math — Mathematical Functions

Purpose: Provides functions for specialized mathematical operations.

The `math` module implements many of the IEEE functions that would normally be found in the native platform C libraries for complex mathematical operations using floating point values, including logarithms and trigonometric operations.

Special Constants

Many math operations depend on special constants. `math` includes values for π (pi), e, nan (not a number), and infinity.

```
# math_constants.py

import math

print('  π: {:.30f}'.format(math.pi))
print('  e: {:.30f}'.format(math.e))
print('nan: {:.30f}'.format(math.nan))
print('inf: {:.30f}'.format(math.inf))
```

Both π and e are limited in precision only by the platform’s floating point C library.

```
$ python3 math_constants.py

  π: 3.141592653589793115997963468544
  e: 2.718281828459045090795598298428
nan: nan
inf: inf
```

Testing for Exceptional Values

Floating point calculations can result in two types of exceptional values. The first of these, `inf` (infinity), appears when the `double` used to hold a floating point value overflows from a value with a large absolute value.

```
# math_isinf.py

import math

print('{:^3} {:6} {:6} {:6}'.format(
    'e', 'x', 'x**2', 'isinf'))
print('{:-^3} {:-^6} {:-^6} {:-^6}'.format(
    '', '', '', ''))

for e in range(0, 201, 20):
    x = 10.0 ** e
    y = x * x
    print('{:3d} {:<6g} {:<6g} {!s:6}'.format(
        e, x, y, math.isinf(y),
    ))
```

When the exponent in this example grows large enough, the square of x no longer fits inside a `double`, and the value is recorded as infinite.

```
$ python3 math_isinf.py

 e  x      x**2   isinf
--- ------ ------ ------
  0 1      1      False
 20 1e+20  1e+40  False
 40 1e+40  1e+80  False
 60 1e+60  1e+120 False
```

```
 80 1e+80  1e+160 False
100 1e+100 1e+200 False
120 1e+120 1e+240 False
140 1e+140 1e+280 False
160 1e+160 inf    True
180 1e+180 inf    True
200 1e+200 inf    True
```

Not all floating point overflows result in `inf` values, however. Calculating an exponent with floating point values, in particular, raises `OverflowError` instead of preserving the `inf` result.

```
# math_overflow.py

x = 10.0 ** 200

print('x    =', x)
print('x*x  =', x * x)
print('x**2 =', end=' ')
try:
    print(x ** 2)
except OverflowError as err:
    print(err)
```

This discrepancy is caused by an implementation difference in the library used by C Python.

```
$ python3 math_overflow.py

x    = 1e+200
x*x  = inf
x**2 = (34, 'Result too large')
```

Division operations using infinite values are undefined. The result of dividing a number by infinity is nan (not a number).

```
# math_isnan.py

import math

x = (10.0 ** 200) * (10.0 ** 200)
y = x / x

print('x =', x)
print('isnan(x) =', math.isnan(x))
print('y = x / x =', x / x)
print('y == nan =', y == float('nan'))
print('isnan(y) =', math.isnan(y))
```

nan does not compare as equal to any value, even itself, so to check for nan use `isnan()`.

```
$ python3 math_isnan.py

x = inf
isnan(x) = False
y = x / x = nan
y == nan = False
isnan(y) = True
```

Use `isfinite()` to check for regular numbers or either of the special values `inf` or nan.

```
# math_isfinite.py

import math

for f in [0.0, 1.0, math.pi, math.e, math.inf, math.nan]:
    print('{:5.2f} {!s}'.format(f, math.isfinite(f)))
```

`isfinite()` returns false for either of the exceptional cases, and true otherwise.

```
$ python3 math_isfinite.py
```

```
 0.00 True
 1.00 True
 3.14 True
 2.72 True
  inf False
  nan False
```

Comparing

Comparisons for floating point values can be error prone, with each step of the computation potentially introducing errors due to the numerical representation. The `isclose()` function uses a stable algorithm to minimize these errors and provide a way for relative as well as absolute comparisons. The formula used is equivalent to

```
abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)
```

By default, `isclose()` uses relative comparison with the tolerance set to `1e-09`, meaning that the difference between the values must be less than or equal to `1e-09` times the larger absolute value between a and b. Passing a keyword argument `rel_tol` to `isclose()` changes the tolerance. In this example, the values must be within 10% of each other.

```
# math_isclose.py
```

```
import math

INPUTS = [
    (1000, 900, 0.1),
    (100, 90, 0.1),
    (10, 9, 0.1),
    (1, 0.9, 0.1),
    (0.1, 0.09, 0.1),
]

print('{:^8} {:^8} {:^8} {:^8} {:^8} {:^8}'.format(
    'a', 'b', 'rel_tol', 'abs(a-b)', 'tolerance', 'close')
)
print('{:-^8} {:-^8} {:-^8} {:-^8} {:-^8} {:-^8}'.format(
    '-', '-', '-', '-', '-', '-'),
)

fmt = '{:8.2f} {:8.2f} {:8.2f} {:8.2f} {:8.2f} {!s:>8}'

for a, b, rel_tol in INPUTS:
    close = math.isclose(a, b, rel_tol=rel_tol)
    tolerance = rel_tol * max(abs(a), abs(b))
    abs_diff = abs(a - b)
    print(fmt.format(a, b, rel_tol, abs_diff, tolerance, close))
```

