
# Out: (1-3j)    # z.conjugate() == z.real - z.imag * 1j
```

The built-in functions `abs` and `complex` are also part of the language itself and don't require any import:

```
abs(1 + 1j)
# Out: 1.4142135623730951     # square root of 2

complex(1)
# Out: (1+0j)

complex(imag=1)
# Out: (1j)

complex(1, 1)
# Out: (1+1j)
```

The `complex` function can take a string, but it can't have spaces:

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
# Exception: ValueError: complex() arg is a malformed string
```

But for most functions we do need the module, for instance `sqrt`:

```
import cmath

cmath.sqrt(-1)
# Out: 1j
```

Naturally the behavior of `sqrt` is different for complex numbers and real numbers. In non-complex `math` the square root of a negative number raises an exception:

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

Functions are provided to convert to and from polar coordinates:

```
cmath.polar(1 + 1j)
```

```
# Out: (1.4142135623730951, 0.7853981633974483)    # == (sqrt(1 + 1), atan2(1, 1))

abs(1 + 1j), cmath.phase(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # same as previous calculation

cmath.rect(math.sqrt(2), math.atan(1))
# Out: (1.0000000000000002+1.0000000000000002j)
```

The mathematical field of complex analysis is beyond the scope of this example, but many functions in the complex plane have a "branch cut", usually along the real axis or the imaginary axis. Most modern platforms support "signed zero" as specified in IEEE 754, which provides continuity of those functions on both sides of the branch cut. The following example is from the Python documentation:

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
# Out: -3.141592653589793
```

The `cmath` module also provides many functions with direct counterparts from the `math` module.

In addition to `sqrt`, there are complex versions of `exp`, `log`, `log10`, the trigonometric functions and their inverses (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`), and the hyperbolic functions and their inverses (`sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`). Note however there is no complex counterpart of `math.atan2`, the two-argument form of arctangent.

```
cmath.log(1+1j)
# Out: (0.34657359027997264+0.7853981633974483j)

cmath.exp(1j * cmath.pi)
# Out: (-1+1.2246467991473532e-16j)   # e to the i pi == -1, within rounding error
```

The constants `pi` and `e` are provided. Note these are `float` and not `complex`.

```
type(cmath.pi)
# Out: <class 'float'>
```

The `cmath` module also provides complex versions of `isinf`, and (for Python 3.2+) `isfinite`. See " Infinity and NaN ". A complex number is considered infinite if either its real part or its imaginary part is infinite.

```
cmath.isinf(complex(float('inf'), 0.0))
# Out: True
```

Likewise, the `cmath` module provides a complex version of `isnan`. See " Infinity and NaN ". A complex number is considered "not a number" if either its real part or its imaginary part is "not a number".

```
cmath.isnan(0.0, float('nan'))
# Out: True
```

Note there is no `cmath` counterpart of the `math.inf` and `math.nan` constants (from Python 3.5 and higher)

Python 3.x ≥3.5

```
cmath.isinf(complex(0.0, math.inf))
# Out: True

cmath.isnan(complex(math.nan, 0.0))
# Out: True

cmath.inf
# Exception: AttributeError: module 'cmath' has no attribute 'inf'
```

In Python 3.5 and higher, there is an `isclose` method in both `cmath` and `math` modules.

Python 3.x ≥3.5

```
z = cmath.rect(*cmath.polar(1+1j))

z
# Out: (1.0000000000000002+1.0000000000000002j)

cmath.isclose(z, 1+1j)
# True
```

Syntax

Parameters

Remarks