The comparison between `0.1` and `0.09` fails because of the error representing `0.1`.

```
$ python3 math_isclose.py

   a        b     rel_tol  abs(a-b) tolerance  close
-------- -------- -------- -------- -------- --------
 1000.00   900.00     0.10   100.00   100.00     True
  100.00    90.00     0.10    10.00    10.00     True
   10.00     9.00     0.10     1.00     1.00     True
    1.00     0.90     0.10     0.10     0.10     True
    0.10     0.09     0.10     0.01     0.01    False
```

To use a fixed or “absolute” tolerance, pass `abs_tol` instead of `rel_tol`.

```
# math_isclose_abs_tol.py
```

```
import math

INPUTS = [
    (1.0, 1.0 + 1e-07, 1e-08),
    (1.0, 1.0 + 1e-08, 1e-08),
```

```
    (1.0, 1.0 + 1e-09, 1e-08),
]

print('{:^8} {:^11} {:^8} {:^10} {:^8}'.format(
    'a', 'b', 'abs_tol', 'abs(a-b)', 'close')
)
print('{:-^8} {:-^11} {:-^8} {:-^10} {:-^8}'.format(
    '-', '-', '-', '-', '-'),
)

for a, b, abs_tol in INPUTS:
    close = math.isclose(a, b, abs_tol=abs_tol)
    abs_diff = abs(a - b)
    print('{:8.2f} {:11} {:8} {:0.9f} {!s:>8}'.format(
        a, b, abs_tol, abs_diff, close))
```

For an absolute tolerance, the difference between the input values must be less than the tolerance given.

```
$ python3 math_isclose_abs_tol.py

   a          b       abs_tol    abs(a-b)    close
-------- ----------- -------- ---------- --------
    1.00   1.0000001    1e-08 0.000000100    False
    1.00  1.00000001    1e-08 0.000000010     True
    1.00 1.000000001    1e-08 0.000000001     True
```

nan and `inf` are special cases.

```
# math_isclose_inf.py

import math

print('nan, nan:', math.isclose(math.nan, math.nan))
print('nan, 1.0:', math.isclose(math.nan, 1.0))
print('inf, inf:', math.isclose(math.inf, math.inf))
print('inf, 1.0:', math.isclose(math.inf, 1.0))
```

nan is never close to another value, including itself. `inf` is only close to itself.

```
$ python3 math_isclose_inf.py

nan, nan: False
nan, 1.0: False
inf, inf: True
inf, 1.0: False
```

Converting Floating Point Values to Integers

The `math` module includes three functions for converting floating point values to whole numbers. Each takes a different approach, and will be useful in different circumstances.

The simplest is `trunc()`, which truncates the digits following the decimal, leaving only the significant digits making up the whole number portion of the value. `floor()` converts its input to the largest preceding integer, and `ceil()` (ceiling) produces the largest integer following sequentially after the input value.

```
# math_integers.py

import math

HEADINGS = ('i', 'int', 'trunk', 'floor', 'ceil')
print('{:^5} {:^5} {:^5} {:^5} {:^5}'.format(*HEADINGS))
print('{:-^5} {:-^5} {:-^5} {:-^5} {:-^5}'.format(
    '', '', '', '', '',
))

fmt = '{:5.1f} {:5.1f} {:5.1f} {:5.1f} {:5.1f}'

TEST_VALUES = [
```

```
    -1.5,
    -0.8,
    -0.5,
    -0.2,
    0,
    0.2,
    0.5,
    0.8,
    1,
]

for i in TEST_VALUES:
    print(fmt.format(
        i,
        int(i),
        math.trunc(i),
        math.floor(i),
        math.ceil(i),
    ))
```

trunc() is equivalent to converting to int directly.

```
$ python3 math_integers.py

  i     int   trunk floor ceil
----- ----- ----- ----- -----
 -1.5  -1.0  -1.0  -2.0  -1.0
 -0.8   0.0   0.0  -1.0   0.0
 -0.5   0.0   0.0  -1.0   0.0
 -0.2   0.0   0.0  -1.0   0.0
  0.0   0.0   0.0   0.0   0.0
  0.2   0.0   0.0   0.0   1.0
  0.5   0.0   0.0   0.0   1.0
  0.8   0.0   0.0   0.0   1.0
  1.0   1.0   1.0   1.0   1.0
```

Alternate Representations of Floating Point Values

modf() takes a single floating point number and returns a tuple containing the fractional and whole number parts of the input value.

```
# math_modf.py

import math

for i in range(6):
    print('{}/2 = {}'.format(i, math.modf(i / 2.0)))
```

Both numbers in the return value are floats.

```
$ python3 math_modf.py

0/2 = (0.0, 0.0)
1/2 = (0.5, 0.0)
2/2 = (0.0, 1.0)
3/2 = (0.5, 1.0)
4/2 = (0.0, 2.0)
5/2 = (0.5, 2.0)
```

frexp() returns the mantissa and exponent of a floating point number, and can be used to create a more portable representation of the value.

```
# math_frexp.py

import math

print('{:^7} {:^7} {:^7}'.format('x', 'm', 'e'))
print('{:-^7} {:-^7} {:-^7}'.format('', '', ''))
```

```
for x in [0.1, 0.5, 4.0]:
    m, e = math.frexp(x)
    print('{:7.2f} {:7.2f} {:7d}'.format(x, m, e))
```

`frexp()` uses the formula `x = m * 2**e`, and returns the values `m` and `e`.

```
$ python3 math_frexp.py

   x       m       e
------- ------- -------
   0.10    0.80      -3
   0.50    0.50       0
   4.00    0.50       3
```

`ldexp()` is the inverse of `frexp()`.

```
# math_ldexp.py

import math

print('{:^7} {:^7} {:^7}'.format('m', 'e', 'x'))
print('{:-^7} {:-^7} {:-^7}'.format('', '', ''))

INPUTS = [
    (0.8, -3),
    (0.5, 0),
    (0.5, 3),
]

for m, e in INPUTS:
    x = math.ldexp(m, e)
    print('{:7.2f} {:7d} {:7.2f}'.format(m, e, x))
```

Using the same formula as `frexp()`, `ldexp()` takes the mantissa and exponent values as arguments and returns a floating point number.

```
$ python3 math_ldexp.py

   m       e       x
------- ------- -------
   0.80      -3    0.10
   0.50       0    0.50
   0.50       3    4.00
```

Positive and Negative Signs

The absolute value of a number is its value without a sign. Use `fabs()` to calculate the absolute value of a floating point number.

```
# math_fabs.py

import math

print(math.fabs(-1.1))
print(math.fabs(-0.0))
print(math.fabs(0.0))
print(math.fabs(1.1))
```

In practical terms, the absolute value of a `float` is represented as a positive value.

```
$ python3 math_fabs.py

1.1
0.0
0.0
1.1
```

To determine the sign of a value, either to give a set of values the same sign or to compare two values, use `copysign()` to set the sign of a known good value.

```
# math_copysign.py

import math

HEADINGS = ('f', 's', '< 0', '> 0', '= 0')
print('{:^5} {:^5} {:^5} {:^5} {:^5}'.format(*HEADINGS))
print('{:-^5} {:-^5} {:-^5} {:-^5} {:-^5}'.format(
    '', '', '', '', '',
))

VALUES = [
    -1.0,
    0.0,
    1.0,
    float('-inf'),
    float('inf'),
    float('-nan'),
    float('nan'),
]

for f in VALUES:
    s = int(math.copysign(1, f))
    print('{:5.1f} {:5d} {!s:5} {!s:5} {!s:5}'.format(
        f, s, f < 0, f > 0, f == 0,
    ))
```

An extra function like `copysign()` is needed because comparing nan and -nan directly with other values does not work.

```
$ python3 math_copysign.py

  f     s    < 0   > 0   = 0
----- ----- ----- ----- -----
 -1.0    -1 True  False False
  0.0     1 False False True
  1.0     1 False True  False
 -inf    -1 True  False False
  inf     1 False True  False
  nan    -1 False False False
  nan     1 False False False
```

Commonly Used Calculations

Representing precise values in binary floating point memory is challenging. Some values cannot be represented exactly, and the more often a value is manipulated through repeated calculations, the more likely a representation error will be introduced. `math` includes a function for computing the sum of a series of floating point numbers using an efficient algorithm that minimizes such errors.

```
# math_fsum.py

import math

values = [0.1] * 10

print('Input values:', values)

print('sum()       : {:.20f}'.format(sum(values)))

s = 0.0
for i in values:
    s += i
print('for-loop    : {:.20f}'.format(s))

print('math.fsum() : {:.20f}'.format(math.fsum(values)))
```

Given a sequence of ten values, each equal to `0.1`, the expected value for the sum of the sequence is `1.0`. Since `0.1` cannot be represented exactly as a floating point value, however, errors are introduced into the sum unless it is calculated with be represented exactly as a floating point value, however, errors are introduced into the sum unless it is calculated with `fsum()`.

```
$ python3 math_fsum.py

Input values: [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
sum()       : 0.99999999999999988898
for-loop    : 0.99999999999999988898
math.fsum() : 1.00000000000000000000
```

`factorial()` is commonly used to calculate the number of permutations and combinations of a series of objects. The factorial of a positive integer n, expressed `n!`, is defined recursively as `(n-1)! * n` and stops with `0! == 1`.

```
# math_factorial.py

import math

for i in [0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.1]:
    try:
        print('{:2.0f} {:6.0f}'.format(i, math.factorial(i)))
    except ValueError as err:
        print('Error computing factorial({}): {}'.format(i, err))
```

`factorial()` only works with whole numbers, but does accept `float` arguments as long as they can be converted to an integer without losing value.

```
$ python3 math_factorial.py

 0      1
 1      1
 2      2
 3      6
 4     24
 5    120
Error computing factorial(6.1): factorial() only accepts integral
 values
```

`gamma()` is like `factorial()`, except it works with real numbers and the value is shifted down by one (gamma is equal to `(n - 1)!`).

```
# math_gamma.py

import math

for i in [0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6]:
    try:
        print('{:2.1f} {:6.2f}'.format(i, math.gamma(i)))
    except ValueError as err:
        print('Error computing gamma({}): {}'.format(i, err))
```

Since zero causes the start value to be negative, it is not allowed.

```
$ python3 math_gamma.py

Error computing gamma(0): math domain error
1.1   0.95
2.2   1.10
3.3   2.68
4.4  10.14
5.5  52.34
6.6 344.70
```

`lgamma()` returns the natural logarithm of the absolute value of gamma for the input value.

```
# math_lgamma.py

import math

for i in [0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6]:
```

```
for i in [0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6]:
    try:
        print('{:2.1f} {:.20f} {:.20f}'.format(
            i,
            math.lgamma(i),
            math.log(math.gamma(i)),
        ))
    except ValueError as err:
        print('Error computing lgamma({}): {}'.format(i, err))
```

Using `lgamma()` retains more precision than calculating the logarithm separately using the results of `gamma()`.

```
$ python3 math_lgamma.py

Error computing lgamma(0): math domain error
1.1 -0.04987244125984036103 -0.04987244125983997245
2.2 0.09694746679063825923 0.09694746679063866168
3.3 0.98709857789473387513 0.98709857789473409717
4.4 2.31610349142485727469 2.31610349142485727469
5.5 3.95781396761871651080 3.95781396761871606671
6.6 5.84268005527463252236 5.84268005527463252236
```

The modulo operator (`%`) computes the remainder of a division expression (i.e., `5 % 2 = 1`). The operator built into the language works well with integers but, as with so many other floating point operations, intermediate calculations cause representational issues that result in a loss of data. `fmod()` provides a more accurate implementation for floating point values.

```
# math_fmod.py

import math

print('{:^4} {:^4} {:^5} {:^5}'.format(
    'x', 'y', '%', 'fmod'))
print('{:-^4} {:-^4} {:-^5} {:-^5}'.format(
    '-', '-', '-', '-'))

INPUTS = [
    (5, 2),
    (5, -2),
    (-5, 2),
]

for x, y in INPUTS:
    print('{:4.1f} {:4.1f} {:5.2f} {:5.2f}'.format(
        x,
        y,
        x % y,
        math.fmod(x, y),
    ))
```

A potentially more frequent source of confusion is the fact that the algorithm used by `fmod()` for computing modulo is also different from that used by `%`, so the sign of the result is different.

```
$ python3 math_fmod.py

 x    y      %    fmod
---- ---- ----- -----
 5.0  2.0  1.00  1.00
 5.0 -2.0 -1.00  1.00
-5.0  2.0  1.00 -1.00
```

Use `gcd()` to find the largest integer that can divide evenly into two integers, the greatest common divisor.

```
# math_gcd.py

import math

print(math.gcd(10, 8))
print(math.gcd(10, 0))
print(math.gcd(50, 225))
print(math.gcd(11, 9))
```

```
print(math.gcd(11, 9))
print(math.gcd(0, 0))
```

If both values are `0`, the result is `0`.

```
$ python3 math_gcd.py

2
10
25
1
0
```

Exponents and Logarithms

Exponential growth curves appear in economics, physics, and other sciences. Python has a built-in exponentiation operator (“**”), but `pow()` can be useful when a callable function is needed as an argument to another function.

```
# math_pow.py

import math

INPUTS = [
    # Typical uses
    (2, 3),
    (2.1, 3.2),

    # Always 1
    (1.0, 5),
    (2.0, 0),

    # Not-a-number
    (2, float('nan')),

    # Roots
    (9.0, 0.5),
    (27.0, 1.0 / 3),
]

for x, y in INPUTS:
    print('{:5.1f} ** {:5.3f} = {:6.3f}'.format(
        x, y, math.pow(x, y)))
```

Raising 1 to any power always returns `1.0`, as does raising any value to a power of `0.0`. Most operations on the not-a-number value nan return nan. If the exponent is less than 1, `pow()` computes a root.

```
$ python3 math_pow.py

  2.0 ** 3.000 =  8.000
  2.1 ** 3.200 = 10.742
  1.0 ** 5.000 =  1.000
  2.0 ** 0.000 =  1.000
  2.0 **   nan =    nan
  9.0 ** 0.500 =  3.000
 27.0 ** 0.333 =  3.000
```

Since square roots (exponent of 1/2) are used so frequently, there is a separate function for computing them.

```
# math_sqrt.py

import math

print(math.sqrt(9.0))
print(math.sqrt(3))
try:
    print(math.sqrt(-1))
except ValueError as err:
    print('Cannot compute sqrt(-1):', err)
```

Computing the square roots of negative numbers requires *complex numbers*, which are not handled by `math`. Any attempt to calculate a square root of a negative value results in a `ValueError`.

```
$ python3 math_sqrt.py

3.0
1.7320508075688772
Cannot compute sqrt(-1): math domain error
```

The logarithm function finds y where `x = b ** y`. By default, `log()` computes the natural logarithm (the base is *e*). If a second argument is provided, that value is used as the base.

```
# math_log.py

import math

print(math.log(8))
print(math.log(8, 2))
print(math.log(0.5, 2))
```

Logarithms where x is less than one yield negative results.

```
$ python3 math_log.py

2.0794415416798357
3.0
-1.0
```

There are three variations of `log()`. Given floating point representation and rounding errors, the computed value produced by `log(x, b)` has limited accuracy, especially for some bases. `log10()` computes `log(x, 10)`, using a more accurate algorithm than `log()`.

```
# math_log10.py

import math

print('{:2} {:^12} {:^10} {:^20} {:8}'.format(
    'i', 'x', 'accurate', 'inaccurate', 'mismatch',
))
print('{:-^2} {:-^12} {:-^10} {:-^20} {:-^8}'.format(
    '', '', '', '', '',
))

for i in range(0, 10):
    x = math.pow(10, i)
    accurate = math.log10(x)
    inaccurate = math.log(x, 10)
    match = '' if int(inaccurate) == i else '*'
    print('{:2d} {:12.1f} {:10.8f} {:20.18f} {:^5}'.format(
        i, x, accurate, inaccurate, match,
    ))
```

The lines in the output with trailing * highlight the inaccurate values.

```
$ python3 math_log10.py

i       x        accurate       inaccurate      mismatch
-- ------------ ---------- -------------------- --------
 0          1.0 0.00000000 0.000000000000000000
 1         10.0 1.00000000 1.000000000000000000
 2        100.0 2.00000000 2.000000000000000000
 3       1000.0 3.00000000 2.999999999999999556   *
 4      10000.0 4.00000000 4.000000000000000000
 5     100000.0 5.00000000 5.000000000000000000
 6    1000000.0 6.00000000 5.999999999999999112   *
 7   10000000.0 7.00000000 7.000000000000000000
 8  100000000.0 8.00000000 8.000000000000000000
 9 1000000000.0 9.00000000 8.999999999999998224   *
```

Similar to `log10()`, `log2()` calculates the equivalent of `math.log(x, 2)`.

```
# math_log2.py

import math

print('{:>2} {:^5} {:^5}'.format(
    'i', 'x', 'log2',
))
print('{:-^2} {:-^5} {:-^5}'.format(
    '', '', '',
))

for i in range(0, 10):
    x = math.pow(2, i)
    result = math.log2(x)
    print('{:2d} {:5.1f} {:5.1f}'.format(
        i, x, result,
    ))
```

Depending on the underlying platform, using the built-in and special-purpose function can offer better performance and accuracy by using special-purpose algorithms for base 2 that are not found in the more general purpose function.

```
$ python3 math_log2.py

 i   x   log2
-- ----- -----
 0   1.0   0.0
 1   2.0   1.0
 2   4.0   2.0
 3   8.0   3.0
 4  16.0   4.0
 5  32.0   5.0
 6  64.0   6.0
 7 128.0   7.0
 8 256.0   8.0
 9 512.0   9.0
```

`log1p()` calculates the Newton-Mercator series (the natural logarithm of 1+x).

```
# math_log1p.py

import math

x = 0.0000000000000000000000001
print('x       :', x)
print('1 + x   :', 1 + x)
print('log(1+x):', math.log(1 + x))
print('log1p(x):', math.log1p(x))
```

`log1p()` is more accurate for values of x very close to zero because it uses an algorithm that compensates for round-off errors from the initial addition.

```
$ python3 math_log1p.py

x       : 1e-25
1 + x   : 1.0
log(1+x): 0.0
log1p(x): 1e-25
```

`exp()` computes the exponential function (e**x).

```
# math_exp.py

import math

x = 2
```

```
fmt = '{:.20f}'
print(fmt.format(math.e ** 2))
print(fmt.format(math.pow(math.e, 2)))
print(fmt.format(math.exp(2)))
```

As with other special-case functions, it uses an algorithm that produces more accurate results than the general-purpose equivalent `math.pow(math.e, x)`.

```
$ python3 math_exp.py

7.38905609893064951876
7.38905609893064951876
7.38905609893065040694
```

`expm1()` is the inverse of `log1p()`, and calculates `e**x - 1`.

```
# math_expm1.py

import math

x = 0.0000000000000000000000001

print(x)
print(math.exp(x) - 1)
print(math.expm1(x))
```

Small values of x lose precision when the subtraction is performed separately, like with `log1p()`.

```
$ python3 math_expm1.py

1e-25
0.0
1e-25
```

Angles

Although degrees are more commonly used in everyday discussions of angles, radians are the standard unit of angular measure in science and math. A radian is the angle created by two lines intersecting at the center of a circle, with their ends on the circumference of the circle spaced one radius apart.

The circumference is calculated as $2\pi r$, so there is a relationship between radians and π, a value that shows up frequently in trigonometric calculations. That relationship leads to radians being used in trigonometry and calculus, because they result in more compact formulas.

To convert from degrees to radians, use `radians()`.

```
# math_radians.py

import math

print('{:^7} {:^7} {:^7}'.format(
    'Degrees', 'Radians', 'Expected'))
print('{:-^7} {:-^7} {:-^7}'.format(
    '', '', ''))

INPUTS = [
    (0, 0),
    (30, math.pi / 6),
    (45, math.pi / 4),
    (60, math.pi / 3),
    (90, math.pi / 2),
    (180, math.pi),
    (270, 3 / 2.0 * math.pi),
    (360, 2 * math.pi),
]

for deg, expected in INPUTS:
    print('{:7d} {:7.2f} {:7.2f}'.format(
```

```
        deg,
        math.radians(deg),
        expected,
    ))
```

The formula for the conversion is `rad = deg * π / 180`.

```
$ python3 math_radians.py

Degrees Radians Expected
------- ------- -------
      0    0.00    0.00
     30    0.52    0.52
     45    0.79    0.79
     60    1.05    1.05
     90    1.57    1.57
    180    3.14    3.14
    270    4.71    4.71
    360    6.28    6.28
```

To convert from radians to degrees, use `degrees()`.

```
# math_degrees.py

import math

INPUTS = [
    (0, 0),
    (math.pi / 6, 30),
    (math.pi / 4, 45),
    (math.pi / 3, 60),
    (math.pi / 2, 90),
    (math.pi, 180),
    (3 * math.pi / 2, 270),
    (2 * math.pi, 360),
]

print('{:^8} {:^8} {:^8}'.format(
    'Radians', 'Degrees', 'Expected'))
print('{:-^8} {:-^8} {:-^8}'.format('', '', ''))
for rad, expected in INPUTS:
    print('{:8.2f} {:8.2f} {:8.2f}'.format(
        rad,
        math.degrees(rad),
        expected,
    ))
```

The formula is `deg = rad * 180 / π`.

```
$ python3 math_degrees.py

Radians  Degrees  Expected
-------- -------- --------
    0.00     0.00     0.00
    0.52    30.00    30.00
    0.79    45.00    45.00
    1.05    60.00    60.00
    1.57    90.00    90.00
    3.14   180.00   180.00
    4.71   270.00   270.00
    6.28   360.00   360.00
```

Trigonometry

Trigonometric functions relate angles in a triangle to the lengths of its sides. They show up in formulas with periodic properties such as harmonics, circular motion, or when dealing with angles. All of the trigonometric functions in the standard library take angles expressed as radians.

Given an angle in a right triangle, the *sine* is the ratio of the length of the side opposite the angle to the hypotenuse (`sin A =`

Given an angle in a right triangle, the *sine* is the ratio of the length of the side opposite the angle to the hypotenuse (sin A = opposite/hypotenuse). The *cosine* is the ratio of the length of the adjacent side to the hypotenuse (cos A = adjacent/hypotenuse). And the *tangent* is the ratio of the opposite side to the adjacent side (tan A = opposite/adjacent).

```
# math_trig.py

import math

print('{:^7} {:^7} {:^7} {:^7} {:^7}'.format(
    'Degrees', 'Radians', 'Sine', 'Cosine', 'Tangent'))
print('{:-^7} {:-^7} {:-^7} {:-^7} {:-^7}'.format(
    '-', '-', '-', '-', '-'))

fmt = '{:7.2f} {:7.2f} {:7.2f} {:7.2f} {:7.2f}'

for deg in range(0, 361, 30):
    rad = math.radians(deg)
    if deg in (90, 270):
        t = float('inf')
    else:
        t = math.tan(rad)
    print(fmt.format(deg, rad, math.sin(rad), math.cos(rad), t))
```

The tangent can also be defined as the ratio of the sine of the angle to its cosine, and since the cosine is 0 for π/2 and 3π/2 radians, the tangent is infinite.

```
$ python3 math_trig.py

Degrees Radians  Sine   Cosine  Tangent
------- ------- ------- ------- -------
   0.00    0.00    0.00    1.00    0.00
  30.00    0.52    0.50    0.87    0.58
  60.00    1.05    0.87    0.50    1.73
  90.00    1.57    1.00    0.00     inf
 120.00    2.09    0.87   -0.50   -1.73
 150.00    2.62    0.50   -0.87   -0.58
 180.00    3.14    0.00   -1.00   -0.00
 210.00    3.67   -0.50   -0.87    0.58
 240.00    4.19   -0.87   -0.50    1.73
 270.00    4.71   -1.00   -0.00     inf
 300.00    5.24   -0.87    0.50   -1.73
 330.00    5.76   -0.50    0.87   -0.58
 360.00    6.28   -0.00    1.00   -0.00
```

Given a point (x, y), the length of the hypotenuse for the triangle between the points [(0, 0), (x, 0), (x, y)] is (x**2 + y**2) ** 1/2, and can be computed with hypot().

```
# math_hypot.py

import math

print('{:^7} {:^7} {:^10}'.format('X', 'Y', 'Hypotenuse'))
print('{:-^7} {:-^7} {:-^10}'.format('', '', ''))

POINTS = [
    # simple points
    (1, 1),
    (-1, -1),
    (math.sqrt(2), math.sqrt(2)),
    (3, 4),  # 3-4-5 triangle
    # on the circle
    (math.sqrt(2) / 2, math.sqrt(2) / 2),  # pi/4 rads
    (0.5, math.sqrt(3) / 2),  # pi/3 rads
]

for x, y in POINTS:
    h = math.hypot(x, y)
    print('{:7.2f} {:7.2f} {:7.2f}'.format(x, y, h))
```

Points on the circle always have hypotenuse equal to 1.

```
$ python3 math_hypot.py

   X       Y    Hypotenuse
------- ------- ----------
   1.00    1.00    1.41
  -1.00   -1.00    1.41
   1.41    1.41    2.00
   3.00    4.00    5.00
   0.71    0.71    1.00
   0.50    0.87    1.00
```

The same function can be used to find the distance between two points.

```
# math_distance_2_points.py

import math

print('{:^8} {:^8} {:^8} {:^8} {:^8}'.format(
    'X1', 'Y1', 'X2', 'Y2', 'Distance',
))
print('{:-^8} {:-^8} {:-^8} {:-^8} {:-^8}'.format(
    '', '', '', '', '',
))

POINTS = [
    ((5, 5), (6, 6)),
    ((-6, -6), (-5, -5)),
    ((0, 0), (3, 4)),  # 3-4-5 triangle
    ((-1, -1), (2, 3)),  # 3-4-5 triangle
]

for (x1, y1), (x2, y2) in POINTS:
    x = x1 - x2
    y = y1 - y2
    h = math.hypot(x, y)
    print('{:8.2f} {:8.2f} {:8.2f} {:8.2f} {:8.2f}'.format(
        x1, y1, x2, y2, h,
    ))
```

Use the difference in the x and y values to move one endpoint to the origin, and then pass the results to `hypot()`.

```
$ python3 math_distance_2_points.py

   X1       Y1       X2       Y2    Distance
-------- -------- -------- -------- --------
    5.00     5.00     6.00     6.00     1.41
   -6.00    -6.00    -5.00    -5.00     1.41
    0.00     0.00     3.00     4.00     5.00
   -1.00    -1.00     2.00     3.00     5.00
```

`math` also defines inverse trigonometric functions.

```
# math_inverse_trig.py

import math

for r in [0, 0.5, 1]:
    print('arcsine({:.1f})    = {:5.2f}'.format(r, math.asin(r)))
    print('arccosine({:.1f})  = {:5.2f}'.format(r, math.acos(r)))
    print('arctangent({:.1f}) = {:5.2f}'.format(r, math.atan(r)))
    print()
```

1.57 is roughly equal to π/2, or 90 degrees, the angle at which the sine is 1 and the cosine is 0.

```
$ python3 math_inverse_trig.py

arcsine(0.0)    =  0.00
arccosine(0.0)  =  1.57
```

```
arccosine(0.0)    =  1.57
arctangent(0.0) =  0.00

arcsine(0.5)    =  0.52
arccosine(0.5)  =  1.05
arctangent(0.5) =  0.46

arcsine(1.0)    =  1.57
arccosine(1.0)  =  0.00
arctangent(1.0) =  0.79
```

Hyperbolic Functions

Hyperbolic functions appear in linear differential equations and are used when working with electromagnetic fields, fluid dynamics, special relativity, and other advanced physics and mathematics.

```
# math_hyperbolic.py

import math

print('{:^6} {:^6} {:^6} {:^6}'.format(
    'X', 'sinh', 'cosh', 'tanh',
))
print('{:-^6} {:-^6} {:-^6} {:-^6}'.format('', '', '', ''))

fmt = '{:6.4f} {:6.4f} {:6.4f} {:6.4f}'

for i in range(0, 11, 2):
    x = i / 10.0
    print(fmt.format(
        x,
        math.sinh(x),
        math.cosh(x),
        math.tanh(x),
    ))
```

Whereas the cosine and sine functions enscribe a circle, the hyperbolic cosine and hyperbolic sine form half of a hyperbola.

```
$ python3 math_hyperbolic.py

  X     sinh   cosh   tanh
------ ------ ------ ------
0.0000 0.0000 1.0000 0.0000
0.2000 0.2013 1.0201 0.1974
0.4000 0.4108 1.0811 0.3799
0.6000 0.6367 1.1855 0.5370
0.8000 0.8881 1.3374 0.6640
1.0000 1.1752 1.5431 0.7616
```

Inverse hyperbolic functions `acosh()`, `asinh()`, and `atanh()` are also available.

Special Functions

The Gauss Error function is used in statistics.

```
# math_erf.py

import math

print('{:^5} {:7}'.format('x', 'erf(x)'))
print('{:-^5} {:-^7}'.format('', ''))

for x in [-3, -2, -1, -0.5, -0.25, 0, 0.25, 0.5, 1, 2, 3]:
    print('{:5.2f} {:7.4f}'.format(x, math.erf(x)))
```

For the error function, `erf(-x) == -erf(x)`.

```
$ python3 math_erf.py
```

```
  x   erf(x)
----- -------
-3.00 -1.0000
-2.00 -0.9953
-1.00 -0.8427
-0.50 -0.5205
-0.25 -0.2763
 0.00  0.0000
 0.25  0.2763
 0.50  0.5205
 1.00  0.8427
 2.00  0.9953
 3.00  1.0000
```

The complimentary error function is `1 - erf(x)`.

```
# math_erfc.py

import math

print('{:^5} {:7}'.format('x', 'erfc(x)'))
print('{:-^5} {:-^7}'.format('', ''))

for x in [-3, -2, -1, -0.5, -0.25, 0, 0.25, 0.5, 1, 2, 3]:
    print('{:5.2f} {:7.4f}'.format(x, math.erfc(x)))
```

The implementation of `erfc()` avoids precision errors for small values of x when subtracting from 1.

```
$ python3 math_erfc.py

  x   erfc(x)
----- -------
-3.00  2.0000
-2.00  1.9953
-1.00  1.8427
-0.50  1.5205
-0.25  1.2763
 0.00  1.0000
 0.25  0.7237
 0.50  0.4795
 1.00  0.1573
 2.00  0.0047
 3.00  0.0000
```

See also

- Standard library documentation for math
- IEEE floating point arithmetic in Python – Blog post by John Cook about how special values arise and are dealt with when doing math in Python.
- SciPy – Open source libraryes for scientific and mathematical calculations in Python.
- **PEP 485** – “A function for testing approximate equality”

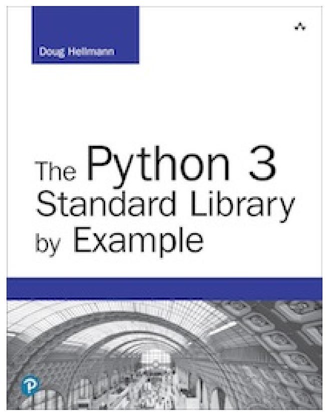
Quick Links

Special Constants
Testing for Exceptional Values
Comparing
Converting Floating Point Values to Integers
Alternate Representations of Floating Point Values
Positive and Negative Signs
Commonly Used Calculations
Exponents and Logarithms
Angles
Trigonometry
Hyperbolic Functions
Special Functions

This page was last updated 2016-12-28.

Navigation

random — Pseudorandom Number Generators
statistics — Statistical Calculations



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example